```
// Kazahana revision 1-++fix+nowait_critical_nixFIX_Wolfram+fixITER+EX+CS_fix_DEFINE, copyleft Kaze 2014-Dec-04.
// SUPERNASTY compiler-side mishavoc fixed (by lowering demands to static arrays with DEFINE 126+100 becoming 156), it appeared as random crashing/overwriting variables and general mayhem.
// Kazahana revision 1-++fix+nowait_critical_nixFIX_Wolfram+fixITER+EX+CS_fix, copyleft Kaze 2014-Nov-19.
// Fixed a stupid parsing bug causing FuzzyExhaustive to search in lines only up to 326 chars:
// MAXboth = MaxLineLength +1+1 +(167*WILDCARD_IP_flag*MaxLineLength); // Buggy line, fixed with next one in r. ...CS_fix
//         if (WILDCARD_IP_flag) {
//                   MAXboth = MaxLineLength +1+1 +(167*WILDCARD_IP_flag*MaxLineLength);
//         } else {
//                   MAXboth = MaxLineLength +1+1 +(167*EXHAUSTIVE_flag*MaxLineLength);
//         }
// Kazahana revision 1-++fix+nowait_critical_nixFIX_Wolfram+fixITER+EX+CS, copyleft Kaze 2014-Mar-25.
// Kazahana revision 1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER+EX (MinGW ready, Linux ready), copyleft Sanmayce 2013-Dec-10.
// Kazahana revision 1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER+EX (MinGW ready, Linux ready), copyleft Sanmayce 2013-Dec-05.
// Kazahana revision 1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER+ (MinGW ready, Linux ready), copyleft Sanmayce 2013-Nov-30.
// Kazahana revision 1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER (MinGW ready, Linux ready), copyleft Sanmayce 2013-Nov-29.
// Kazahana revision 1-++fix+nowait_critical_nixFIX_WolfRAM+fix (MinGW ready, Linux ready), copyleft Sanmayce 2013-Nov-24.
// Stupid bug was crushed: 'unsigned int' became AGAIN 'int' as it was in Galadriel, simply forgot that it can be negative.
// Kazahana revision 1-++fix+nowait_critical_nixFIX_WolfRAM+ (MinGW ready, Linux ready), copyleft Sanmayce 2013-Nov-21.
// Kazahana revision 1-++fix+nowait_critical_nixFIX_WolfRAM (MinGW ready, Linux ready), copyleft Sanmayce 2013-Nov-15.
// Kazahana revision 1-++fix+nowait_critical_nixFIX_Bari (MinGW ready, Linux ready), copyleft Sanmayce 2013-Oct-23.

// Kazahana revision 1-++fix+nowait_critical_nixFIX (MinGW ready, Linux ready), copyleft Sanmayce 2013-Apr-07.
// Grr... a leftover/overlooked parsing bug was crushed.

// Kazahana revision 1-++fix+nowait_critical_nix (MinGW ready, Linux ready), copyleft Sanmayce 2013-Feb-24.
// TO-DO in r.1: recursive calls to be simulated with my own stack.
// In this revision 16 threads are enforced.
// If you want to help me to port it to *nix your name will appear as contributor in the credit part.
// Please give me a buzz (sanmayce@sanmayce.com) if you find faster implementation.
// Special thanks go to Igor Pavlov, VIVA.
// Enfun!

// How to compile under Windows using Intel compiler:
// icl /Ox Kazahana_r1-++fix+nowait_critical_nix.c /FAcs /FeKazahana_r1-++fix+nowait_critical_nix_HEXADECAD-Threads_IntelV12 /Qopenmp /Qopenmp-link:static -DCommence_OpenMP -D_icl_mumbo_jumbo_
// icl /Ox Kazahana_r1-++fix+nowait_critical_nix.c /FeKazahana_r1-++fix+nowait_critical_nix_MONAD-Thread_IntelV12 -D_icl_mumbo_jumbo_

// How to compile under Windows using MinGW:
// gcc -O3 -funroll-loops -static -std=c99 -o Kazahana_r1-++fix+nowait_critical_nix_GCC_472 Kazahana_r1-++fix+nowait_critical_nix.c -fopenmp -DCommence_OpenMP -D_gcc_mumbo_jumbo_

// Change accordingly from command line:
//#define _icl_mumbo_jumbo_
//#define _gcc_mumbo_jumbo_

// Change appropriately:
#define _WIN32_ENVIRONMENT_
//#define _POSIX_ENVIRONMENT_

#define _WildFastKaze_

// If you comment next, then the light-weight and faster on small haystacks 'Railgun_Quadruplet_7' will take over:
#define RG7Gulliver
// In fact Bari replaces Gulliver.

// How much MB the master-buffer will be? My tests show that 7 is a very good (but not excellent) value on my 4MB cache T7500.
// 11, 14, 19 are good values as they are one less than L3 cache of fast CPUs.
// Since Wikipedia has got some very long lines, 7 is also the minimal one if you want to search in her.
//#define MasterBuffer 7

/*
hatsutoukou : first (written) contribution
hatsuyuki : first snow (of season)
hatsuyume : year's first dream

fubuki : snow storm

amenochiyuki : rain then snow

fubon : uncommon, outstanding
fubuki : snow storm
```

```
fubun : unwritten, illiterate, uneducated

hyouden : field of eternal snow

koyuki : light snow
ooyuki : heavy snow

setsuzou : snow sculpture

shinshin : sound of heavy snow-fall
shinshin : mind body

yukionna : snow woman, fairy
*/

// [[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[
// Modified Igor Pavlov's wildcard.cpp from 7zip package wildcard code [

#ifndef NULL
#ifdef __cplusplus
#define NULL 0
#else
#define NULL ((void*)0)
#endif
#endif

#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
#define TRUE 1
#endif

#ifndef false
#define false 0
#endif
#ifndef true
#define true 1
#endif

typedef unsigned char char_t;
typedef char_t *string;
char TAGfree[10] = "*@#^$|~`.";
char TAGfreeFAST[3] = "&+"; // '&' stands for "standard" '*', '+' stands for "standard" '?'.
typedef unsigned char boolean;

#if defined(_icl_mumbo_jumbo_)
// GetRDTSC() taken from strchr.com
#if defined(_M_IX86)
unsigned long long __forceinline GetRDTSC(void) {
   __asm {
      ; Flush the pipeline
      XOR eax, eax
      CPUID
      ; Get RDTSC counter in edx:eax
      RDTSC
   }
}
#elif defined(_M_X64)
unsigned long long __forceinline GetRDTSC(void) {
           return __rdtsc();
}
#else
unsigned long long __forceinline GetRDTSC(void) {
           return GetTickCount();
}
#endif
#endif
```

```
#if defined(_gcc_mumbo_jumbo_)

#endif

#define KAZE_tolower(c) ( (((c) >= 'A') && ((c) <= 'Z')) ? ((c) - 'A' + 'a') : (c) )
#define KAZE_toupper(c) ( (((c) >= 'a') && ((c) <= 'z')) ? ((c) - 'a' + 'A') : (c) )

int maskGLOBALlen;
int nameGLOBALlen1;
int nameGLOBALlen2;
int nameGLOBALlen3;
int nameGLOBALlen4;
int nameGLOBALlen5;
int nameGLOBALlen6;
int nameGLOBALlen7;
int nameGLOBALlen8;
int nameGLOBALlen9;
int nameGLOBALlen0;
int nameGLOBALlena;
int nameGLOBALlenb;
int nameGLOBALlenc;
int nameGLOBALlend;
int nameGLOBALlene;
int nameGLOBALlenf;

int CaseSensitiveWildcardMatching_flag; // ZERO for INSENSITIVE, NONZERO for SENSITIVE

//long VIVA_IgorPavlov_invocations_global_counter = 0;
//long WildGLOBALhits = 0;

#include <stdint.h> // Needed for uint32_t
//typedef unsigned char uint8_t;
//typedef unsigned short uint16_t;
//typedef unsigned int uint32_t;

#define ASIZE 256
// For speed up next 3 arrays are global:
unsigned int bm_bc[256]; //BMH needed
unsigned int bm_bc2nd[256]; //BMS needed

// Railgun_Sekireigan_Wolfram, copyleft 2013-Nov-11, Kaze.
// Do you know what is really COOL?
// Wolfram, if you ask me, with melting point of 3,410 Celsius.
// tungsten
// n. Symbol W
// A hard, brittle, corrosion-resistant, gray to white metallic element extracted from wolframite, scheelite, and other minerals, having the highest melting point and lowest vapor pressure of any metal.
// Tungsten and its alloys are used in high-temperature structural materials; in electrical elements, notably lamp filaments; and in instruments requiring thermally compatible glass-to-metal seals.
// Atomic number 74; atomic weight 183.84; melting point 3,410°C; boiling point 5,900°C; specific gravity 19.3 (20°C); valence 2, 3, 4, 5, 6. Also called wolfram. See Table at element.
// /Heritage/
#define _rotl_KAZE(x, n) (((x) << (n)) | ((x) >> (32-(n))))
#define HaystackThresholdSekireiTchittoGritto 961 // Quadruplet works up to this value, if bigger then BMH2 takes over.
#define NeedleThreshold2vs4TchittoGritto 22 // Should be bigger than 8. BMH2 works up to this value (inclusive), if bigger then BMH4 takes over.
#define NeedleThresholdBIGSekireiTchittoGritto 12+700 // Should be bigger than 'HasherezadeOrder'. BMH2 works up to this value (inclusive).
#define HashTableSizeSekireiTchittoGritto 17-1 // In fact the real size is -3, because it is BITwise, when 17-3=14 it means 16KB, (17-1)-3=13 it means 8KB.
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.

unsigned char bm_Horspool_Order2[256*256]; //BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
unsigned char bm_Hasherezade_HASH[1<<(HashTableSizeSekireiTchittoGritto-3)];

char * Railgun_Sekireigan_Wolfram_1 (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
{
            char * pbTargetMax = pbTarget + cbTarget;
            register uint32_t ulHashPattern;
            register uint32_t ulHashTarget;
            signed long count;
            //signed long countSTATIC;

            unsigned char SINGLET;
            uint32_t Quadruplet2nd;
```

```
	uint32_t Quadruplet3rd;
	uint32_t Quadruplet4th;

	uint32_t AdvanceHopperGrass;

	uint32_t a, i, j;
//Global is next line already:
	//unsigned char bm_Horspool_Order2[256*256]; // BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
	uint32_t Gulliver; // or unsigned char or unsigned short

	//unsigned char bm_Hasherezade_HASH[1<<(HashTableSizeSekireiTchittoGritto-3)];
	uint32_t hash32;
	uint32_t hash32B;
	uint32_t hash32C;

	if (cbPattern > cbTarget) return(NULL);

	if ( cbPattern<4 ) {

	pbTarget = pbTarget+cbPattern;
		ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
		if ( cbPattern==3 ) {
			for ( ;; ) {
				if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
					if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
				}
				if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) {
					pbTarget++;
					if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
				}
				pbTarget++;
				if (pbTarget > pbTargetMax) return(NULL);
			}
		} else {
		}
		for ( ;; ) {
			if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) ) return((pbTarget-2));
			if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
			pbTarget++;
			if (pbTarget > pbTargetMax) return(NULL);
		}

	} else {
		if (cbTarget<HaystackThresholdSekireiTchittoGritto) { // This value is arbitrary (don't know how exactly), it ensures (at least must) better performance than 'Boyer_Moore_Horspool'.

		pbTarget = pbTarget+cbPattern;
		ulHashPattern = *(uint32_t *)(pbPattern);
		SINGLET = ulHashPattern & 0xFF;
		Quadruplet2nd = SINGLET<<8;
		Quadruplet3rd = SINGLET<<16;
		Quadruplet4th = SINGLET<<24;
		for ( ;; ) {
			AdvanceHopperGrass = 0;
			ulHashTarget = *(uint32_t *)(pbTarget-cbPattern);
			if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
				count = cbPattern-1;
				while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
					if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
					count--;
				}
				if ( count == 0) return((pbTarget-cbPattern));
			} else { // The goal here: to avoid memory accesses by stressing the registers.
				if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
					AdvanceHopperGrass++;
					if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
						AdvanceHopperGrass++;
						if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
					}
				}
```

```
                                }
                                AdvanceHopperGrass++;
                                pbTarget = pbTarget + AdvanceHopperGrass;
                                if (pbTarget > pbTargetMax) return(NULL);
                        }

                        } else { //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
            if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto ) {

            // BMH order 2:
            if ( cbPattern<=NeedleThreshold2vs4TchittoGritto ) {
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(uint32_t *)(pbPattern+cbPattern-4); // Last four bytes
                                i=0;
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
//Possible commenting of next line:
                                for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=1;
                                while (i <= cbTarget-cbPattern) {
                                        Gulliver = 1; // 'Gulliver' is the skip
            // Few thoughts regarding an excellent Skip Performance etude:
            // Something "still" considered crazy: using BITwise order 3, not pseudo order 3, though!
            // 2^24 = 16MB BYTEwise or 2^(24-3) = 2MB BITwise, when searching big haystacks e.g. Wikipedia 42GB with Kazahana then this 2MB lookup table seems not so atrocious.
            // The code is like:
            // ulHashTarget = *(uint32_t *)&pbTarget[i+cbPattern-4]; // One memory access instead of 2
            // if ( bm_Horspool_Order3[ulHashTarget>>8] == 0 ) Gulliver = cbPattern-(3-1); else
            // if ( bm_Horspool_Order3[ulHashTarget&0xFFFFFF] == 0 ) Gulliver = cbPattern-(3-1)-1; else
            // {
            // ...
            // }
                                        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]] != 0 ) {
                                                if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-2]] + bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-1]] != 2 ) Gulliver = cbPattern-(2-1)-2; else {
                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for remainder) when going under 0 in loop below:
                                                        // Order 4 [
                                                // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
                                                //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                        count = cbPattern-4+1;
                                                        //count = count-4; // Double-beauty here of already being checked 'ulHashTarget' and not polluting/repeating the final lookup below.
                                                        while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                        if ( count <= 0 ) {
                                                                return(pbTarget+i);
                                                        }
                                                        //if ( count <= 0 ) {
                                                        //        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                        //}
                                                        //else {
                                                        //        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+count-1]] == 0 ) Gulliver = count; // 1 or bigger, as it should
                                                        //}
                                                        // Order 4 ]
                                                        }
                                                } // Means AT LEAST one of the BBs is 0, enforce lower skip: MIN( cbPattern-(2-1)-1 , cbPattern-(2-1)-2 )
                                        } else Gulliver = cbPattern-(2-1);
                                        i = i + Gulliver;
                                        //GlobalI++; // Comment it, it is only for stats.
                                }
                                return(NULL);
            // BMH order 4, needle should be >=8:
```

```
	} else { //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )
		//countSTATIC = cbPattern-2-2;
		ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
		//ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
		i=0;
		//for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
		//for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
		//for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
		// In line below we "hash" 4bytes to 2bytes i.e. 16bit table, how to compute TOTAL number of BBs, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox'
and Order=4 we have BBs = 11-4+1=8:
		//"fast"
		//"aste"
		//"stes"
		//"test"
		//"est "
		//"st f"
		//"t fo"
		//" fox"
		//for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( *(unsigned short *)(pbPattern+j+0) + *(unsigned short *)(pbPattern+j+2) ) & ( (1<<16)-1 )]=1;
//Possible commenting of next line:
		for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( (*(uint32_t *)(pbPattern+j+0)>>16)+(*(uint32_t *)(pbPattern+j+0)&0xFFFF) ) & ( (1<<16)-1 )]=1;
		while (i <= cbTarget-cbPattern) {
			Gulliver = 1;
			if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]&0xFFFF) ) & ( (1<<16)-1 )] != 0 ) {
				if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = cbPattern-(2-1)-2-4; else {
					// Order 4 [
				// Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
				// Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:

				//0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
				//1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
				//2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
				//3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
				//4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
				//5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
				//6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
				//7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
					count = cbPattern-4+1;
					while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
						count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
					if ( count <= 0 ) {
						if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
					}
					//else {
					//	if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+count-1]>>16)+(*(uint32_t *)&pbTarget[i+count-1]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = count; // 1 or
bigger, as it should

					//}
					// Order 4 ]
				}
			} else Gulliver = cbPattern-(2-1)-2; // -2 because we check the 4 rightmost bytes not 2.
			i = i + Gulliver;
			//GlobalI++; // Comment it, it is only for stats.
		}
		return(NULL);
	} //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )

	} else { // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() [
		//countSTATIC = cbPattern-2-2;
		ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
		//ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
		i=0;
		for (a=0; a < 1<<(HashTableSizeSekireiTchittoGritto-3); a++) {bm_Hasherezade_HASH[a]= 0;} // to-do: 'memset' if not optimized
		// cbPattern - Order + 1 i.e. number of BBs for 11 'fastest fox' 11-8+1=4: 'fastest ', 'astest f', 'stest fo', 'test fox'
		for (j=0; j < cbPattern-12+1; j++) {
			hash32 = (2166136261UL ^ *(uint32_t *)(pbPattern+j+0)) * 709607;
			hash32B = (2166136261UL ^ *(uint32_t *)(pbPattern+j+4)) * 709607;
			hash32C = (2166136261UL ^ *(uint32_t *)(pbPattern+j+8)) * 709607;
```

```
                                        hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                        hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                        hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                        bm_Hasherezade_HASH[hash32>>3]= bm_Hasherezade_HASH[hash32>>3] | (1<<(hash32&0x7));
                                }
                                while (i <= cbTarget-cbPattern) {
                                        Gulliver = 1; // Assume minimal jump as initial value.
                                        // The goal: to jump when the rightmost 8bytes (Order 8 Horspool) of window do not look like any of Needle prefixes i.e. are not to be found. This maximum jump equals cbPattern-(Order-1) or 11-(8-1)=4 for
'fastest fox' - a small one but for Needle 31 bytes the jump equals 31-(8-1)=24
                                        //GlobalHashSectionExecution++; // Comment it, it is only for stats.
                                                hash32 = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+0)) * 709607;
                                                hash32B = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+4)) * 709607;
                                                hash32C = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+8)) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                if ( (bm_Hasherezade_HASH[hash32>>3] & (1<<(hash32&0x7))) ==0 )     Gulliver = cbPattern-(12-1);
                                                else {
                                                //if ( Gulliver == 1 ) { // Means the Building-Block order 8/12 is found somewhere i.e. NO MAXIMUM SKIP
                                                                // Order 4 [
                                                        // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                        // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:

                                                        //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                        //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                        //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                        //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                        //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                        //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                        //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                        //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                count = cbPattern-4+1;
                                                                while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                        count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                if ( count <= 0 ) {
                                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                }
                                                                // Order 4 ]
                                                }
                                        i = i + Gulliver;
                                        //GlobalI++; // Comment it, it is only for stats.
                                } // while (i <= cbTarget-cbPattern)
                                return(NULL);
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() ]
        } // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
                } //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
        } //if ( cbPattern<4 )
}
char * Railgun_Sekireigan_Wolfram_2 (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
{
        char * pbTargetMax = pbTarget + cbTarget;
        register uint32_t ulHashPattern;
        register uint32_t ulHashTarget;
        signed long count;
        //signed long countSTATIC;

        unsigned char SINGLET;
        uint32_t Quadruplet2nd;
        uint32_t Quadruplet3rd;
        uint32_t Quadruplet4th;

        uint32_t AdvanceHopperGrass;

        uint32_t a, i, j;
//Global is next line already:
        //unsigned char bm_Horspool_Order2[256*256]; // BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
        uint32_t Gulliver; // or unsigned char or unsigned short

        //unsigned char bm_Hasherezade_HASH[1<<(HashTableSizeSekireiTchittoGritto-3)];
```

```
	uint32_t hash32;
	uint32_t hash32B;
	uint32_t hash32C;

	if (cbPattern > cbTarget) return(NULL);

	if ( cbPattern<4 ) {

	pbTarget = pbTarget+cbPattern;
		ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
		if ( cbPattern==3 ) {
			for ( ;; ) {
				if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
					if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
				}
				if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) {
					pbTarget++;
					if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
				}
				pbTarget++;
				if (pbTarget > pbTargetMax) return(NULL);
			}
		} else {
		}
		for ( ;; ) {
			if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) ) return((pbTarget-2));
			if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
			pbTarget++;
			if (pbTarget > pbTargetMax) return(NULL);
		}

	} else {
		if (cbTarget<HaystackThresholdSekireiTchittoGritto) { // This value is arbitrary (don't know how exactly), it ensures (at least must) better performance than 'Boyer_Moore_Horspool'.

		pbTarget = pbTarget+cbPattern;
		ulHashPattern = *(uint32_t *)(pbPattern);
		SINGLET = ulHashPattern & 0xFF;
		Quadruplet2nd = SINGLET<<8;
		Quadruplet3rd = SINGLET<<16;
		Quadruplet4th = SINGLET<<24;
		for ( ;; ) {
			AdvanceHopperGrass = 0;
			ulHashTarget = *(uint32_t *)(pbTarget-cbPattern);
			if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
				count = cbPattern-1;
				while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
					if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
					count--;
				}
				if ( count == 0) return((pbTarget-cbPattern));
			} else { // The goal here: to avoid memory accesses by stressing the registers.
				if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
					AdvanceHopperGrass++;
					if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
						AdvanceHopperGrass++;
						if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
					}
				}
			}
			AdvanceHopperGrass++;
			pbTarget = pbTarget + AdvanceHopperGrass;
			if (pbTarget > pbTargetMax) return(NULL);
		}

		} else { //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
	if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto ) {

	// BMH order 2:
	if ( cbPattern<=NeedleThreshold2vs4TchittoGritto ) {
```

```
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(uint32_t *)(pbPattern+cbPattern-4); // Last four bytes
                                i=0;
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
//Possible commenting of next line:
                                for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=1;
                                while (i <= cbTarget-cbPattern) {
                                        Gulliver = 1; // 'Gulliver' is the skip
        // Few thoughts regarding an excellent Skip Performance etude:
        // Something "still" considered crazy: using BITwise order 3, not pseudo order 3, though!
        // 2^24 = 16MB BYTEwise or 2^(24-3) = 2MB BITwise, when searching big haystacks e.g. Wikipedia 42GB with Kazahana then this 2MB lookup table seems not so atrocious.
        // The code is like:
        // ulHashTarget = *(uint32_t *)&pbTarget[i+cbPattern-4]; // One memory access instead of 2
        // if ( bm_Horspool_Order3[ulHashTarget>>8] == 0 ) Gulliver = cbPattern-(3-1); else
        // if ( bm_Horspool_Order3[ulHashTarget&0xFFFFFF] == 0 ) Gulliver = cbPattern-(3-1)-1; else
        // {
        // ...
        // }
                                        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]] != 0 ) {
                                                if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-2]] + bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-1]] != 2 ) Gulliver = cbPattern-(2-1)-2; else {
                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for remainder) when going under 0 in loop below:
                                                        // Order 4 [
                                                // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
                                                //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                        count = cbPattern-4+1;
                                                        //count = count-4; // Double-beauty here of already being checked 'ulHashTarget' and not polluting/repeating the final lookup below.
                                                        while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                        if ( count <= 0 ) {
                                                                return(pbTarget+i);
                                                        }
                                                        //if ( count <= 0 ) {
                                                        //        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                        //}
                                                        //else {
                                                        //        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+count-1]] == 0 ) Gulliver = count; // 1 or bigger, as it should
                                                        //}
                                                        // Order 4 ]
                                                        }
                                                } // Means AT LEAST one of the BBs is 0, enforce lower skip: MIN( cbPattern-(2-1)-1 , cbPattern-(2-1)-2 )
                                        } else Gulliver = cbPattern-(2-1);
                                        i = i + Gulliver;
                                        //GlobalI++; // Comment it, it is only for stats.
                                }
                                return(NULL);
        // BMH order 4, needle should be >=8:
        } else { //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                i=0;
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
                                // In line below we "hash" 4bytes to 2bytes i.e. 16bit table, how to compute TOTAL number of BBs, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox'
and Order=4 we have BBs = 11-4+1=8:
```

```
                                        //"fast"
                                        //"aste"
                                        //"stes"
                                        //"test"
                                        //"est "
                                        //"st f"
                                        //"t fo"
                                        //" fox"
                                        //for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( *(unsigned short *)(pbPattern+j+0) + *(unsigned short *)(pbPattern+j+2) ) & ( (1<<16)-1 )]=1;
//Possible commenting of next line:
                                        for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( (*(uint32_t *)(pbPattern+j+0)>>16)+(*(uint32_t *)(pbPattern+j+0)&0xFFFF) ) & ( (1<<16)-1 )]=1;
                                        while (i <= cbTarget-cbPattern) {
                                                Gulliver = 1;
                                                if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]&0xFFFF) ) & ( (1<<16)-1 )] != 0 ) {
                                                        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = cbPattern-(2-1)-2-4; else {
                                                                // Order 4 [
                                                                // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                                // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:
                                                                //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                                //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                                //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                                //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                                //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                                //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                                //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                                //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                count = cbPattern-4+1;
                                                                while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                        count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                if ( count <= 0 ) {
                                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                }
                                                                //else {
                                                                //        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+count-1]>>16)+(*(uint32_t *)&pbTarget[i+count-1]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = count; // 1 or
bigger, as it should

                                                                //}
                                                                // Order 4 ]
                                                        }
                                                } else Gulliver = cbPattern-(2-1)-2; // -2 because we check the 4 rightmost bytes not 2.
                                                i = i + Gulliver;
                                                //GlobalI++; // Comment it, it is only for stats.
                                        }
                                        return(NULL);
                } //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )

                } else { // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() [
                                        //countSTATIC = cbPattern-2-2;
                                        ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                        //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                        i=0;
                                        for (a=0; a < 1<<(HashTableSizeSekireiTchittoGritto-3); a++) {bm_Hasherezade_HASH[a]= 0;} // to-do: 'memset' if not optimized
                                        // cbPattern - Order + 1 i.e. number of BBs for 11 'fastest fox' 11-8+1=4: 'fastest ', 'astest f', 'stest fo', 'test fox'
                                        for (j=0; j < cbPattern-12+1; j++) {
                                                hash32 = (2166136261UL ^ *(uint32_t *)(pbPattern+j+0)) * 709607;
                                                hash32B = (2166136261UL ^ *(uint32_t *)(pbPattern+j+4)) * 709607;
                                                hash32C = (2166136261UL ^ *(uint32_t *)(pbPattern+j+8)) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                bm_Hasherezade_HASH[hash32>>3]= bm_Hasherezade_HASH[hash32>>3] | (1<<(hash32&0x7));
                                        }
                                        while (i <= cbTarget-cbPattern) {
                                                Gulliver = 1; // Assume minimal jump as initial value.
                                                // The goal: to jump when the rightmost 8bytes (Order 8 Horspool) of window do not look like any of Needle prefixes i.e. are not to be found. This maximum jump equals cbPattern-(Order-1) or 11-(8-1)=4 for
'fastest fox' - a small one but for Needle 31 bytes the jump equals 31-(8-1)=24
                                                //GlobalHashSectionExecution++; // Comment it, it is only for stats.
                                                        hash32 = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+0)) * 709607;
```

```
                                                hash32B = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+4)) * 709607;
                                                hash32C = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+8)) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                if ( (bm_Hasherezade_HASH[hash32>>3] & (1<<(hash32&0x7))) ==0 )     Gulliver = cbPattern-(12-1);
                                                else {
                                                //if ( Gulliver == 1 ) { // Means the Building-Block order 8/12 is found somewhere i.e. NO MAXIMUM SKIP
                                                                // Order 4 [
                                                                // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                                // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:
                                                                //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                                //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                                //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                                //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                                //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                                //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                                //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                                //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                count = cbPattern-4+1;
                                                                while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                          count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                if ( count <= 0 ) {
                                                                          if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                }
                                                                // Order 4 ]
                                                }
                                                i = i + Gulliver;
                                                //GlobalI++; // Comment it, it is only for stats.
                                        } // while (i <= cbTarget-cbPattern)
                                        return(NULL);
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() ]
                    } // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
                              } //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
                    } //if ( cbPattern<4 )
}
char * Railgun_Sekireigan_Wolfram_3 (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
{
                    char * pbTargetMax = pbTarget + cbTarget;
                    register uint32_t ulHashPattern;
                    register uint32_t ulHashTarget;
                    signed long count;
                    //signed long countSTATIC;

                    unsigned char SINGLET;
                    uint32_t Quadruplet2nd;
                    uint32_t Quadruplet3rd;
                    uint32_t Quadruplet4th;

                    uint32_t AdvanceHopperGrass;

                    uint32_t a, i, j;
//Global is next line already:
                    //unsigned char bm_Horspool_Order2[256*256]; // BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
                    uint32_t Gulliver; // or unsigned char or unsigned short

                    //unsigned char bm_Hasherezade_HASH[1<<(HashTableSizeSekireiTchittoGritto-3)];
                    uint32_t hash32;
                    uint32_t hash32B;
                    uint32_t hash32C;

                    if (cbPattern > cbTarget) return(NULL);

                    if ( cbPattern<4 ) {

                    pbTarget = pbTarget+cbPattern;
                              ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
                              if ( cbPattern==3 ) {
```

```
                        for ( ;; ) {
                                if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
                                        if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
                                }
                                if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) {
                                        pbTarget++;
                                        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
                                }
                                pbTarget++;
                                if (pbTarget > pbTargetMax) return(NULL);
                        }
                } else {
                }
                for ( ;; ) {
                        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) ) return((pbTarget-2));
                        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
                        pbTarget++;
                        if (pbTarget > pbTargetMax) return(NULL);
                }

        } else {
                if (cbTarget<HaystackThresholdSekireiTchittoGritto) { // This value is arbitrary (don't know how exactly), it ensures (at least must) better performance than 'Boyer_Moore_Horspool'.

                pbTarget = pbTarget+cbPattern;
                ulHashPattern = *(uint32_t *)(pbPattern);
                SINGLET = ulHashPattern & 0xFF;
                Quadruplet2nd = SINGLET<<8;
                Quadruplet3rd = SINGLET<<16;
                Quadruplet4th = SINGLET<<24;
                for ( ;; ) {
                        AdvanceHopperGrass = 0;
                        ulHashTarget = *(uint32_t *)(pbTarget-cbPattern);
                        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
                                count = cbPattern-1;
                                while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                                        if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
                                        count--;
                                }
                                if ( count == 0) return((pbTarget-cbPattern));
                        } else { // The goal here: to avoid memory accesses by stressing the registers.
                                if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                        AdvanceHopperGrass++;
                                        if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                AdvanceHopperGrass++;
                                                if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                        }
                                }
                        }
                        AdvanceHopperGrass++;
                        pbTarget = pbTarget + AdvanceHopperGrass;
                        if (pbTarget > pbTargetMax) return(NULL);
                }

                } else { //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
        if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto ) {

        // BMH order 2:
        if ( cbPattern<=NeedleThreshold2vs4TchittoGritto ) {
                        //countSTATIC = cbPattern-2-2;
                        ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                        //ulHashTarget = *(uint32_t *)(pbPattern+cbPattern-4); // Last four bytes
                        i=0;
                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                        //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
//Possible commenting of next line:
                        for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=1;
                        while (i <= cbTarget-cbPattern) {
```

```
                                        Gulliver = 1; // 'Gulliver' is the skip
        // Few thoughts regarding an excellent Skip Performance etude:
        // Something "still" considered crazy: using BITwise order 3, not pseudo order 3, though!
        // 2^24 = 16MB BYTEwise or 2^(24-3) = 2MB BITwise, when searching big haystacks e.g. Wikipedia 42GB with Kazahana then this 2MB lookup table seems not so atrocious.
        // The code is like:
        // ulHashTarget = *(uint32_t *)&pbTarget[i+cbPattern-4]; // One memory access instead of 2
        // if ( bm_Horspool_Order3[ulHashTarget>>8] == 0 ) Gulliver = cbPattern-(3-1); else
        // if ( bm_Horspool_Order3[ulHashTarget&0xFFFFFF] == 0 ) Gulliver = cbPattern-(3-1)-1; else
        // {
        // ...
        // }
                                        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]] != 0 ) {
                                                if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-2]] + bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-1]] != 2 ) Gulliver = cbPattern-(2-1)-2; else {
                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for remainder) when going under 0 in loop below:
                                                        // Order 4 [
                                                // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
                                                //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                        count = cbPattern-4+1;
                                                        //count = count-4; // Double-beauty here of already being checked 'ulHashTarget' and not polluting/repeating the final lookup below.
                                                        while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                        if ( count <= 0 ) {
                                                                return(pbTarget+i);
                                                        }
                                                        //if ( count <= 0 ) {
                                                        //        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                        //}
                                                        //else {
                                                        //        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+count-1]] == 0 ) Gulliver = count; // 1 or bigger, as it should
                                                        //}
                                                        // Order 4 ]
                                                        }
                                                } // Means AT LEAST one of the BBs is 0, enforce lower skip: MIN( cbPattern-(2-1)-1 , cbPattern-(2-1)-2 )
                                        } else Gulliver = cbPattern-(2-1);
                                        i = i + Gulliver;
                                        //GlobalI++; // Comment it, it is only for stats.
                                }
                                return(NULL);
        // BMH order 4, needle should be >=8:
        } else { //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                i=0;
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
                                // In line below we "hash" 4bytes to 2bytes i.e. 16bit table, how to compute TOTAL number of BBs, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox'
and Order=4 we have BBs = 11-4+1=8:
                                //"fast"
                                //"aste"
                                //"stes"
                                //"test"
                                //"est "
                                //"st f"
                                //"t fo"
                                //" fox"
                                //for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( *(unsigned short *)(pbPattern+j+0) + *(unsigned short *)(pbPattern+j+2) ) & ( (1<<16)-1 )]=1;
//Possible commenting of next line:
                                for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( (*(uint32_t *)(pbPattern+j+0)>>16)+(*(uint32_t *)(pbPattern+j+0)&0xFFFF) ) & ( (1<<16)-1 )]=1;
```

```
                                        while (i <= cbTarget-cbPattern) {
                                                Gulliver = 1;
                                                if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]&0xFFFF) ) & ( (1<<16)-1 )] != 0 ) {
                                                        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = cbPattern-(2-1)-2-4; else {
                                                                // Order 4 [
                                                                // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                                // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:
                                                                //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                                //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                                //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                                //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                                //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                                //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                                //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                                //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                        count = cbPattern-4+1;
                                                                        while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                                count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                        if ( count <= 0 ) {
                                                                                if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                        }
                                                                        //else {
                                                                        //        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+count-1]>>16)+(*(uint32_t *)&pbTarget[i+count-1]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = count; // 1 or
bigger, as it should
                                                                        //}
                                                                        // Order 4 ]
                                                        }
                                                } else Gulliver = cbPattern-(2-1)-2; // -2 because we check the 4 rightmost bytes not 2.
                                                i = i + Gulliver;
                                                //GlobalI++; // Comment it, it is only for stats.
                                        }
                                        return(NULL);
                                } //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )

                                } else { // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() [
                                        //countSTATIC = cbPattern-2-2;
                                        ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                        //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                        i=0;
                                        for (a=0; a < 1<<(HashTableSizeSekireiTchittoGritto-3); a++) {bm_Hasherezade_HASH[a]= 0;} // to-do: 'memset' if not optimized
                                        // cbPattern - Order + 1 i.e. number of BBs for 11 'fastest fox' 11-8+1=4: 'fastest ', 'astest f', 'stest fo', 'test fox'
                                        for (j=0; j < cbPattern-12+1; j++) {
                                                hash32 = (2166136261UL ^ *(uint32_t *)(pbPattern+j+0)) * 709607;
                                                hash32B = (2166136261UL ^ *(uint32_t *)(pbPattern+j+4)) * 709607;
                                                hash32C = (2166136261UL ^ *(uint32_t *)(pbPattern+j+8)) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                bm_Hasherezade_HASH[hash32>>3]= bm_Hasherezade_HASH[hash32>>3] | (1<<(hash32&0x7));
                                        }
                                        while (i <= cbTarget-cbPattern) {
                                                Gulliver = 1; // Assume minimal jump as initial value.
                                                // The goal: to jump when the rightmost 8bytes (Order 8 Horspool) of window do not look like any of Needle prefixes i.e. are not to be found. This maximum jump equals cbPattern-(Order-1) or 11-(8-1)=4 for
'fastest fox' - a small one but for Needle 31 bytes the jump equals 31-(8-1)=24
                                                //GlobalHashSectionExecution++; // Comment it, it is only for stats.
                                                        hash32 = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+0)) * 709607;
                                                        hash32B = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+4)) * 709607;
                                                        hash32C = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+8)) * 709607;
                                                        hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                        hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                        hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                        if ( (bm_Hasherezade_HASH[hash32>>3] & (1<<(hash32&0x7))) ==0 )     Gulliver = cbPattern-(12-1);
                                                        else {
                                                        //if ( Gulliver == 1 ) { // Means the Building-Block order 8/12 is found somewhere i.e. NO MAXIMUM SKIP
                                                                        // Order 4 [
                                                                // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                                // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
```

```
4+1=8:
                                                                        //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                                        //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                                        //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                                        //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                                        //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                                        //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                                        //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                                        //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                                  count = cbPattern-4+1;
                                                                                  while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                                            count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                                  if ( count <= 0 ) {
                                                                                            if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                                  }
                                                                                  // Order 4 ]
                                                                        }
                                                            i = i + Gulliver;
                                                            //GlobalI++; // Comment it, it is only for stats.
                                                  } // while (i <= cbTarget-cbPattern)
                                                  return(NULL);
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() ]
          } // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
                    } //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
          } //if ( cbPattern<4 )
}
char * Railgun_Sekireigan_Wolfram_4 (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
{
          char * pbTargetMax = pbTarget + cbTarget;
          register uint32_t ulHashPattern;
          register uint32_t ulHashTarget;
          signed long count;
          //signed long countSTATIC;

          unsigned char SINGLET;
          uint32_t Quadruplet2nd;
          uint32_t Quadruplet3rd;
          uint32_t Quadruplet4th;

          uint32_t AdvanceHopperGrass;

          uint32_t a, i, j;
//Global is next line already:
          //unsigned char bm_Horspool_Order2[256*256]; // BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
          uint32_t Gulliver; // or unsigned char or unsigned short

          //unsigned char bm_Hasherezade_HASH[1<<(HashTableSizeSekireiTchittoGritto-3)];
          uint32_t hash32;
          uint32_t hash32B;
          uint32_t hash32C;

          if (cbPattern > cbTarget) return(NULL);

          if ( cbPattern<4 ) {

          pbTarget = pbTarget+cbPattern;
                    ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
                    if ( cbPattern==3 ) {
                              for ( ;; ) {
                                        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
                                                  if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
                                        }
                                        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) {
                                                  pbTarget++;
                                                  if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
                                        }
                                        pbTarget++;
                                        if (pbTarget > pbTargetMax) return(NULL);
                              }
```

```
                } else {
                }
                for ( ;; ) {
                        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) ) return((pbTarget-2));
                        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
                        pbTarget++;
                        if (pbTarget > pbTargetMax) return(NULL);
                }

        } else {
                if (cbTarget<HaystackThresholdSekireiTchittoGritto) { // This value is arbitrary (don't know how exactly), it ensures (at least must) better performance than 'Boyer_Moore_Horspool'.

                pbTarget = pbTarget+cbPattern;
                ulHashPattern = *(uint32_t *)(pbPattern);
                SINGLET = ulHashPattern & 0xFF;
                Quadruplet2nd = SINGLET<<8;
                Quadruplet3rd = SINGLET<<16;
                Quadruplet4th = SINGLET<<24;
                for ( ;; ) {
                        AdvanceHopperGrass = 0;
                        ulHashTarget = *(uint32_t *)(pbTarget-cbPattern);
                        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
                                count = cbPattern-1;
                                while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                                        if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
                                        count--;
                                }
                                if ( count == 0) return((pbTarget-cbPattern));
                        } else { // The goal here: to avoid memory accesses by stressing the registers.
                                if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                        AdvanceHopperGrass++;
                                        if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                AdvanceHopperGrass++;
                                                if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                        }
                                }
                        }
                        AdvanceHopperGrass++;
                        pbTarget = pbTarget + AdvanceHopperGrass;
                        if (pbTarget > pbTargetMax) return(NULL);
                }

                } else { //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
        if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto ) {

        // BMH order 2:
        if ( cbPattern<=NeedleThreshold2vs4TchittoGritto ) {
                        //countSTATIC = cbPattern-2-2;
                        ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                        //ulHashTarget = *(uint32_t *)(pbPattern+cbPattern-4); // Last four bytes
                        i=0;
                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                        //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
//Possible commenting of next line:
                        for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=1;
                        while (i <= cbTarget-cbPattern) {
                                Gulliver = 1; // 'Gulliver' is the skip
        // Few thoughts regarding an excellent Skip Performance etude:
        // Something "still" considered crazy: using BITwise order 3, not pseudo order 3, though!
        // 2^24 = 16MB BYTEwise or 2^(24-3) = 2MB BITwise, when searching big haystacks e.g. Wikipedia 42GB with Kazahana then this 2MB lookup table seems not so atrocious.
        // The code is like:
        // ulHashTarget = *(uint32_t *)&pbTarget[i+cbPattern-4]; // One memory access instead of 2
        // if ( bm_Horspool_Order3[ulHashTarget>>8] == 0 ) Gulliver = cbPattern-(3-1); else
        // if ( bm_Horspool_Order3[ulHashTarget&0xFFFFFF] == 0 ) Gulliver = cbPattern-(3-1)-1; else
        // {
        // ...
        // }
```

```
                                        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]] != 0 ) {
                                                if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-2]] + bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-1]] != 2 ) Gulliver = cbPattern-(2-1)-2; else {
                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for remainder) when going under 0 in loop below:
                                                        // Order 4 [
                                                // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
                                                //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                        count = cbPattern-4+1;
                                                        //count = count-4; // Double-beauty here of already being checked 'ulHashTarget' and not polluting/repeating the final lookup below.
                                                        while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                        if ( count <= 0 ) {
                                                                return(pbTarget+i);
                                                        }
                                                        //if ( count <= 0 ) {
                                                        //        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                        //}
                                                        //else {
                                                        //        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+count-1]] == 0 ) Gulliver = count; // 1 or bigger, as it should
                                                        //}
                                                        // Order 4 ]
                                                        }
                                                } // Means AT LEAST one of the BBs is 0, enforce lower skip: MIN( cbPattern-(2-1)-1 , cbPattern-(2-1)-2 )
                                        } else Gulliver = cbPattern-(2-1);
                                        i = i + Gulliver;
                                        //GlobalI++; // Comment it, it is only for stats.
                                }
                                return(NULL);
        // BMH order 4, needle should be >=8:
        } else { //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                i=0;
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
                                // In line below we "hash" 4bytes to 2bytes i.e. 16bit table, how to compute TOTAL number of BBs, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox'
and Order=4 we have BBs = 11-4+1=8:
                                //"fast"
                                //"aste"
                                //"stes"
                                //"test"
                                //"est "
                                //"st f"
                                //"t fo"
                                //" fox"
                                //for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( *(unsigned short *)(pbPattern+j+0) + *(unsigned short *)(pbPattern+j+2) ) & ( (1<<16)-1 )]=1;
//Possible commenting of next line:
                                for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( (*(uint32_t *)(pbPattern+j+0)>>16)+(*(uint32_t *)(pbPattern+j+0)&0xFFFF) ) & ( (1<<16)-1 )]=1;
                                while (i <= cbTarget-cbPattern) {
                                        Gulliver = 1;
                                        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]&0xFFFF) ) & ( (1<<16)-1 )] != 0 ) {
                                                if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = cbPattern-(2-1)-2-4; else {
                                                                // Order 4 [
                                                        // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                        // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:
                                                        //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                        //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                        //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
```

```
                                                //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                          count = cbPattern-4+1;
                                                          while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                    count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                          if ( count <= 0 ) {
                                                                    if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                          }
                                                          //else {
                                                          //        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+count-1]>>16)+(*(uint32_t *)&pbTarget[i+count-1]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = count; // 1 or
bigger, as it should
                                                          //}
                                                          // Order 4 ]
                                                }
                                        } else Gulliver = cbPattern-(2-1)-2; // -2 because we check the 4 rightmost bytes not 2.
                                        i = i + Gulliver;
                                        //GlobalI++; // Comment it, it is only for stats.
                                }
                                return(NULL);
          } //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )

          } else { // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() [
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                i=0;
                                for (a=0; a < 1<<(HashTableSizeSekireiTchittoGritto-3); a++) {bm_Hasherezade_HASH[a]= 0;} // to-do: 'memset' if not optimized
                                // cbPattern - Order + 1 i.e. number of BBs for 11 'fastest fox' 11-8+1=4: 'fastest ', 'astest f', 'stest fo', 'test fox'
                                for (j=0; j < cbPattern-12+1; j++) {
                                        hash32 = (2166136261UL ^ *(uint32_t *)(pbPattern+j+0)) * 709607;
                                        hash32B = (2166136261UL ^ *(uint32_t *)(pbPattern+j+4)) * 709607;
                                        hash32C = (2166136261UL ^ *(uint32_t *)(pbPattern+j+8)) * 709607;
                                        hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                        hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                        hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                        bm_Hasherezade_HASH[hash32>>3]= bm_Hasherezade_HASH[hash32>>3] | (1<<(hash32&0x7));
                                }
                                while (i <= cbTarget-cbPattern) {
                                        Gulliver = 1; // Assume minimal jump as initial value.
                                        // The goal: to jump when the rightmost 8bytes (Order 8 Horspool) of window do not look like any of Needle prefixes i.e. are not to be found. This maximum jump equals cbPattern-(Order-1) or 11-(8-1)=4 for
'fastest fox' - a small one but for Needle 31 bytes the jump equals 31-(8-1)=24
                                        //GlobalHashSectionExecution++; // Comment it, it is only for stats.
                                                hash32 = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+0)) * 709607;
                                                hash32B = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+4)) * 709607;
                                                hash32C = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+8)) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                if ( (bm_Hasherezade_HASH[hash32>>3] & (1<<(hash32&0x7))) ==0 )     Gulliver = cbPattern-(12-1);
                                                else {
                                                //if ( Gulliver == 1 ) { // Means the Building-Block order 8/12 is found somewhere i.e. NO MAXIMUM SKIP
                                                          // Order 4 [
                                                // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:
                                                //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                          count = cbPattern-4+1;
                                                          while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
```

```
								count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
							if ( count <= 0 ) {
								if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
							}
							// Order 4 ]
					}
				i = i + Gulliver;
				//GlobalI++; // Comment it, it is only for stats.
			} // while (i <= cbTarget-cbPattern)
			return(NULL);
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() ]
	} // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
		} //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
	} //if ( cbPattern<4 )
}
char * Railgun_Sekireigan_Wolfram_5 (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
{
	char * pbTargetMax = pbTarget + cbTarget;
	register uint32_t ulHashPattern;
	register uint32_t ulHashTarget;
	signed long count;
	//signed long countSTATIC;

	unsigned char SINGLET;
	uint32_t Quadruplet2nd;
	uint32_t Quadruplet3rd;
	uint32_t Quadruplet4th;

	uint32_t AdvanceHopperGrass;

	uint32_t a, i, j;
//Global is next line already:
	//unsigned char bm_Horspool_Order2[256*256]; // BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
	uint32_t Gulliver; // or unsigned char or unsigned short

	//unsigned char bm_Hasherezade_HASH[1<<(HashTableSizeSekireiTchittoGritto-3)];
	uint32_t hash32;
	uint32_t hash32B;
	uint32_t hash32C;

	if (cbPattern > cbTarget) return(NULL);

	if ( cbPattern<4 ) {

	pbTarget = pbTarget+cbPattern;
		ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
		if ( cbPattern==3 ) {
			for ( ;; ) {
				if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
					if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
				}
				if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) {
					pbTarget++;
					if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
				}
				pbTarget++;
				if (pbTarget > pbTargetMax) return(NULL);
			}
		} else {
		}
		for ( ;; ) {
			if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) ) return((pbTarget-2));
			if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
			pbTarget++;
			if (pbTarget > pbTargetMax) return(NULL);
		}

	} else {
		if (cbTarget<HaystackThresholdSekireiTchittoGritto) { // This value is arbitrary (don't know how exactly), it ensures (at least must) better performance than 'Boyer_Moore_Horspool'.
```

```
                pbTarget = pbTarget+cbPattern;
                ulHashPattern = *(uint32_t *)(pbPattern);
                SINGLET = ulHashPattern & 0xFF;
                Quadruplet2nd = SINGLET<<8;
                Quadruplet3rd = SINGLET<<16;
                Quadruplet4th = SINGLET<<24;
                for ( ;; ) {
                        AdvanceHopperGrass = 0;
                        ulHashTarget = *(uint32_t *)(pbTarget-cbPattern);
                        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
                                count = cbPattern-1;
                                while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                                        if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
                                        count--;
                                }
                                if ( count == 0) return((pbTarget-cbPattern));
                        } else { // The goal here: to avoid memory accesses by stressing the registers.
                                if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                        AdvanceHopperGrass++;
                                        if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                AdvanceHopperGrass++;
                                                if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                        }
                                }
                        }
                        AdvanceHopperGrass++;
                        pbTarget = pbTarget + AdvanceHopperGrass;
                        if (pbTarget > pbTargetMax) return(NULL);
                }

                } else { //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
        if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto ) {

        // BMH order 2:
        if ( cbPattern<=NeedleThreshold2vs4TchittoGritto ) {
                        //countSTATIC = cbPattern-2-2;
                        ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                        //ulHashTarget = *(uint32_t *)(pbPattern+cbPattern-4); // Last four bytes
                        i=0;
                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                        //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
//Possible commenting of next line:
                        for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=1;
                        while (i <= cbTarget-cbPattern) {
                                Gulliver = 1; // 'Gulliver' is the skip
        // Few thoughts regarding an excellent Skip Performance etude:
        // Something "still" considered crazy: using BITwise order 3, not pseudo order 3, though!
        // 2^24 = 16MB BYTEwise or 2^(24-3) = 2MB BITwise, when searching big haystacks e.g. Wikipedia 42GB with Kazahana then this 2MB lookup table seems not so atrocious.
        // The code is like:
        // ulHashTarget = *(uint32_t *)&pbTarget[i+cbPattern-4]; // One memory access instead of 2
        // if ( bm_Horspool_Order3[ulHashTarget>>8] == 0 ) Gulliver = cbPattern-(3-1); else
        // if ( bm_Horspool_Order3[ulHashTarget&0xFFFFFF] == 0 ) Gulliver = cbPattern-(3-1)-1; else
        // {
        // ...
        // }
                                if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]] != 0 ) {
                                        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-2]] + bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-1]] != 2 ) Gulliver = cbPattern-(2-1)-2; else {
                                                if ( *(uint32_t *)&pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for remainder) when going under 0 in loop below:
                                                // Order 4 [
                                        // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                        // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
                                        //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                        //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                        //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                        //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                        //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
```

```
                                        //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                        //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                        //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                count = cbPattern-4+1;
                                                //count = count-4; // Double-beauty here of already being checked 'ulHashTarget' and not polluting/repeating the final lookup below.
                                                while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                        count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                if ( count <= 0 ) {
                                                        return(pbTarget+i);
                                                }
                                                //if ( count <= 0 ) {
                                                //        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                //}
                                                //else {
                                                //        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+count-1]] == 0 ) Gulliver = count; // 1 or bigger, as it should
                                                //}
                                                // Order 4 ]
                                                }
                                        } // Means AT LEAST one of the BBs is 0, enforce lower skip: MIN( cbPattern-(2-1)-1 , cbPattern-(2-1)-2 )
                                } else Gulliver = cbPattern-(2-1);
                                i = i + Gulliver;
                                //GlobalI++; // Comment it, it is only for stats.
                        }
                        return(NULL);
        // BMH order 4, needle should be >=8:
        } else { //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )
                        //countSTATIC = cbPattern-2-2;
                        ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                        //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                        i=0;
                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                        //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
                        // In line below we "hash" 4bytes to 2bytes i.e. 16bit table, how to compute TOTAL number of BBs, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox'
and Order=4 we have BBs = 11-4+1=8:
                        //"fast"
                        //"aste"
                        //"stes"
                        //"test"
                        //"est "
                        //"st f"
                        //"t fo"
                        //" fox"
                        //for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( *(unsigned short *)(pbPattern+j+0) + *(unsigned short *)(pbPattern+j+2) ) & ( (1<<16)-1 )]=1;
//Possible commenting of next line:
                        for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( (*(uint32_t *)(pbPattern+j+0)>>16)+(*(uint32_t *)(pbPattern+j+0)&0xFFFF) ) & ( (1<<16)-1 )]=1;
                        while (i <= cbTarget-cbPattern) {
                                Gulliver = 1;
                                if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]&0xFFFF) ) & ( (1<<16)-1 )] != 0 ) {
                                        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = cbPattern-(2-1)-2-4; else {
                                                // Order 4 [
                                        // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                        // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:
                                        //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                        //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                        //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                        //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                        //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                        //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                        //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                        //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                count = cbPattern-4+1;
                                                while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                        count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                if ( count <= 0 ) {
                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                }
```

```
                                                                        //else {
                                                                        //        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+count-1]>>16)+(*(uint32_t *)&pbTarget[i+count-1]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = count; // 1 or
bigger, as it should
                                                                        //}
                                                                        // Order 4 ]
                                                                }
                                                        } else Gulliver = cbPattern-(2-1)-2; // -2 because we check the 4 rightmost bytes not 2.
                                                        i = i + Gulliver;
                                                        //GlobalI++; // Comment it, it is only for stats.
                                                }
                                                return(NULL);
                        } //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )

                        } else { // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() [
                                                //countSTATIC = cbPattern-2-2;
                                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                                i=0;
                                                for (a=0; a < 1<<(HashTableSizeSekireiTchittoGritto-3); a++) {bm_Hasherezade_HASH[a]= 0;} // to-do: 'memset' if not optimized
                                                // cbPattern - Order + 1 i.e. number of BBs for 11 'fastest fox' 11-8+1=4: 'fastest ', 'astest f', 'stest fo', 'test fox'
                                                for (j=0; j < cbPattern-12+1; j++) {
                                                        hash32 = (2166136261UL ^ *(uint32_t *)(pbPattern+j+0)) * 709607;
                                                        hash32B = (2166136261UL ^ *(uint32_t *)(pbPattern+j+4)) * 709607;
                                                        hash32C = (2166136261UL ^ *(uint32_t *)(pbPattern+j+8)) * 709607;
                                                        hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                        hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                        hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                        bm_Hasherezade_HASH[hash32>>3]= bm_Hasherezade_HASH[hash32>>3] | (1<<(hash32&0x7));
                                                }
                                                while (i <= cbTarget-cbPattern) {
                                                        Gulliver = 1; // Assume minimal jump as initial value.
                                                        // The goal: to jump when the rightmost 8bytes (Order 8 Horspool) of window do not look like any of Needle prefixes i.e. are not to be found. This maximum jump equals cbPattern-(Order-1) or 11-(8-1)=4 for
'fastest fox' - a small one but for Needle 31 bytes the jump equals 31-(8-1)=24
                                                        //GlobalHashSectionExecution++; // Comment it, it is only for stats.
                                                                hash32 = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+0)) * 709607;
                                                                hash32B = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+4)) * 709607;
                                                                hash32C = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+8)) * 709607;
                                                                hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                                hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                                hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                                if ( (bm_Hasherezade_HASH[hash32>>3] & (1<<(hash32&0x7))) ==0 )     Gulliver = cbPattern-(12-1);
                                                                else {
                                                                //if ( Gulliver == 1 ) { // Means the Building-Block order 8/12 is found somewhere i.e. NO MAXIMUM SKIP
                                                                        // Order 4 [
                                                                // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                                // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:

                                                                //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                                //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                                //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                                //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                                //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                                //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                                //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                                //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                        count = cbPattern-4+1;
                                                                        while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                                count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                        if ( count <= 0 ) {
                                                                                if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                        }
                                                                        // Order 4 ]
                                                                }
                                                        i = i + Gulliver;
                                                        //GlobalI++; // Comment it, it is only for stats.
                                                } // while (i <= cbTarget-cbPattern)
                                                return(NULL);
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() ]
```

```
		} // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
			} //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
		} //if ( cbPattern<4 )
}
char * Railgun_Sekireigan_Wolfram_6 (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
{
		char * pbTargetMax = pbTarget + cbTarget;
		register uint32_t ulHashPattern;
		register uint32_t ulHashTarget;
		signed long count;
		//signed long countSTATIC;

		unsigned char SINGLET;
		uint32_t Quadruplet2nd;
		uint32_t Quadruplet3rd;
		uint32_t Quadruplet4th;

		uint32_t AdvanceHopperGrass;

		uint32_t a, i, j;
//Global is next line already:
		//unsigned char bm_Horspool_Order2[256*256]; // BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
		uint32_t Gulliver; // or unsigned char or unsigned short

		//unsigned char bm_Hasherezade_HASH[1<<(HashTableSizeSekireiTchittoGritto-3)];
		uint32_t hash32;
		uint32_t hash32B;
		uint32_t hash32C;

		if (cbPattern > cbTarget) return(NULL);

		if ( cbPattern<4 ) {

		pbTarget = pbTarget+cbPattern;
			ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
			if ( cbPattern==3 ) {
				for ( ;; ) {
					if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
						if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
					}
					if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) {
						pbTarget++;
						if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
					}
					pbTarget++;
					if (pbTarget > pbTargetMax) return(NULL);
				}
			} else {
			}
			for ( ;; ) {
				if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) ) return((pbTarget-2));
				if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
				pbTarget++;
				if (pbTarget > pbTargetMax) return(NULL);
			}

		} else {
			if (cbTarget<HaystackThresholdSekireiTchittoGritto) { // This value is arbitrary (don't know how exactly), it ensures (at least must) better performance than 'Boyer_Moore_Horspool'.

			pbTarget = pbTarget+cbPattern;
			ulHashPattern = *(uint32_t *)(pbPattern);
			SINGLET = ulHashPattern & 0xFF;
			Quadruplet2nd = SINGLET<<8;
			Quadruplet3rd = SINGLET<<16;
			Quadruplet4th = SINGLET<<24;
			for ( ;; ) {
				AdvanceHopperGrass = 0;
				ulHashTarget = *(uint32_t *)(pbTarget-cbPattern);
				if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
```

```
                                        count = cbPattern-1;
                                        while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                                                    if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
                                                    count--;
                                        }
                                        if ( count == 0) return((pbTarget-cbPattern));
                                } else { // The goal here: to avoid memory accesses by stressing the registers.
                                        if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                                    AdvanceHopperGrass++;
                                                    if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                                AdvanceHopperGrass++;
                                                                if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                                    }
                                        }
                                }
                                AdvanceHopperGrass++;
                                pbTarget = pbTarget + AdvanceHopperGrass;
                                if (pbTarget > pbTargetMax) return(NULL);
                        }

                        } else { //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
            if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto ) {

            // BMH order 2:
            if ( cbPattern<=NeedleThreshold2vs4TchittoGritto ) {
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(uint32_t *)(pbPattern+cbPattern-4); // Last four bytes
                                i=0;
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
//Possible commenting of next line:
                                for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=1;
                                while (i <= cbTarget-cbPattern) {
                                        Gulliver = 1; // 'Gulliver' is the skip
            // Few thoughts regarding an excellent Skip Performance etude:
            // Something "still" considered crazy: using BITwise order 3, not pseudo order 3, though!
            // 2^24 = 16MB BYTEwise or 2^(24-3) = 2MB BITwise, when searching big haystacks e.g. Wikipedia 42GB with Kazahana then this 2MB lookup table seems not so atrocious.
            // The code is like:
            // ulHashTarget = *(uint32_t *)&pbTarget[i+cbPattern-4]; // One memory access instead of 2
            // if ( bm_Horspool_Order3[ulHashTarget>>8] == 0 ) Gulliver = cbPattern-(3-1); else
            // if ( bm_Horspool_Order3[ulHashTarget&0xFFFFFF] == 0 ) Gulliver = cbPattern-(3-1)-1; else
            // {
            // ...
            // }
                                        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]] != 0 ) {
                                                    if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-2]] + bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-1]] != 2 ) Gulliver = cbPattern-(2-1)-2; else {
                                                                if ( *(uint32_t *)&pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for remainder) when going under 0 in loop below:
                                                                // Order 4 [
                                                    // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                    // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
                                                    //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                    //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                    //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                    //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                    //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                    //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                    //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                    //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                count = cbPattern-4+1;
                                                                //count = count-4; // Double-beauty here of already being checked 'ulHashTarget' and not polluting/repeating the final lookup below.
                                                                while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                            count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                if ( count <= 0 ) {
                                                                            return(pbTarget+i);
                                                                }
                                                                //if ( count <= 0 ) {
```

```
                                                //        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                //}
                                                //else {
                                                //        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+count-1]] == 0 ) Gulliver = count; // 1 or bigger, as it should
                                                //}
                                                // Order 4 ]
                                                }
                                        } // Means AT LEAST one of the BBs is 0, enforce lower skip: MIN( cbPattern-(2-1)-1 , cbPattern-(2-1)-2 )
                                } else Gulliver = cbPattern-(2-1);
                                i = i + Gulliver;
                                //GlobalI++; // Comment it, it is only for stats.
                        }
                        return(NULL);
        // BMH order 4, needle should be >=8:
        } else { //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )
                        //countSTATIC = cbPattern-2-2;
                        ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                        //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                        i=0;
                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                        //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
                        // In line below we "hash" 4bytes to 2bytes i.e. 16bit table, how to compute TOTAL number of BBs, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox'
and Order=4 we have BBs = 11-4+1=8:
                        //"fast"
                        //"aste"
                        //"stes"
                        //"test"
                        //"est "
                        //"st f"
                        //"t fo"
                        //" fox"
                        //for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( *(unsigned short *)(pbPattern+j+0) + *(unsigned short *)(pbPattern+j+2) ) & ( (1<<16)-1 )]=1;
//Possible commenting of next line:
                        for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( (*(uint32_t *)(pbPattern+j+0)>>16)+(*(uint32_t *)(pbPattern+j+0)&0xFFFF) ) & ( (1<<16)-1 )]=1;
                        while (i <= cbTarget-cbPattern) {
                                Gulliver = 1;
                                if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]&0xFFFF) ) & ( (1<<16)-1 )] != 0 ) {
                                        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) ) & ( (1<16)-1 )] == 0 ) Gulliver = cbPattern-(2-1)-2-4; else {
                                                // Order 4 [
                                                // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:
                                                //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                        count = cbPattern-4+1;
                                                        while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                        if ( count <= 0 ) {
                                                                if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                        }
                                                        //else {
                                                        //        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+count-1]>>16)+(*(uint32_t *)&pbTarget[i+count-1]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = count; // 1 or
bigger, as it should
                                                        //}
                                                        // Order 4 ]
                                        }
                                } else Gulliver = cbPattern-(2-1)-2; // -2 because we check the 4 rightmost bytes not 2.
                                i = i + Gulliver;
                                //GlobalI++; // Comment it, it is only for stats.
                        }
                        return(NULL);
```

```c
	} //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )

	} else { // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() [
			//countSTATIC = cbPattern-2-2;
			ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
			//ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
			i=0;
			for (a=0; a < 1<<(HashTableSizeSekireiTchittoGritto-3); a++) {bm_Hasherezade_HASH[a]= 0;} // to-do: 'memset' if not optimized
			// cbPattern - Order + 1 i.e. number of BBs for 11 'fastest fox' 11-8+1=4: 'fastest ', 'astest f', 'stest fo', 'test fox'
			for (j=0; j < cbPattern-12+1; j++) {
				hash32 = (2166136261UL ^ *(uint32_t *)(pbPattern+j+0)) * 709607;
				hash32B = (2166136261UL ^ *(uint32_t *)(pbPattern+j+4)) * 709607;
				hash32C = (2166136261UL ^ *(uint32_t *)(pbPattern+j+8)) * 709607;
				hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
				hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
				hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
				bm_Hasherezade_HASH[hash32>>3]= bm_Hasherezade_HASH[hash32>>3] | (1<<(hash32&0x7));
			}
			while (i <= cbTarget-cbPattern) {
				Gulliver = 1; // Assume minimal jump as initial value.
				// The goal: to jump when the rightmost 8bytes (Order 8 Horspool) of window do not look like any of Needle prefixes i.e. are not to be found. This maximum jump equals cbPattern-(Order-1) or 11-(8-1)=4 for 'fastest fox' - a small one but for Needle 31 bytes the jump equals 31-(8-1)=24
				//GlobalHashSectionExecution++; // Comment it, it is only for stats.
					hash32 = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+0)) * 709607;
					hash32B = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+4)) * 709607;
					hash32C = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+8)) * 709607;
					hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
					hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
					hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
					if ( (bm_Hasherezade_HASH[hash32>>3] & (1<<(hash32&0x7))) ==0 )	Gulliver = cbPattern-(12-1);
					else {
					//if ( Gulliver == 1 ) { // Means the Building-Block order 8/12 is found somewhere i.e. NO MAXIMUM SKIP
							// Order 4 [
						// Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
						// Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
						//0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
						//1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
						//2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
						//3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
						//4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
						//5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
						//6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
						//7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
							count = cbPattern-4+1;
							while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
								count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
							if ( count <= 0 ) {
								if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
							}
							// Order 4 ]
					}
				i = i + Gulliver;
				//GlobalI++; // Comment it, it is only for stats.
			} // while (i <= cbTarget-cbPattern)
			return(NULL);
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() ]
	} // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
		} //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
	} //if ( cbPattern<4 )
}
char * Railgun_Sekireigan_Wolfram_7 (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
{
	char * pbTargetMax = pbTarget + cbTarget;
	register uint32_t ulHashPattern;
	register uint32_t ulHashTarget;
	signed long count;
	//signed long countSTATIC;
```

```
	unsigned char SINGLET;
	uint32_t Quadruplet2nd;
	uint32_t Quadruplet3rd;
	uint32_t Quadruplet4th;

	uint32_t AdvanceHopperGrass;

	uint32_t a, i, j;
//Global is next line already:
	//unsigned char bm_Horspool_Order2[256*256]; // BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
	uint32_t Gulliver; // or unsigned char or unsigned short

	//unsigned char bm_Hasherezade_HASH[1<<(HashTableSizeSekireiTchittoGritto-3)];
	uint32_t hash32;
	uint32_t hash32B;
	uint32_t hash32C;

	if (cbPattern > cbTarget) return(NULL);

	if ( cbPattern<4 ) {

	pbTarget = pbTarget+cbPattern;
		ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
		if ( cbPattern==3 ) {
			for ( ;; ) {
				if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
					if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
				}
				if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) {
					pbTarget++;
					if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
				}
				pbTarget++;
				if (pbTarget > pbTargetMax) return(NULL);
			}
		} else {
		}
		for ( ;; ) {
			if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) ) return((pbTarget-2));
			if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
			pbTarget++;
			if (pbTarget > pbTargetMax) return(NULL);
		}

	} else {
		if (cbTarget<HaystackThresholdSekireiTchittoGritto) { // This value is arbitrary (don't know how exactly), it ensures (at least must) better performance than 'Boyer_Moore_Horspool'.

		pbTarget = pbTarget+cbPattern;
		ulHashPattern = *(uint32_t *)(pbPattern);
		SINGLET = ulHashPattern & 0xFF;
		Quadruplet2nd = SINGLET<<8;
		Quadruplet3rd = SINGLET<<16;
		Quadruplet4th = SINGLET<<24;
		for ( ;; ) {
			AdvanceHopperGrass = 0;
			ulHashTarget = *(uint32_t *)(pbTarget-cbPattern);
			if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
				count = cbPattern-1;
				while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
					if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
					count--;
				}
				if ( count == 0) return((pbTarget-cbPattern));
			} else { // The goal here: to avoid memory accesses by stressing the registers.
				if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
					AdvanceHopperGrass++;
					if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
						AdvanceHopperGrass++;
```

```
                                                                        if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                                                }
                                                        }
                                                }
                                                AdvanceHopperGrass++;
                                                pbTarget = pbTarget + AdvanceHopperGrass;
                                                if (pbTarget > pbTargetMax) return(NULL);
                                        }

                                        } else { //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
                                if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto ) {

                                // BMH order 2:
                                if ( cbPattern<=NeedleThreshold2vs4TchittoGritto ) {
                                                //countSTATIC = cbPattern-2-2;
                                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                                //ulHashTarget = *(uint32_t *)(pbPattern+cbPattern-4); // Last four bytes
                                                i=0;
                                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
//Possible commenting of next line:
                                                for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=1;
                                                while (i <= cbTarget-cbPattern) {
                                                        Gulliver = 1; // 'Gulliver' is the skip
                                // Few thoughts regarding an excellent Skip Performance etude:
                                // Something "still" considered crazy: using BITwise order 3, not pseudo order 3, though!
                                // 2^24 = 16MB BYTEwise or 2^(24-3) = 2MB BITwise, when searching big haystacks e.g. Wikipedia 42GB with Kazahana then this 2MB lookup table seems not so atrocious.
                                // The code is like:
                                // ulHashTarget = *(uint32_t *)&pbTarget[i+cbPattern-4]; // One memory access instead of 2
                                // if ( bm_Horspool_Order3[ulHashTarget>>8] == 0 ) Gulliver = cbPattern-(3-1); else
                                // if ( bm_Horspool_Order3[ulHashTarget&0xFFFFFF] == 0 ) Gulliver = cbPattern-(3-1)-1; else
                                // {
                                // ...
                                // }
                                                        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]] != 0 ) {
                                                                if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-2]] + bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-1]] != 2 ) Gulliver = cbPattern-(2-1)-2; else {
                                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for remainder) when going under 0 in loop below:
                                                                        // Order 4 [
                                                                // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                                // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
                                                                //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                                //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                                //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                                //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                                //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                                //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                                //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                                //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                        count = cbPattern-4+1;
                                                                        //count = count-4; // Double-beauty here of already being checked 'ulHashTarget' and not polluting/repeating the final lookup below.
                                                                        while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                                count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                        if ( count <= 0 ) {
                                                                                return(pbTarget+i);
                                                                        }
                                                                        //if ( count <= 0 ) {
                                                                        //        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                        //}
                                                                        //else {
                                                                        //        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+count-1]] == 0 ) Gulliver = count; // 1 or bigger, as it should
                                                                        //}
                                                                        // Order 4 ]
                                                                        }
                                                                } // Means AT LEAST one of the BBs is 0, enforce lower skip: MIN( cbPattern-(2-1)-1 , cbPattern-(2-1)-2 )
                                                        } else Gulliver = cbPattern-(2-1);
                                                        i = i + Gulliver;
                                                        //GlobalI++; // Comment it, it is only for stats.
```

```
                                }
                                return(NULL);
            // BMH order 4, needle should be >=8:
            } else { //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                i=0;
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
                                // In line below we "hash" 4bytes to 2bytes i.e. 16bit table, how to compute TOTAL number of BBs, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox'
and Order=4 we have BBs = 11-4+1=8:
                                //"fast"
                                //"aste"
                                //"stes"
                                //"test"
                                //"est "
                                //"st f"
                                //"t fo"
                                //" fox"
                                //for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( *(unsigned short *)(pbPattern+j+0) + *(unsigned short *)(pbPattern+j+2) ) & ( (1<<16)-1 )]=1;
//Possible commenting of next line:
                                for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( (*(uint32_t *)(pbPattern+j+0)>>16)+(*(uint32_t *)(pbPattern+j+0)&0xFFFF) ) & ( (1<<16)-1 )]=1;
                                while (i <= cbTarget-cbPattern) {
                                                Gulliver = 1;
                                                if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]&0xFFFF) ) & ( (1<<16)-1 )] != 0 ) {
                                                                if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = cbPattern-(2-1)-2-4; else {
                                                                                // Order 4 [
                                                                                // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                                                // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:
                                                                                //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                                                //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                                                //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                                                //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                                                //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                                                //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                                                //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                                                //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                                count = cbPattern-4+1;
                                                                                while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                                                count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                                if ( count <= 0 ) {
                                                                                                if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                                }
                                                                                //else {
                                                                                //        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+count-1]>>16)+(*(uint32_t *)&pbTarget[i+count-1]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = count; // 1 or
bigger, as it should
                                                                                //}
                                                                                // Order 4 ]
                                                                }
                                                } else Gulliver = cbPattern-(2-1)-2; // -2 because we check the 4 rightmost bytes not 2.
                                                i = i + Gulliver;
                                                //GlobalI++; // Comment it, it is only for stats.
                                }
                                return(NULL);
            } //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )

            } else { // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() [
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                i=0;
                                for (a=0; a < 1<<(HashTableSizeSekireiTchittoGritto-3); a++) {bm_Hasherezade_HASH[a]= 0;} // to-do: 'memset' if not optimized
                                // cbPattern - Order + 1 i.e. number of BBs for 11 'fastest fox' 11-8+1=4: 'fastest ', 'astest f', 'stest fo', 'test fox'
                                for (j=0; j < cbPattern-12+1; j++) {
```

```
                                        hash32 = (2166136261UL ^ *(uint32_t *)(pbPattern+j+0)) * 709607;
                                        hash32B = (2166136261UL ^ *(uint32_t *)(pbPattern+j+4)) * 709607;
                                        hash32C = (2166136261UL ^ *(uint32_t *)(pbPattern+j+8)) * 709607;
                                        hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                        hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                        hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                        bm_Hasherezade_HASH[hash32>>3]= bm_Hasherezade_HASH[hash32>>3] | (1<<(hash32&0x7));
                                }
                                while (i <= cbTarget-cbPattern) {
                                        Gulliver = 1; // Assume minimal jump as initial value.
                                        // The goal: to jump when the rightmost 8bytes (Order 8 Horspool) of window do not look like any of Needle prefixes i.e. are not to be found. This maximum jump equals cbPattern-(Order-1) or 11-(8-1)=4 for
'fastest fox' - a small one but for Needle 31 bytes the jump equals 31-(8-1)=24
                                        //GlobalHashSectionExecution++; // Comment it, it is only for stats.
                                                hash32 = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+0)) * 709607;
                                                hash32B = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+4)) * 709607;
                                                hash32C = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+8)) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                if ( (bm_Hasherezade_HASH[hash32>>3] & (1<<(hash32&0x7))) ==0 )     Gulliver = cbPattern-(12-1);
                                                else {
                                                //if ( Gulliver == 1 ) { // Means the Building-Block order 8/12 is found somewhere i.e. NO MAXIMUM SKIP
                                                                // Order 4 [
                                                        // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                        // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:

                                                        //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                        //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                        //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                        //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                        //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                        //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                        //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                        //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                count = cbPattern-4+1;
                                                                while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                        count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                if ( count <= 0 ) {
                                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                }
                                                                // Order 4 ]
                                                }
                                        i = i + Gulliver;
                                        //GlobalI++; // Comment it, it is only for stats.
                                } // while (i <= cbTarget-cbPattern)
                                return(NULL);
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() ]
        } // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
                } //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
        } //if ( cbPattern<4 )
}
char * Railgun_Sekireigan_Wolfram_8 (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
{
        char * pbTargetMax = pbTarget + cbTarget;
        register uint32_t ulHashPattern;
        register uint32_t ulHashTarget;
        signed long count;
        //signed long countSTATIC;

        unsigned char SINGLET;
        uint32_t Quadruplet2nd;
        uint32_t Quadruplet3rd;
        uint32_t Quadruplet4th;

        uint32_t AdvanceHopperGrass;

        uint32_t a, i, j;
//Global is next line already:
        //unsigned char bm_Horspool_Order2[256*256]; // BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
```

```
	uint32_t Gulliver; // or unsigned char or unsigned short

	//unsigned char bm_Hasherezade_HASH[1<<(HashTableSizeSekireiTchittoGritto-3)];
	uint32_t hash32;
	uint32_t hash32B;
	uint32_t hash32C;

	if (cbPattern > cbTarget) return(NULL);

	if ( cbPattern<4 ) {

	pbTarget = pbTarget+cbPattern;
		ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
		if ( cbPattern==3 ) {
			for ( ;; ) {
				if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
					if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
				}
				if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) {
					pbTarget++;
					if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
				}
				pbTarget++;
				if (pbTarget > pbTargetMax) return(NULL);
			}
		} else {
		}
		for ( ;; ) {
			if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) ) return((pbTarget-2));
			if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
			pbTarget++;
			if (pbTarget > pbTargetMax) return(NULL);
		}

	} else {
		if (cbTarget<HaystackThresholdSekireiTchittoGritto) { // This value is arbitrary (don't know how exactly), it ensures (at least must) better performance than 'Boyer_Moore_Horspool'.

		pbTarget = pbTarget+cbPattern;
		ulHashPattern = *(uint32_t *)(pbPattern);
		SINGLET = ulHashPattern & 0xFF;
		Quadruplet2nd = SINGLET<<8;
		Quadruplet3rd = SINGLET<<16;
		Quadruplet4th = SINGLET<<24;
		for ( ;; ) {
			AdvanceHopperGrass = 0;
			ulHashTarget = *(uint32_t *)(pbTarget-cbPattern);
			if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
				count = cbPattern-1;
				while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
					if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
					count--;
				}
				if ( count == 0) return((pbTarget-cbPattern));
			} else { // The goal here: to avoid memory accesses by stressing the registers.
				if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
					AdvanceHopperGrass++;
					if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
						AdvanceHopperGrass++;
						if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
					}
				}
			}
			AdvanceHopperGrass++;
			pbTarget = pbTarget + AdvanceHopperGrass;
			if (pbTarget > pbTargetMax) return(NULL);
		}

		} else { //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
	if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto ) {
```

```
		// BMH order 2:
		if ( cbPattern<=NeedleThreshold2vs4TchittoGritto ) {
				//countSTATIC = cbPattern-2-2;
				ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
				//ulHashTarget = *(uint32_t *)(pbPattern+cbPattern-4); // Last four bytes
				i=0;
				//for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
				//for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
				//for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
//Possible commenting of next line:
				for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=1;
				while (i <= cbTarget-cbPattern) {
					Gulliver = 1; // 'Gulliver' is the skip
		// Few thoughts regarding an excellent Skip Performance etude:
		// Something "still" considered crazy: using BITwise order 3, not pseudo order 3, though!
		// 2^24 = 16MB BYTEwise or 2^(24-3) = 2MB BITwise, when searching big haystacks e.g. Wikipedia 42GB with Kazahana then this 2MB lookup table seems not so atrocious.
		// The code is like:
		// ulHashTarget = *(uint32_t *)&pbTarget[i+cbPattern-4]; // One memory access instead of 2
		// if ( bm_Horspool_Order3[ulHashTarget>>8] == 0 ) Gulliver = cbPattern-(3-1); else
		// if ( bm_Horspool_Order3[ulHashTarget&0xFFFFFF] == 0 ) Gulliver = cbPattern-(3-1)-1; else
		// {
		// ...
		// }
					if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]] != 0 ) {
						if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-2]] + bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-1]] != 2 ) Gulliver = cbPattern-(2-1)-2; else {
							if ( *(uint32_t *)&pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for remainder) when going under 0 in loop below:
							// Order 4 [
						// Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
						// Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
						//0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
						//1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
						//2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
						//3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
						//4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
						//5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
						//6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
						//7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
							count = cbPattern-4+1;
							//count = count-4; // Double-beauty here of already being checked 'ulHashTarget' and not polluting/repeating the final lookup below.
							while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
								count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
							if ( count <= 0 ) {
								return(pbTarget+i);
							}
							//if ( count <= 0 ) {
							//	if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
							//}
							//else {
							//	if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+count-1]] == 0 ) Gulliver = count; // 1 or bigger, as it should
							//}
							// Order 4 ]
							}
						} // Means AT LEAST one of the BBs is 0, enforce lower skip: MIN( cbPattern-(2-1)-1 , cbPattern-(2-1)-2 )
					} else Gulliver = cbPattern-(2-1);
					i = i + Gulliver;
					//GlobalI++; // Comment it, it is only for stats.
				}
				return(NULL);
		// BMH order 4, needle should be >=8:
		} else { //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )
				//countSTATIC = cbPattern-2-2;
				ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
				//ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
				i=0;
				//for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
				//for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
```

```
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
                                // In line below we "hash" 4bytes to 2bytes i.e. 16bit table, how to compute TOTAL number of BBs, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox'
and Order=4 we have BBs = 11-4+1=8:
                                //"fast"
                                //"aste"
                                //"stes"
                                //"test"
                                //"est "
                                //"st f"
                                //"t fo"
                                //" fox"
                                //for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( *(unsigned short *)(pbPattern+j+0) + *(unsigned short *)(pbPattern+j+2) ) & ( (1<<16)-1 )]=1;
//Possible commenting of next line:
                                for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( (*(uint32_t *)(pbPattern+j+0)>>16)+(*(uint32_t *)(pbPattern+j+0)&0xFFFF) ) & ( (1<<16)-1 )]=1;
                                while (i <= cbTarget-cbPattern) {
                                          Gulliver = 1;
                                          if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]&0xFFFF) ) & ( (1<<16)-1 )] != 0 ) {
                                                    if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = cbPattern-(2-1)-2-4; else {
                                                              // Order 4 [
                                                    // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                    // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:
                                                    //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                    //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                    //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                    //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                    //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                    //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                    //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                    //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                              count = cbPattern-4+1;
                                                              while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                        count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                              if ( count <= 0 ) {
                                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                              }
                                                              //else {
                                                              //        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+count-1]>>16)+(*(uint32_t *)&pbTarget[i+count-1]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = count; // 1 or
bigger, as it should
                                                              //}
                                                              // Order 4 ]
                                                    }
                                          } else Gulliver = cbPattern-(2-1)-2; // -2 because we check the 4 rightmost bytes not 2.
                                          i = i + Gulliver;
                                          //GlobalI++; // Comment it, it is only for stats.
                                }
                                return(NULL);
          } //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )

          } else { // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() [
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                i=0;
                                for (a=0; a < 1<<(HashTableSizeSekireiTchittoGritto-3); a++) {bm_Hasherezade_HASH[a]= 0;} // to-do: 'memset' if not optimized
                                // cbPattern - Order + 1 i.e. number of BBs for 11 'fastest fox' 11-8+1=4: 'fastest ', 'astest f', 'stest fo', 'test fox'
                                for (j=0; j < cbPattern-12+1; j++) {
                                          hash32 = (2166136261UL ^ *(uint32_t *)(pbPattern+j+0)) * 709607;
                                          hash32B = (2166136261UL ^ *(uint32_t *)(pbPattern+j+4)) * 709607;
                                          hash32C = (2166136261UL ^ *(uint32_t *)(pbPattern+j+8)) * 709607;
                                          hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                          hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                          hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                          bm_Hasherezade_HASH[hash32>>3]= bm_Hasherezade_HASH[hash32>>3] | (1<<(hash32&0x7));
                                }
                                while (i <= cbTarget-cbPattern) {
                                          Gulliver = 1; // Assume minimal jump as initial value.
                                          // The goal: to jump when the rightmost 8bytes (Order 8 Horspool) of window do not look like any of Needle prefixes i.e. are not to be found. This maximum jump equals cbPattern-(Order-1) or 11-(8-1)=4 for
```

```
'fastest fox' - a small one but for Needle 31 bytes the jump equals 31-(8-1)=24
                                                //GlobalHashSectionExecution++; // Comment it, it is only for stats.
                                                          hash32 = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+0)) * 709607;
                                                          hash32B = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+4)) * 709607;
                                                          hash32C = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+8)) * 709607;
                                                          hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                          hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                          hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                          if ( (bm_Hasherezade_HASH[hash32>>3] & (1<<(hash32&0x7))) ==0 )     Gulliver = cbPattern-(12-1);
                                                          else {
                                                          //if ( Gulliver == 1 ) { // Means the Building-Block order 8/12 is found somewhere i.e. NO MAXIMUM SKIP
                                                                                    // Order 4 [
                                                                    // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                                    // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:
                                                                    //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                                    //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                                    //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                                    //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                                    //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                                    //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                                    //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                                    //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                              count = cbPattern-4+1;
                                                                              while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                                        count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                              if ( count <= 0 ) {
                                                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                              }
                                                                              // Order 4 ]
                                                          }
                                                i = i + Gulliver;
                                                //GlobalI++; // Comment it, it is only for stats.
                                      } // while (i <= cbTarget-cbPattern)
                                      return(NULL);
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() ]
          } // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
                    } //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
          } //if ( cbPattern<4 )
}
char * Railgun_Sekireigan_Wolfram_9 (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
{
          char * pbTargetMax = pbTarget + cbTarget;
          register uint32_t ulHashPattern;
          register uint32_t ulHashTarget;
          signed long count;
          //signed long countSTATIC;

          unsigned char SINGLET;
          uint32_t Quadruplet2nd;
          uint32_t Quadruplet3rd;
          uint32_t Quadruplet4th;

          uint32_t AdvanceHopperGrass;

          uint32_t a, i, j;
//Global is next line already:
          //unsigned char bm_Horspool_Order2[256*256]; // BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
          uint32_t Gulliver; // or unsigned char or unsigned short

          //unsigned char bm_Hasherezade_HASH[1<<(HashTableSizeSekireiTchittoGritto-3)];
          uint32_t hash32;
          uint32_t hash32B;
          uint32_t hash32C;

          if (cbPattern > cbTarget) return(NULL);

          if ( cbPattern<4 ) {
```

```
		pbTarget = pbTarget+cbPattern;
			ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
			if ( cbPattern==3 ) {
				for ( ;; ) {
					if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
						if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
					}
					if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) {
						pbTarget++;
						if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
					}
					pbTarget++;
					if (pbTarget > pbTargetMax) return(NULL);
				}
			} else {
			}
			for ( ;; ) {
				if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) ) return((pbTarget-2));
				if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
				pbTarget++;
				if (pbTarget > pbTargetMax) return(NULL);
			}

		} else {
			if (cbTarget<HaystackThresholdSekireiTchittoGritto) { // This value is arbitrary (don't know how exactly), it ensures (at least must) better performance than 'Boyer_Moore_Horspool'.

			pbTarget = pbTarget+cbPattern;
			ulHashPattern = *(uint32_t *)(pbPattern);
			SINGLET = ulHashPattern & 0xFF;
			Quadruplet2nd = SINGLET<<8;
			Quadruplet3rd = SINGLET<<16;
			Quadruplet4th = SINGLET<<24;
			for ( ;; ) {
				AdvanceHopperGrass = 0;
				ulHashTarget = *(uint32_t *)(pbTarget-cbPattern);
				if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
					count = cbPattern-1;
					while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
						if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
						count--;
					}
					if ( count == 0) return((pbTarget-cbPattern));
				} else { // The goal here: to avoid memory accesses by stressing the registers.
					if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
						AdvanceHopperGrass++;
						if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
							AdvanceHopperGrass++;
							if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
						}
					}
				}
				AdvanceHopperGrass++;
				pbTarget = pbTarget + AdvanceHopperGrass;
				if (pbTarget > pbTargetMax) return(NULL);
			}

			} else { //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
		if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto ) {

		// BMH order 2:
		if ( cbPattern<=NeedleThreshold2vs4TchittoGritto ) {
				//countSTATIC = cbPattern-2-2;
				ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
				//ulHashTarget = *(uint32_t *)(pbPattern+cbPattern-4); // Last four bytes
				i=0;
				//for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
				//for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
				//for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
```

```
//Possible commenting of next line:
                                for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=1;
                                while (i <= cbTarget-cbPattern) {
                                        Gulliver = 1; // 'Gulliver' is the skip
            // Few thoughts regarding an excellent Skip Performance etude:
            // Something "still" considered crazy: using BITwise order 3, not pseudo order 3, though!
            // 2^24 = 16MB BYTEwise or 2^(24-3) = 2MB BITwise, when searching big haystacks e.g. Wikipedia 42GB with Kazahana then this 2MB lookup table seems not so atrocious.
            // The code is like:
            // ulHashTarget = *(uint32_t *)&pbTarget[i+cbPattern-4]; // One memory access instead of 2
            // if ( bm_Horspool_Order3[ulHashTarget>>8] == 0 ) Gulliver = cbPattern-(3-1); else
            // if ( bm_Horspool_Order3[ulHashTarget&0xFFFFFF] == 0 ) Gulliver = cbPattern-(3-1)-1; else
            // {
            // ...
            // }
                                        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]] != 0 ) {
                                                if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-2]] + bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-1]] != 2 ) Gulliver = cbPattern-(2-1)-2; else {
                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for remainder) when going under 0 in loop below:
                                                        // Order 4 [
                                                // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
                                                //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                        count = cbPattern-4+1;
                                                        //count = count-4; // Double-beauty here of already being checked 'ulHashTarget' and not polluting/repeating the final lookup below.
                                                        while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                        if ( count <= 0 ) {
                                                                return(pbTarget+i);
                                                        }
                                                        //if ( count <= 0 ) {
                                                        //        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                        //}
                                                        //else {
                                                        //        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+count-1]] == 0 ) Gulliver = count; // 1 or bigger, as it should
                                                        //}
                                                        // Order 4 ]
                                                        }
                                                } // Means AT LEAST one of the BBs is 0, enforce lower skip: MIN( cbPattern-(2-1)-1 , cbPattern-(2-1)-2 )
                                        } else Gulliver = cbPattern-(2-1);
                                        i = i + Gulliver;
                                        //GlobalI++; // Comment it, it is only for stats.
                                }
                                return(NULL);
            // BMH order 4, needle should be >=8:
            } else { //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                i=0;
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
                                // In line below we "hash" 4bytes to 2bytes i.e. 16bit table, how to compute TOTAL number of BBs, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox'
and Order=4 we have BBs = 11-4+1=8:
                                //"fast"
                                //"aste"
                                //"stes"
                                //"test"
                                //"est "
                                //"st f"
                                //"t fo"
                                //" fox"
```

```
                                //for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( *(unsigned short *)(pbPattern+j+0) + *(unsigned short *)(pbPattern+j+2) ) & ( (1<<16)-1 )]=1;
//Possible commenting of next line:
                                for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( (*(uint32_t *)(pbPattern+j+0)>>16)+(*(uint32_t *)(pbPattern+j+0)&0xFFFF) ) & ( (1<<16)-1 )]=1;
                                while (i <= cbTarget-cbPattern) {
                                        Gulliver = 1;
                                        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]&0xFFFF) ) & ( (1<<16)-1 )] != 0 ) {
                                                if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = cbPattern-(2-1)-2-4; else {
                                                                // Order 4 [
                                                        // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                        // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:

                                                        //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                        //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                        //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                        //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                        //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                        //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                        //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                        //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                count = cbPattern-4+1;
                                                                while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                        count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                if ( count <= 0 ) {
                                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                }
                                                                //else {
                                                                //      if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+count-1]>>16)+(*(uint32_t *)&pbTarget[i+count-1]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = count; // 1 or
bigger, as it should

                                                                //}
                                                                // Order 4 ]
                                                }
                                        } else Gulliver = cbPattern-(2-1)-2; // -2 because we check the 4 rightmost bytes not 2.
                                        i = i + Gulliver;
                                        //GlobalI++; // Comment it, it is only for stats.
                                }
                                return(NULL);
        } //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )

        } else { // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() [
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                i=0;
                                for (a=0; a < 1<<(HashTableSizeSekireiTchittoGritto-3); a++) {bm_Hasherezade_HASH[a]= 0;} // to-do: 'memset' if not optimized
                                // cbPattern - Order + 1 i.e. number of BBs for 11 'fastest fox' 11-8+1=4: 'fastest ', 'astest f', 'stest fo', 'test fox'
                                for (j=0; j < cbPattern-12+1; j++) {
                                        hash32 = (2166136261UL ^ *(uint32_t *)(pbPattern+j+0)) * 709607;
                                        hash32B = (2166136261UL ^ *(uint32_t *)(pbPattern+j+4)) * 709607;
                                        hash32C = (2166136261UL ^ *(uint32_t *)(pbPattern+j+8)) * 709607;
                                        hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                        hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                        hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                        bm_Hasherezade_HASH[hash32>>3]= bm_Hasherezade_HASH[hash32>>3] | (1<<(hash32&0x7));
                                }
                                while (i <= cbTarget-cbPattern) {
                                        Gulliver = 1; // Assume minimal jump as initial value.
                                        // The goal: to jump when the rightmost 8bytes (Order 8 Horspool) of window do not look like any of Needle prefixes i.e. are not to be found. This maximum jump equals cbPattern-(Order-1) or 11-(8-1)=4 for
'fastest fox' - a small one but for Needle 31 bytes the jump equals 31-(8-1)=24
                                        //GlobalHashSectionExecution++; // Comment it, it is only for stats.
                                                hash32 = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+0)) * 709607;
                                                hash32B = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+4)) * 709607;
                                                hash32C = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+8)) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                if ( (bm_Hasherezade_HASH[hash32>>3] & (1<<(hash32&0x7))) ==0 )     Gulliver = cbPattern-(12-1);
                                                else {
                                                //if ( Gulliver == 1 ) { // Means the Building-Block order 8/12 is found somewhere i.e. NO MAXIMUM SKIP
```

```
                            // Order 4 [
                        // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                        // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
                        //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                        //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                        //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                        //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                        //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                        //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                        //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                        //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                            count = cbPattern-4+1;
                            while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                            if ( count <= 0 ) {
                                if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                            }
                            // Order 4 ]
                    }
                i = i + Gulliver;
                //GlobalI++; // Comment it, it is only for stats.
            } // while (i <= cbTarget-cbPattern)
            return(NULL);
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() ]
    } // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
        } //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
    } //if ( cbPattern<4 )
}
char * Railgun_Sekireigan_Wolfram_0 (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
{
    char * pbTargetMax = pbTarget + cbTarget;
    register uint32_t ulHashPattern;
    register uint32_t ulHashTarget;
    signed long count;
    //signed long countSTATIC;

    unsigned char SINGLET;
    uint32_t Quadruplet2nd;
    uint32_t Quadruplet3rd;
    uint32_t Quadruplet4th;

    uint32_t AdvanceHopperGrass;

    uint32_t a, i, j;
//Global is next line already:
    //unsigned char bm_Horspool_Order2[256*256]; // BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
    uint32_t Gulliver; // or unsigned char or unsigned short

    //unsigned char bm_Hasherezade_HASH[1<<(HashTableSizeSekireiTchittoGritto-3)];
    uint32_t hash32;
    uint32_t hash32B;
    uint32_t hash32C;

    if (cbPattern > cbTarget) return(NULL);

    if ( cbPattern<4 ) {

    pbTarget = pbTarget+cbPattern;
        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
        if ( cbPattern==3 ) {
            for ( ;; ) {
                if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
                    if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
                }
                if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) {
                    pbTarget++;
                    if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
                }
```

```
                                        pbTarget++;
                                        if (pbTarget > pbTargetMax) return(NULL);
                                }
                        } else {
                        }
                        for ( ;; ) {
                                if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) ) return((pbTarget-2));
                                if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
                                pbTarget++;
                                if (pbTarget > pbTargetMax) return(NULL);
                        }

                } else {
                        if (cbTarget<HaystackThresholdSekireiTchittoGritto) { // This value is arbitrary (don't know how exactly), it ensures (at least must) better performance than 'Boyer_Moore_Horspool'.

                        pbTarget = pbTarget+cbPattern;
                        ulHashPattern = *(uint32_t *)(pbPattern);
                        SINGLET = ulHashPattern & 0xFF;
                        Quadruplet2nd = SINGLET<<8;
                        Quadruplet3rd = SINGLET<<16;
                        Quadruplet4th = SINGLET<<24;
                        for ( ;; ) {
                                AdvanceHopperGrass = 0;
                                ulHashTarget = *(uint32_t *)(pbTarget-cbPattern);
                                if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
                                        count = cbPattern-1;
                                        while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                                                if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
                                                count--;
                                        }
                                        if ( count == 0) return((pbTarget-cbPattern));
                                } else { // The goal here: to avoid memory accesses by stressing the registers.
                                        if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                                AdvanceHopperGrass++;
                                                if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                        AdvanceHopperGrass++;
                                                        if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                                }
                                        }
                                }
                                AdvanceHopperGrass++;
                                pbTarget = pbTarget + AdvanceHopperGrass;
                                if (pbTarget > pbTargetMax) return(NULL);
                        }

                        } else { //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
                if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto ) {

                // BMH order 2:
                if ( cbPattern<=NeedleThreshold2vs4TchittoGritto ) {
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(uint32_t *)(pbPattern+cbPattern-4); // Last four bytes
                                i=0;
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
//Possible commenting of next line:
                                for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=1;
                                while (i <= cbTarget-cbPattern) {
                                        Gulliver = 1; // 'Gulliver' is the skip
                // Few thoughts regarding an excellent Skip Performance etude:
                // Something "still" considered crazy: using BITwise order 3, not pseudo order 3, though!
                // 2^24 = 16MB BYTEwise or 2^(24-3) = 2MB BITwise, when searching big haystacks e.g. Wikipedia 42GB with Kazahana then this 2MB lookup table seems not so atrocious.
                // The code is like:
                // ulHashTarget = *(uint32_t *)&pbTarget[i+cbPattern-4]; // One memory access instead of 2
                // if ( bm_Horspool_Order3[ulHashTarget>>8] == 0 ) Gulliver = cbPattern-(3-1); else
                // if ( bm_Horspool_Order3[ulHashTarget&0xFFFFFF] == 0 ) Gulliver = cbPattern-(3-1)-1; else
```

```
		// {
		// ...
		// }
				if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]] != 0 ) {
					if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-2]] + bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-1]] != 2 ) Gulliver = cbPattern-(2-1)-2; else {
						if ( *(uint32_t *)&pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for remainder) when going under 0 in loop below:
						// Order 4 [
					// Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
					// Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
					//0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
					//1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
					//2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
					//3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
					//4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
					//5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
					//6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
					//7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
						count = cbPattern-4+1;
						//count = count-4; // Double-beauty here of already being checked 'ulHashTarget' and not polluting/repeating the final lookup below.
						while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
							count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
						if ( count <= 0 ) {
							return(pbTarget+i);
						}
						//if ( count <= 0 ) {
						//	if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
						//}
						//else {
						//	if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+count-1]] == 0 ) Gulliver = count; // 1 or bigger, as it should
						//}
						// Order 4 ]
						}
					} // Means AT LEAST one of the BBs is 0, enforce lower skip: MIN( cbPattern-(2-1)-1 , cbPattern-(2-1)-2 )
				} else Gulliver = cbPattern-(2-1);
				i = i + Gulliver;
				//GlobalI++; // Comment it, it is only for stats.
			}
			return(NULL);
		// BMH order 4, needle should be >=8:
		} else { //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )
			//countSTATIC = cbPattern-2-2;
			ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
			//ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
			i=0;
			//for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
			//for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
			//for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
			// In line below we "hash" 4bytes to 2bytes i.e. 16bit table, how to compute TOTAL number of BBs, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox'
and Order=4 we have BBs = 11-4+1=8:
			//"fast"
			//"aste"
			//"stes"
			//"test"
			//"est "
			//"st f"
			//"t fo"
			//" fox"
			//for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( *(unsigned short *)(pbPattern+j+0) + *(unsigned short *)(pbPattern+j+2) ) & ( (1<<16)-1 )]=1;
//Possible commenting of next line:
			for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( (*(uint32_t *)(pbPattern+j+0)>>16)+(*(uint32_t *)(pbPattern+j+0)&0xFFFF) ) & ( (1<<16)-1 )]=1;
			while (i <= cbTarget-cbPattern) {
				Gulliver = 1;
				if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]&0xFFFF) ) & ( (1<<16)-1 )] != 0 ) {
					if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = cbPattern-(2-1)-2-4; else {
						// Order 4 [
					// Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
					// Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:
```

```
                                                            //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                            //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                            //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                            //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                            //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                            //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                            //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                            //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                        count = cbPattern-4+1;
                                                                        while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                                    count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                        if ( count <= 0 ) {
                                                                                    if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                        }
                                                                        //else {
                                                                        //        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+count-1]>>16)+(*(uint32_t *)&pbTarget[i+count-1]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = count; // 1 or
bigger, as it should
                                                                        //}
                                                                        // Order 4 ]
                                                                }
                                                        } else Gulliver = cbPattern-(2-1)-2; // -2 because we check the 4 rightmost bytes not 2.
                                                        i = i + Gulliver;
                                                        //GlobalI++; // Comment it, it is only for stats.
                                                }
                                                return(NULL);
                                        } //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )

                                        } else { // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() [
                                                //countSTATIC = cbPattern-2-2;
                                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                                i=0;
                                                for (a=0; a < 1<<(HashTableSizeSekireiTchittoGritto-3); a++) {bm_Hasherezade_HASH[a]= 0;} // to-do: 'memset' if not optimized
                                                // cbPattern - Order + 1 i.e. number of BBs for 11 'fastest fox' 11-8+1=4: 'fastest ', 'astest f', 'stest fo', 'test fox'
                                                for (j=0; j < cbPattern-12+1; j++) {
                                                        hash32 = (2166136261UL ^ *(uint32_t *)(pbPattern+j+0)) * 709607;
                                                        hash32B = (2166136261UL ^ *(uint32_t *)(pbPattern+j+4)) * 709607;
                                                        hash32C = (2166136261UL ^ *(uint32_t *)(pbPattern+j+8)) * 709607;
                                                        hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                        hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                        hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                        bm_Hasherezade_HASH[hash32>>3]= bm_Hasherezade_HASH[hash32>>3] | (1<<(hash32&0x7));
                                                }
                                                while (i <= cbTarget-cbPattern) {
                                                        Gulliver = 1; // Assume minimal jump as initial value.
                                                        // The goal: to jump when the rightmost 8bytes (Order 8 Horspool) of window do not look like any of Needle prefixes i.e. are not to be found. This maximum jump equals cbPattern-(Order-1) or 11-(8-1)=4 for
'fastest fox' - a small one but for Needle 31 bytes the jump equals 31-(8-1)=24
                                                        //GlobalHashSectionExecution++; // Comment it, it is only for stats.
                                                                hash32 = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+0)) * 709607;
                                                                hash32B = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+4)) * 709607;
                                                                hash32C = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+8)) * 709607;
                                                                hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                                hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                                hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                                if ( (bm_Hasherezade_HASH[hash32>>3] & (1<<(hash32&0x7))) ==0 )     Gulliver = cbPattern-(12-1);
                                                                else {
                                                                //if ( Gulliver == 1 ) { // Means the Building-Block order 8/12 is found somewhere i.e. NO MAXIMUM SKIP
                                                                                // Order 4 [
                                                            // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                            // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:
                                                            //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                            //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                            //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                            //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                            //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                            //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                            //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
```

```
                                                                        //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                                count = cbPattern-4+1;
                                                                                while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                                        count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                                if ( count <= 0 ) {
                                                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                                }
                                                                                // Order 4 ]
                                                                        }
                                                                i = i + Gulliver;
                                                                //GlobalI++; // Comment it, it is only for stats.
                                                        } // while (i <= cbTarget-cbPattern)
                                                        return(NULL);
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() ]
                        } // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
                                } //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
                        } //if ( cbPattern<4 )
}
char * Railgun_Sekireigan_Wolfram_a (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
{
                        char * pbTargetMax = pbTarget + cbTarget;
                        register uint32_t ulHashPattern;
                        register uint32_t ulHashTarget;
                        signed long count;
                        //signed long countSTATIC;

                        unsigned char SINGLET;
                        uint32_t Quadruplet2nd;
                        uint32_t Quadruplet3rd;
                        uint32_t Quadruplet4th;

                        uint32_t AdvanceHopperGrass;

                        uint32_t a, i, j;
//Global is next line already:
                        //unsigned char bm_Horspool_Order2[256*256]; // BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
                        uint32_t Gulliver; // or unsigned char or unsigned short

                        //unsigned char bm_Hasherezade_HASH[1<<(HashTableSizeSekireiTchittoGritto-3)];
                        uint32_t hash32;
                        uint32_t hash32B;
                        uint32_t hash32C;

                        if (cbPattern > cbTarget) return(NULL);

                        if ( cbPattern<4 ) {

                        pbTarget = pbTarget+cbPattern;
                                        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
                                        if ( cbPattern==3 ) {
                                                        for ( ;; ) {
                                                                        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
                                                                                        if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
                                                                        }
                                                                        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) {
                                                                                        pbTarget++;
                                                                                        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
                                                                        }
                                                                        pbTarget++;
                                                                        if (pbTarget > pbTargetMax) return(NULL);
                                                        }
                                        } else {
                                        }
                                        for ( ;; ) {
                                                        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) ) return((pbTarget-2));
                                                        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
                                                        pbTarget++;
                                                        if (pbTarget > pbTargetMax) return(NULL);
                                        }
```

```
	} else {
		if (cbTarget<HaystackThresholdSekireiTchittoGritto) { // This value is arbitrary (don't know how exactly), it ensures (at least must) better performance than 'Boyer_Moore_Horspool'.

		pbTarget = pbTarget+cbPattern;
		ulHashPattern = *(uint32_t *)(pbPattern);
		SINGLET = ulHashPattern & 0xFF;
		Quadruplet2nd = SINGLET<<8;
		Quadruplet3rd = SINGLET<<16;
		Quadruplet4th = SINGLET<<24;
		for ( ;; ) {
			AdvanceHopperGrass = 0;
			ulHashTarget = *(uint32_t *)(pbTarget-cbPattern);
			if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
				count = cbPattern-1;
				while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
					if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
					count--;
				}
				if ( count == 0) return((pbTarget-cbPattern));
			} else { // The goal here: to avoid memory accesses by stressing the registers.
				if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
					AdvanceHopperGrass++;
					if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
						AdvanceHopperGrass++;
						if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
					}
				}
			}
			AdvanceHopperGrass++;
			pbTarget = pbTarget + AdvanceHopperGrass;
			if (pbTarget > pbTargetMax) return(NULL);
		}

		} else { //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
	if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto ) {

	// BMH order 2:
	if ( cbPattern<=NeedleThreshold2vs4TchittoGritto ) {
			//countSTATIC = cbPattern-2-2;
			ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
			//ulHashTarget = *(uint32_t *)(pbPattern+cbPattern-4); // Last four bytes
			i=0;
			//for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
			//for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
			//for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
//Possible commenting of next line:
			for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=1;
			while (i <= cbTarget-cbPattern) {
				Gulliver = 1; // 'Gulliver' is the skip
	// Few thoughts regarding an excellent Skip Performance etude:
	// Something "still" considered crazy: using BITwise order 3, not pseudo order 3, though!
	// 2^24 = 16MB BYTEwise or 2^(24-3) = 2MB BITwise, when searching big haystacks e.g. Wikipedia 42GB with Kazahana then this 2MB lookup table seems not so atrocious.
	// The code is like:
	// ulHashTarget = *(uint32_t *)&pbTarget[i+cbPattern-4]; // One memory access instead of 2
	// if ( bm_Horspool_Order3[ulHashTarget>>8] == 0 ) Gulliver = cbPattern-(3-1); else
	// if ( bm_Horspool_Order3[ulHashTarget&0xFFFFFF] == 0 ) Gulliver = cbPattern-(3-1)-1; else
	// {
	// ...
	// }
				if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]] != 0 ) {
					if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-2]] + bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-1]] != 2 ) Gulliver = cbPattern-(2-1)-2; else {
						if ( *(uint32_t *)&pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for remainder) when going under 0 in loop below:
						// Order 4 [
					// Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
					// Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
					//0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
					//1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
```

```
                                        //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                        //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                        //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                        //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                        //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                        //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                  count = cbPattern-4+1;
                                                  //count = count-4; // Double-beauty here of already being checked 'ulHashTarget' and not polluting/repeating the final lookup below.
                                                  while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                            count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                  if ( count <= 0 ) {
                                                            return(pbTarget+i);
                                                  }
                                                  //if ( count <= 0 ) {
                                                  //         if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                  //}
                                                  //else {
                                                  //         if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+count-1]] == 0 ) Gulliver = count; // 1 or bigger, as it should
                                                  //}
                                                  // Order 4 ]
                                                  }
                                        } // Means AT LEAST one of the BBs is 0, enforce lower skip: MIN( cbPattern-(2-1)-1 , cbPattern-(2-1)-2 )
                                } else Gulliver = cbPattern-(2-1);
                                i = i + Gulliver;
                                //GlobalI++; // Comment it, it is only for stats.
                        }
                        return(NULL);
            // BMH order 4, needle should be >=8:
            } else { //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )
                        //countSTATIC = cbPattern-2-2;
                        ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                        //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                        i=0;
                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                        //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
                        // In line below we "hash" 4bytes to 2bytes i.e. 16bit table, how to compute TOTAL number of BBs, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox'
and Order=4 we have BBs = 11-4+1=8:
                        //"fast"
                        //"aste"
                        //"stes"
                        //"test"
                        //"est "
                        //"st f"
                        //"t fo"
                        //" fox"
                        //for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( *(unsigned short *)(pbPattern+j+0) + *(unsigned short *)(pbPattern+j+2) ) & ( (1<<16)-1 )]=1;
//Possible commenting of next line:
                        for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( (*(uint32_t *)(pbPattern+j+0)>>16)+(*(uint32_t *)(pbPattern+j+0)&0xFFFF) ) & ( (1<<16)-1 )]=1;
                        while (i <= cbTarget-cbPattern) {
                                Gulliver = 1;
                                if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]&0xFFFF) ) & ( (1<<16)-1 )] != 0 ) {
                                        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = cbPattern-(2-1)-2-4; else {
                                                  // Order 4 [
                                                  // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                  // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:
                                                  //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                  //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                  //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                  //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                  //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                  //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                  //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                  //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                            count = cbPattern-4+1;
                                                            while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                      count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
```

```
                                                                        if ( count <= 0 ) {
                                                                                if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                        }
                                                                        //else {
                                                                        //        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+count-1]>>16)+(*(uint32_t *)&pbTarget[i+count-1]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = count; // 1 or
bigger, as it should
                                                                        //}
                                                                        // Order 4 ]
                                                                }
                                                        } else Gulliver = cbPattern-(2-1)-2; // -2 because we check the 4 rightmost bytes not 2.
                                                        i = i + Gulliver;
                                                        //GlobalI++; // Comment it, it is only for stats.
                                                }
                                                return(NULL);
                        } //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )

                        } else { // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() [
                                                //countSTATIC = cbPattern-2-2;
                                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                                i=0;
                                                for (a=0; a < 1<<(HashTableSizeSekireiTchittoGritto-3); a++) {bm_Hasherezade_HASH[a]= 0;} // to-do: 'memset' if not optimized
                                                // cbPattern - Order + 1 i.e. number of BBs for 11 'fastest fox' 11-8+1=4: 'fastest ', 'astest f', 'stest fo', 'test fox'
                                                for (j=0; j < cbPattern-12+1; j++) {
                                                        hash32 = (2166136261UL ^ *(uint32_t *)(pbPattern+j+0)) * 709607;
                                                        hash32B = (2166136261UL ^ *(uint32_t *)(pbPattern+j+4)) * 709607;
                                                        hash32C = (2166136261UL ^ *(uint32_t *)(pbPattern+j+8)) * 709607;
                                                        hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                        hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                        hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                        bm_Hasherezade_HASH[hash32>>3]= bm_Hasherezade_HASH[hash32>>3] | (1<<(hash32&0x7));
                                                }
                                                while (i <= cbTarget-cbPattern) {
                                                        Gulliver = 1; // Assume minimal jump as initial value.
                                                        // The goal: to jump when the rightmost 8bytes (Order 8 Horspool) of window do not look like any of Needle prefixes i.e. are not to be found. This maximum jump equals cbPattern-(Order-1) or 11-(8-1)=4 for
'fastest fox' - a small one but for Needle 31 bytes the jump equals 31-(8-1)=24
                                                        //GlobalHashSectionExecution++; // Comment it, it is only for stats.
                                                                hash32 = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+0)) * 709607;
                                                                hash32B = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+4)) * 709607;
                                                                hash32C = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+8)) * 709607;
                                                                hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                                hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                                hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                                if ( (bm_Hasherezade_HASH[hash32>>3] & (1<<(hash32&0x7))) ==0 )     Gulliver = cbPattern-(12-1);
                                                                else {
                                                                //if ( Gulliver == 1 ) { // Means the Building-Block order 8/12 is found somewhere i.e. NO MAXIMUM SKIP
                                                                                // Order 4 [
                                                                        // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                                        // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:

                                                                        //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                                        //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                                        //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                                        //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                                        //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                                        //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                                        //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                                        //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                                count = cbPattern-4+1;
                                                                                while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                                        count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                                if ( count <= 0 ) {
                                                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                                }
                                                                                // Order 4 ]
                                                                }
                                                        i = i + Gulliver;
                                                        //GlobalI++; // Comment it, it is only for stats.
```

```
                                } // while (i <= cbTarget-cbPattern)
                                return(NULL);
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() ]
            } // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
                        } //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
            } //if ( cbPattern<4 )
}
char * Railgun_Sekireigan_Wolfram_b (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
{
            char * pbTargetMax = pbTarget + cbTarget;
            register uint32_t ulHashPattern;
            register uint32_t ulHashTarget;
            signed long count;
            //signed long countSTATIC;

            unsigned char SINGLET;
            uint32_t Quadruplet2nd;
            uint32_t Quadruplet3rd;
            uint32_t Quadruplet4th;

            uint32_t AdvanceHopperGrass;

            uint32_t a, i, j;
//Global is next line already:
            //unsigned char bm_Horspool_Order2[256*256]; // BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
            uint32_t Gulliver; // or unsigned char or unsigned short

            //unsigned char bm_Hasherezade_HASH[1<<(HashTableSizeSekireiTchittoGritto-3)];
            uint32_t hash32;
            uint32_t hash32B;
            uint32_t hash32C;

            if (cbPattern > cbTarget) return(NULL);

            if ( cbPattern<4 ) {

            pbTarget = pbTarget+cbPattern;
                        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
                        if ( cbPattern==3 ) {
                                    for ( ;; ) {
                                                if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
                                                            if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
                                                }
                                                if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) {
                                                            pbTarget++;
                                                            if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
                                                }
                                                pbTarget++;
                                                if (pbTarget > pbTargetMax) return(NULL);
                                    }
                        } else {
                        }
                        for ( ;; ) {
                                    if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) ) return((pbTarget-2));
                                    if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
                                    pbTarget++;
                                    if (pbTarget > pbTargetMax) return(NULL);
                        }

            } else {
                        if (cbTarget<HaystackThresholdSekireiTchittoGritto) { // This value is arbitrary (don't know how exactly), it ensures (at least must) better performance than 'Boyer_Moore_Horspool'.

                        pbTarget = pbTarget+cbPattern;
                        ulHashPattern = *(uint32_t *)(pbPattern);
                        SINGLET = ulHashPattern & 0xFF;
                        Quadruplet2nd = SINGLET<<8;
                        Quadruplet3rd = SINGLET<<16;
                        Quadruplet4th = SINGLET<<24;
                        for ( ;; ) {
```

```
                                AdvanceHopperGrass = 0;
                                ulHashTarget = *(uint32_t *)(pbTarget-cbPattern);
                                if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
                                          count = cbPattern-1;
                                          while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                                                    if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
                                                    count--;
                                          }
                                          if ( count == 0) return((pbTarget-cbPattern));
                                } else { // The goal here: to avoid memory accesses by stressing the registers.
                                          if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                                    AdvanceHopperGrass++;
                                                    if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                              AdvanceHopperGrass++;
                                                              if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                                    }
                                          }
                                }
                                AdvanceHopperGrass++;
                                pbTarget = pbTarget + AdvanceHopperGrass;
                                if (pbTarget > pbTargetMax) return(NULL);
                          }

                          } else { //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
          if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto ) {

          // BMH order 2:
          if ( cbPattern<=NeedleThreshold2vs4TchittoGritto ) {
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(uint32_t *)(pbPattern+cbPattern-4); // Last four bytes
                                i=0;
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
//Possible commenting of next line:
                                for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=1;
                                while (i <= cbTarget-cbPattern) {
                                          Gulliver = 1; // 'Gulliver' is the skip
          // Few thoughts regarding an excellent Skip Performance etude:
          // Something "still" considered crazy: using BITwise order 3, not pseudo order 3, though!
          // 2^24 = 16MB BYTEwise or 2^(24-3) = 2MB BITwise, when searching big haystacks e.g. Wikipedia 42GB with Kazahana then this 2MB lookup table seems not so atrocious.
          // The code is like:
          // ulHashTarget = *(uint32_t *)&pbTarget[i+cbPattern-4]; // One memory access instead of 2
          // if ( bm_Horspool_Order3[ulHashTarget>>8] == 0 ) Gulliver = cbPattern-(3-1); else
          // if ( bm_Horspool_Order3[ulHashTarget&0xFFFFFF] == 0 ) Gulliver = cbPattern-(3-1)-1; else
          // {
          // ...
          // }
                                          if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]] != 0 ) {
                                                    if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-2]] + bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-1]] != 2 ) Gulliver = cbPattern-(2-1)-2; else {
                                                              if ( *(uint32_t *)&pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for remainder) when going under 0 in loop below:
                                                              // Order 4 [
                                                    // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                    // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
                                                    //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                    //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                    //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                    //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                    //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                    //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                    //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                    //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                              count = cbPattern-4+1;
                                                              //count = count-4; // Double-beauty here of already being checked 'ulHashTarget' and not polluting/repeating the final lookup below.
                                                              while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                        count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                              if ( count <= 0 ) {
```

```
                                                                        return(pbTarget+i);
                                                                }
                                                                //if ( count <= 0 ) {
                                                                //        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                //}
                                                                //else {
                                                                //        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+count-1]] == 0 ) Gulliver = count; // 1 or bigger, as it should
                                                                //}
                                                                // Order 4 ]
                                                                }
                                                        } // Means AT LEAST one of the BBs is 0, enforce lower skip: MIN( cbPattern-(2-1)-1 , cbPattern-(2-1)-2 )
                                                } else Gulliver = cbPattern-(2-1);
                                                i = i + Gulliver;
                                                //GlobalI++; // Comment it, it is only for stats.
                                        }
                                        return(NULL);
                        // BMH order 4, needle should be >=8:
                        } else { //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )
                                        //countSTATIC = cbPattern-2-2;
                                        ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                        //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                        i=0;
                                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                                        //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
                                        // In line below we "hash" 4bytes to 2bytes i.e. 16bit table, how to compute TOTAL number of BBs, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox'
and Order=4 we have BBs = 11-4+1=8:
                                        //"fast"
                                        //"aste"
                                        //"stes"
                                        //"test"
                                        //"est "
                                        //"st f"
                                        //"t fo"
                                        //" fox"
                                        //for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( *(unsigned short *)(pbPattern+j+0) + *(unsigned short *)(pbPattern+j+2) ) & ( (1<<16)-1 )]=1;
//Possible commenting of next line:
                                        for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( (*(uint32_t *)(pbPattern+j+0)>>16)+(*(uint32_t *)(pbPattern+j+0)&0xFFFF) ) & ( (1<<16)-1 )]=1;
                                        while (i <= cbTarget-cbPattern) {
                                                Gulliver = 1;
                                                if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]&0xFFFF) ) & ( (1<<16)-1 )] != 0 ) {
                                                        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = cbPattern-(2-1)-2-4; else {
                                                                // Order 4 [
                                                        // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                        // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:
                                                        //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                        //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                        //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                        //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                        //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                        //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                        //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                        //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                count = cbPattern-4+1;
                                                                while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                        count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                if ( count <= 0 ) {
                                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                }
                                                                //else {
                                                                //        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+count-1]>>16)+(*(uint32_t *)&pbTarget[i+count-1]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = count; // 1 or
bigger, as it should
                                                                //}
                                                                // Order 4 ]
                                                        }
                                                } else Gulliver = cbPattern-(2-1)-2; // -2 because we check the 4 rightmost bytes not 2.
                                                i = i + Gulliver;
```

```
                                        //GlobalI++; // Comment it, it is only for stats.
                                }
                                return(NULL);
            } //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )

            } else { // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() [
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                i=0;
                                for (a=0; a < 1<<(HashTableSizeSekireiTchittoGritto-3); a++) {bm_Hasherezade_HASH[a]= 0;} // to-do: 'memset' if not optimized
                                // cbPattern - Order + 1 i.e. number of BBs for 11 'fastest fox' 11-8+1=4: 'fastest ', 'astest f', 'stest fo', 'test fox'
                                for (j=0; j < cbPattern-12+1; j++) {
                                        hash32 = (2166136261UL ^ *(uint32_t *)(pbPattern+j+0)) * 709607;
                                        hash32B = (2166136261UL ^ *(uint32_t *)(pbPattern+j+4)) * 709607;
                                        hash32C = (2166136261UL ^ *(uint32_t *)(pbPattern+j+8)) * 709607;
                                        hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                        hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                        hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                        bm_Hasherezade_HASH[hash32>>3]= bm_Hasherezade_HASH[hash32>>3] | (1<<(hash32&0x7));
                                }
                                while (i <= cbTarget-cbPattern) {
                                        Gulliver = 1; // Assume minimal jump as initial value.
                                        // The goal: to jump when the rightmost 8bytes (Order 8 Horspool) of window do not look like any of Needle prefixes i.e. are not to be found. This maximum jump equals cbPattern-(Order-1) or 11-(8-1)=4 for
'fastest fox' - a small one but for Needle 31 bytes the jump equals 31-(8-1)=24
                                        //GlobalHashSectionExecution++; // Comment it, it is only for stats.
                                                hash32 = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+0)) * 709607;
                                                hash32B = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+4)) * 709607;
                                                hash32C = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+8)) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                if ( (bm_Hasherezade_HASH[hash32>>3] & (1<<(hash32&0x7))) ==0 )     Gulliver = cbPattern-(12-1);
                                                else {
                                                //if ( Gulliver == 1 ) { // Means the Building-Block order 8/12 is found somewhere i.e. NO MAXIMUM SKIP
                                                                        // Order 4 [
                                                        // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                        // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:

                                                        //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                        //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                        //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                        //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                        //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                        //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                        //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                        //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                count = cbPattern-4+1;
                                                                while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                        count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                if ( count <= 0 ) {
                                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                }
                                                                // Order 4 ]
                                                }
                                        i = i + Gulliver;
                                        //GlobalI++; // Comment it, it is only for stats.
                                } // while (i <= cbTarget-cbPattern)
                                return(NULL);
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() ]
            } // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
                        } //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
            } //if ( cbPattern<4 )
}
char * Railgun_Sekireigan_Wolfram_c (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
{
            char * pbTargetMax = pbTarget + cbTarget;
            register uint32_t ulHashPattern;
```

```
	register uint32_t ulHashTarget;
	signed long count;
	//signed long countSTATIC;

	unsigned char SINGLET;
	uint32_t Quadruplet2nd;
	uint32_t Quadruplet3rd;
	uint32_t Quadruplet4th;

	uint32_t AdvanceHopperGrass;

	uint32_t a, i, j;
//Global is next line already:
	//unsigned char bm_Horspool_Order2[256*256]; // BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
	uint32_t Gulliver; // or unsigned char or unsigned short

	//unsigned char bm_Hasherezade_HASH[1<<(HashTableSizeSekireiTchittoGritto-3)];
	uint32_t hash32;
	uint32_t hash32B;
	uint32_t hash32C;

	if (cbPattern > cbTarget) return(NULL);

	if ( cbPattern<4 ) {

	pbTarget = pbTarget+cbPattern;
		ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
		if ( cbPattern==3 ) {
			for ( ;; ) {
				if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
					if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
				}
				if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) {
					pbTarget++;
					if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
				}
				pbTarget++;
				if (pbTarget > pbTargetMax) return(NULL);
			}
		} else {
		}
		for ( ;; ) {
			if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) ) return((pbTarget-2));
			if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
			pbTarget++;
			if (pbTarget > pbTargetMax) return(NULL);
		}

	} else {
		if (cbTarget<HaystackThresholdSekireiTchittoGritto) { // This value is arbitrary (don't know how exactly), it ensures (at least must) better performance than 'Boyer_Moore_Horspool'.

		pbTarget = pbTarget+cbPattern;
		ulHashPattern = *(uint32_t *)(pbPattern);
		SINGLET = ulHashPattern & 0xFF;
		Quadruplet2nd = SINGLET<<8;
		Quadruplet3rd = SINGLET<<16;
		Quadruplet4th = SINGLET<<24;
		for ( ;; ) {
			AdvanceHopperGrass = 0;
			ulHashTarget = *(uint32_t *)(pbTarget-cbPattern);
			if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
				count = cbPattern-1;
				while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
					if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
					count--;
				}
				if ( count == 0) return((pbTarget-cbPattern));
			} else { // The goal here: to avoid memory accesses by stressing the registers.
				if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
```

```
                                                AdvanceHopperGrass++;
                                                if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                          AdvanceHopperGrass++;
                                                          if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                                }
                                          }
                                }
                                AdvanceHopperGrass++;
                                pbTarget = pbTarget + AdvanceHopperGrass;
                                if (pbTarget > pbTargetMax) return(NULL);
                      }

                      } else { //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
            if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto ) {

            // BMH order 2:
            if ( cbPattern<=NeedleThreshold2vs4TchittoGritto ) {
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(uint32_t *)(pbPattern+cbPattern-4); // Last four bytes
                                i=0;
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
//Possible commenting of next line:
                                for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=1;
                                while (i <= cbTarget-cbPattern) {
                                          Gulliver = 1; // 'Gulliver' is the skip
            // Few thoughts regarding an excellent Skip Performance etude:
            // Something "still" considered crazy: using BITwise order 3, not pseudo order 3, though!
            // 2^24 = 16MB BYTEwise or 2^(24-3) = 2MB BITwise, when searching big haystacks e.g. Wikipedia 42GB with Kazahana then this 2MB lookup table seems not so atrocious.
            // The code is like:
            // ulHashTarget = *(uint32_t *)&pbTarget[i+cbPattern-4]; // One memory access instead of 2
            // if ( bm_Horspool_Order3[ulHashTarget>>8] == 0 ) Gulliver = cbPattern-(3-1); else
            // if ( bm_Horspool_Order3[ulHashTarget&0xFFFFFF] == 0 ) Gulliver = cbPattern-(3-1)-1; else
            // {
            // ...
            // }
                                          if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]] != 0 ) {
                                                if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-2]] + bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-1]] != 2 ) Gulliver = cbPattern-(2-1)-2; else {
                                                          if ( *(uint32_t *)&pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for remainder) when going under 0 in loop below:
                                                          // Order 4 [
                                                // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
                                                //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                          count = cbPattern-4+1;
                                                          //count = count-4; // Double-beauty here of already being checked 'ulHashTarget' and not polluting/repeating the final lookup below.
                                                          while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                    count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                          if ( count <= 0 ) {
                                                                    return(pbTarget+i);
                                                          }
                                                          //if ( count <= 0 ) {
                                                          //        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                          //}
                                                          //else {
                                                          //        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+count-1]] == 0 ) Gulliver = count; // 1 or bigger, as it should
                                                          //}
                                                          // Order 4 ]
                                                          }
                                                } // Means AT LEAST one of the BBs is 0, enforce lower skip: MIN( cbPattern-(2-1)-1 , cbPattern-(2-1)-2 )
```

```
                                        } else Gulliver = cbPattern-(2-1);
                                        i = i + Gulliver;
                                        //GlobalI++; // Comment it, it is only for stats.
                                }
                                return(NULL);
                // BMH order 4, needle should be >=8:
                } else { //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                i=0;
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
                                // In line below we "hash" 4bytes to 2bytes i.e. 16bit table, how to compute TOTAL number of BBs, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox'
and Order=4 we have BBs = 11-4+1=8:
                                //"fast"
                                //"aste"
                                //"stes"
                                //"test"
                                //"est "
                                //"st f"
                                //"t fo"
                                //" fox"
                                //for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( *(unsigned short *)(pbPattern+j+0) + *(unsigned short *)(pbPattern+j+2) ) & ( (1<<16)-1 )]=1;
//Possible commenting of next line:
                                for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( (*(uint32_t *)(pbPattern+j+0)>>16)+(*(uint32_t *)(pbPattern+j+0)&0xFFFF) ) & ( (1<<16)-1 )]=1;
                                while (i <= cbTarget-cbPattern) {
                                        Gulliver = 1;
                                        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]&0xFFFF) ) & ( (1<<16)-1 )] != 0 ) {
                                                if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = cbPattern-(2-1)-2-4; else {
                                                                // Order 4 [
                                                        // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                        // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:

                                                        //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                        //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                        //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                        //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                        //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                        //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                        //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                        //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                count = cbPattern-4+1;
                                                                while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                        count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                if ( count <= 0 ) {
                                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                }
                                                                //else {
                                                                //      if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+count-1]>>16)+(*(uint32_t *)&pbTarget[i+count-1]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = count; // 1 or
bigger, as it should

                                                                //}
                                                                // Order 4 ]
                                                }
                                        } else Gulliver = cbPattern-(2-1)-2; // -2 because we check the 4 rightmost bytes not 2.
                                        i = i + Gulliver;
                                        //GlobalI++; // Comment it, it is only for stats.
                                }
                                return(NULL);
                } //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )

                } else { // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() [
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                i=0;
```

```
                                        for (a=0; a < 1<<(HashTableSizeSekireiTchittoGritto-3); a++) {bm_Hasherezade_HASH[a]= 0;} // to-do: 'memset' if not optimized
                                        // cbPattern - Order + 1 i.e. number of BBs for 11 'fastest fox' 11-8+1=4: 'fastest ', 'astest f', 'stest fo', 'test fox'
                                        for (j=0; j < cbPattern-12+1; j++) {
                                                    hash32 = (2166136261UL ^ *(uint32_t *)(pbPattern+j+0)) * 709607;
                                                    hash32B = (2166136261UL ^ *(uint32_t *)(pbPattern+j+4)) * 709607;
                                                    hash32C = (2166136261UL ^ *(uint32_t *)(pbPattern+j+8)) * 709607;
                                                    hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                    hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                    hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                    bm_Hasherezade_HASH[hash32>>3]= bm_Hasherezade_HASH[hash32>>3] | (1<<(hash32&0x7));
                                        }
                                        while (i <= cbTarget-cbPattern) {
                                                    Gulliver = 1; // Assume minimal jump as initial value.
                                                    // The goal: to jump when the rightmost 8bytes (Order 8 Horspool) of window do not look like any of Needle prefixes i.e. are not to be found. This maximum jump equals cbPattern-(Order-1) or 11-(8-1)=4 for
'fastest fox' - a small one but for Needle 31 bytes the jump equals 31-(8-1)=24
                                                    //GlobalHashSectionExecution++; // Comment it, it is only for stats.
                                                                hash32 = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+0)) * 709607;
                                                                hash32B = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+4)) * 709607;
                                                                hash32C = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+8)) * 709607;
                                                                hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                                hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                                hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                                if ( (bm_Hasherezade_HASH[hash32>>3] & (1<<(hash32&0x7))) ==0 )     Gulliver = cbPattern-(12-1);
                                                                else {
                                                                //if ( Gulliver == 1 ) { // Means the Building-Block order 8/12 is found somewhere i.e. NO MAXIMUM SKIP
                                                                                // Order 4 [
                                                                            // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                                            // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:
                                                                            //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                                            //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                                            //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                                            //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                                            //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                                            //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                                            //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                                            //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                                        count = cbPattern-4+1;
                                                                                        while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                                                    count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                                        if ( count <= 0 ) {
                                                                                                    if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                                        }
                                                                                        // Order 4 ]
                                                                                }
                                                    i = i + Gulliver;
                                                    //GlobalI++; // Comment it, it is only for stats.
                                        } // while (i <= cbTarget-cbPattern)
                                        return(NULL);
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() ]
            } // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
                        } //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
            } //if ( cbPattern<4 )
}
char * Railgun_Sekireigan_Wolfram_d (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
{
            char * pbTargetMax = pbTarget + cbTarget;
            register uint32_t ulHashPattern;
            register uint32_t ulHashTarget;
            signed long count;
            //signed long countSTATIC;

            unsigned char SINGLET;
            uint32_t Quadruplet2nd;
            uint32_t Quadruplet3rd;
            uint32_t Quadruplet4th;

            uint32_t AdvanceHopperGrass;
```

```
	uint32_t a, i, j;
//Global is next line already:
	//unsigned char bm_Horspool_Order2[256*256]; // BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
	uint32_t Gulliver; // or unsigned char or unsigned short

	//unsigned char bm_Hasherezade_HASH[1<<(HashTableSizeSekireiTchittoGritto-3)];
	uint32_t hash32;
	uint32_t hash32B;
	uint32_t hash32C;

	if (cbPattern > cbTarget) return(NULL);

	if ( cbPattern<4 ) {

	pbTarget = pbTarget+cbPattern;
		ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
		if ( cbPattern==3 ) {
			for ( ;; ) {
				if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
					if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
				}
				if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) {
					pbTarget++;
					if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
				}
				pbTarget++;
				if (pbTarget > pbTargetMax) return(NULL);
			}
		} else {
		}
		for ( ;; ) {
			if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) ) return((pbTarget-2));
			if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
			pbTarget++;
			if (pbTarget > pbTargetMax) return(NULL);
		}

	} else {
		if (cbTarget<HaystackThresholdSekireiTchittoGritto) { // This value is arbitrary (don't know how exactly), it ensures (at least must) better performance than 'Boyer_Moore_Horspool'.

		pbTarget = pbTarget+cbPattern;
		ulHashPattern = *(uint32_t *)(pbPattern);
		SINGLET = ulHashPattern & 0xFF;
		Quadruplet2nd = SINGLET<<8;
		Quadruplet3rd = SINGLET<<16;
		Quadruplet4th = SINGLET<<24;
		for ( ;; ) {
			AdvanceHopperGrass = 0;
			ulHashTarget = *(uint32_t *)(pbTarget-cbPattern);
			if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
				count = cbPattern-1;
				while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
					if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
					count--;
				}
				if ( count == 0) return((pbTarget-cbPattern));
			} else { // The goal here: to avoid memory accesses by stressing the registers.
				if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
					AdvanceHopperGrass++;
					if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
						AdvanceHopperGrass++;
						if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
					}
				}
			}
			AdvanceHopperGrass++;
			pbTarget = pbTarget + AdvanceHopperGrass;
			if (pbTarget > pbTargetMax) return(NULL);
		}
```

```
                        } else { //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
                if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto ) {

                // BMH order 2:
                if ( cbPattern<=NeedleThreshold2vs4TchittoGritto ) {
                                        //countSTATIC = cbPattern-2-2;
                                        ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                        //ulHashTarget = *(uint32_t *)(pbPattern+cbPattern-4); // Last four bytes
                                        i=0;
                                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                                        //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
//Possible commenting of next line:
                                        for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=1;
                                        while (i <= cbTarget-cbPattern) {
                                                        Gulliver = 1; // 'Gulliver' is the skip
                // Few thoughts regarding an excellent Skip Performance etude:
                // Something "still" considered crazy: using BITwise order 3, not pseudo order 3, though!
                // 2^24 = 16MB BYTEwise or 2^(24-3) = 2MB BITwise, when searching big haystacks e.g. Wikipedia 42GB with Kazahana then this 2MB lookup table seems not so atrocious.
                // The code is like:
                // ulHashTarget = *(uint32_t *)&pbTarget[i+cbPattern-4]; // One memory access instead of 2
                // if ( bm_Horspool_Order3[ulHashTarget>>8] == 0 ) Gulliver = cbPattern-(3-1); else
                // if ( bm_Horspool_Order3[ulHashTarget&0xFFFFFF] == 0 ) Gulliver = cbPattern-(3-1)-1; else
                // {
                // ...
                // }
                                                        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]] != 0 ) {
                                                                if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-2]] + bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-1]] != 2 ) Gulliver = cbPattern-(2-1)-2; else {
                                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for remainder) when going under 0 in loop below:
                                                                        // Order 4 [
                                                                // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                                // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
                                                                //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                                //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                                //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                                //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                                //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                                //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                                //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                                //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                        count = cbPattern-4+1;
                                                                        //count = count-4; // Double-beauty here of already being checked 'ulHashTarget' and not polluting/repeating the final lookup below.
                                                                        while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                                count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                        if ( count <= 0 ) {
                                                                                return(pbTarget+i);
                                                                        }
                                                                        //if ( count <= 0 ) {
                                                                        //        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                        //}
                                                                        //else {
                                                                        //        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+count-1]] == 0 ) Gulliver = count; // 1 or bigger, as it should
                                                                        //}
                                                                        // Order 4 ]
                                                                        }
                                                                } // Means AT LEAST one of the BBs is 0, enforce lower skip: MIN( cbPattern-(2-1)-1 , cbPattern-(2-1)-2 )
                                                        } else Gulliver = cbPattern-(2-1);
                                                        i = i + Gulliver;
                                                        //GlobalI++; // Comment it, it is only for stats.
                                        }
                                        return(NULL);
                // BMH order 4, needle should be >=8:
                } else { //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )
                                        //countSTATIC = cbPattern-2-2;
                                        ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                        //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                        i=0;
```

```
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
                                // In line below we "hash" 4bytes to 2bytes i.e. 16bit table, how to compute TOTAL number of BBs, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox'
and Order=4 we have BBs = 11-4+1=8:
                                //"fast"
                                //"aste"
                                //"stes"
                                //"test"
                                //"est "
                                //"st f"
                                //"t fo"
                                //" fox"
                                //for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( *(unsigned short *)(pbPattern+j+0) + *(unsigned short *)(pbPattern+j+2) ) & ( (1<<16)-1 )]=1;
//Possible commenting of next line:
                                for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( (*(uint32_t *)(pbPattern+j+0)>>16)+(*(uint32_t *)(pbPattern+j+0)&0xFFFF) ) & ( (1<<16)-1 )]=1;
                                while (i <= cbTarget-cbPattern) {
                                        Gulliver = 1;
                                        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]&0xFFFF) ) & ( (1<<16)-1 )] != 0 ) {
                                                if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = cbPattern-(2-1)-2-4; else {
                                                        // Order 4 [
                                                // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:
                                                //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                        count = cbPattern-4+1;
                                                        while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                        if ( count <= 0 ) {
                                                                if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                        }
                                                        //else {
                                                        //        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+count-1]>>16)+(*(uint32_t *)&pbTarget[i+count-1]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = count; // 1 or
bigger, as it should
                                                        //}
                                                        // Order 4 ]
                                                }
                                        } else Gulliver = cbPattern-(2-1)-2; // -2 because we check the 4 rightmost bytes not 2.
                                        i = i + Gulliver;
                                        //GlobalI++; // Comment it, it is only for stats.
                                }
                                return(NULL);
        } //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )

        } else { // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() [
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                i=0;
                                for (a=0; a < 1<<(HashTableSizeSekireiTchittoGritto-3); a++) {bm_Hasherezade_HASH[a]= 0;} // to-do: 'memset' if not optimized
                                // cbPattern - Order + 1 i.e. number of BBs for 11 'fastest fox' 11-8+1=4: 'fastest ', 'astest f', 'stest fo', 'test fox'
                                for (j=0; j < cbPattern-12+1; j++) {
                                        hash32 = (2166136261UL ^ *(uint32_t *)(pbPattern+j+0)) * 709607;
                                        hash32B = (2166136261UL ^ *(uint32_t *)(pbPattern+j+4)) * 709607;
                                        hash32C = (2166136261UL ^ *(uint32_t *)(pbPattern+j+8)) * 709607;
                                        hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                        hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                        hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                        bm_Hasherezade_HASH[hash32>>3]= bm_Hasherezade_HASH[hash32>>3] | (1<<(hash32&0x7));
                                }
```

```
                                while (i <= cbTarget-cbPattern) {
                                        Gulliver = 1; // Assume minimal jump as initial value.
                                        // The goal: to jump when the rightmost 8bytes (Order 8 Horspool) of window do not look like any of Needle prefixes i.e. are not to be found. This maximum jump equals cbPattern-(Order-1) or 11-(8-1)=4 for
'fastest fox' - a small one but for Needle 31 bytes the jump equals 31-(8-1)=24
                                        //GlobalHashSectionExecution++; // Comment it, it is only for stats.
                                                hash32 = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+0)) * 709607;
                                                hash32B = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+4)) * 709607;
                                                hash32C = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+8)) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                if ( (bm_Hasherezade_HASH[hash32>>3] & (1<<(hash32&0x7))) ==0 )     Gulliver = cbPattern-(12-1);
                                                else {
                                                //if ( Gulliver == 1 ) { // Means the Building-Block order 8/12 is found somewhere i.e. NO MAXIMUM SKIP
                                                                // Order 4 [
                                                        // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                        // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:

                                                        //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                        //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                        //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                        //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                        //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                        //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                        //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                        //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                count = cbPattern-4+1;
                                                                while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                        count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                if ( count <= 0 ) {
                                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                }
                                                                // Order 4 ]
                                                }
                                        i = i + Gulliver;
                                        //GlobalI++; // Comment it, it is only for stats.
                                } // while (i <= cbTarget-cbPattern)
                                return(NULL);
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() ]
        } // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
                } //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
        } //if ( cbPattern<4 )
}
char * Railgun_Sekireigan_Wolfram_e (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
{
        char * pbTargetMax = pbTarget + cbTarget;
        register uint32_t ulHashPattern;
        register uint32_t ulHashTarget;
        signed long count;
        //signed long countSTATIC;

        unsigned char SINGLET;
        uint32_t Quadruplet2nd;
        uint32_t Quadruplet3rd;
        uint32_t Quadruplet4th;

        uint32_t AdvanceHopperGrass;

        uint32_t a, i, j;
//Global is next line already:
        //unsigned char bm_Horspool_Order2[256*256]; // BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
        uint32_t Gulliver; // or unsigned char or unsigned short

        //unsigned char bm_Hasherezade_HASH[1<<(HashTableSizeSekireiTchittoGritto-3)];
        uint32_t hash32;
        uint32_t hash32B;
        uint32_t hash32C;

        if (cbPattern > cbTarget) return(NULL);
```

```
if ( cbPattern<4 ) {

pbTarget = pbTarget+cbPattern;
        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
        if ( cbPattern==3 ) {
                for ( ;; ) {
                        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
                                if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
                        }
                        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) {
                                pbTarget++;
                                if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
                        }
                        pbTarget++;
                        if (pbTarget > pbTargetMax) return(NULL);
                }
        } else {
        }
        for ( ;; ) {
                if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) ) return((pbTarget-2));
                if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
                pbTarget++;
                if (pbTarget > pbTargetMax) return(NULL);
        }

} else {
        if (cbTarget<HaystackThresholdSekireiTchittoGritto) { // This value is arbitrary (don't know how exactly), it ensures (at least must) better performance than 'Boyer_Moore_Horspool'.

        pbTarget = pbTarget+cbPattern;
        ulHashPattern = *(uint32_t *)(pbPattern);
        SINGLET = ulHashPattern & 0xFF;
        Quadruplet2nd = SINGLET<<8;
        Quadruplet3rd = SINGLET<<16;
        Quadruplet4th = SINGLET<<24;
        for ( ;; ) {
                AdvanceHopperGrass = 0;
                ulHashTarget = *(uint32_t *)(pbTarget-cbPattern);
                if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
                        count = cbPattern-1;
                        while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                                if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
                                count--;
                        }
                        if ( count == 0) return((pbTarget-cbPattern));
                } else { // The goal here: to avoid memory accesses by stressing the registers.
                        if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                AdvanceHopperGrass++;
                                if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                        AdvanceHopperGrass++;
                                        if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                }
                        }
                }
                AdvanceHopperGrass++;
                pbTarget = pbTarget + AdvanceHopperGrass;
                if (pbTarget > pbTargetMax) return(NULL);
        }

        } else { //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto ) {

// BMH order 2:
if ( cbPattern<=NeedleThreshold2vs4TchittoGritto ) {
                //countSTATIC = cbPattern-2-2;
                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                //ulHashTarget = *(uint32_t *)(pbPattern+cbPattern-4); // Last four bytes
                i=0;
                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
```

```
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
//Possible commenting of next line:
                                for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=1;
                                while (i <= cbTarget-cbPattern) {
                                            Gulliver = 1; // 'Gulliver' is the skip
            // Few thoughts regarding an excellent Skip Performance etude:
            // Something "still" considered crazy: using BITwise order 3, not pseudo order 3, though!
            // 2^24 = 16MB BYTEwise or 2^(24-3) = 2MB BITwise, when searching big haystacks e.g. Wikipedia 42GB with Kazahana then this 2MB lookup table seems not so atrocious.
            // The code is like:
            // ulHashTarget = *(uint32_t *)&pbTarget[i+cbPattern-4]; // One memory access instead of 2
            // if ( bm_Horspool_Order3[ulHashTarget>>8] == 0 ) Gulliver = cbPattern-(3-1); else
            // if ( bm_Horspool_Order3[ulHashTarget&0xFFFFFF] == 0 ) Gulliver = cbPattern-(3-1)-1; else
            // {
            // ...
            // }
                                            if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]] != 0 ) {
                                                        if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-2]] + bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-1]] != 2 ) Gulliver = cbPattern-(2-1)-2; else {
                                                                    if ( *(uint32_t *)&pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for remainder) when going under 0 in loop below:
                                                                    // Order 4 [
                                                        // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                        // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
                                                        //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                        //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                        //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                        //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                        //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                        //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                        //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                        //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                    count = cbPattern-4+1;
                                                                    //count = count-4; // Double-beauty here of already being checked 'ulHashTarget' and not polluting/repeating the final lookup below.
                                                                    while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                                count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                    if ( count <= 0 ) {
                                                                                return(pbTarget+i);
                                                                    }
                                                                    //if ( count <= 0 ) {
                                                                    //          if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                    //}
                                                                    //else {
                                                                    //          if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+count-1]] == 0 ) Gulliver = count; // 1 or bigger, as it should
                                                                    //}
                                                                    // Order 4 ]
                                                                    }
                                                        } // Means AT LEAST one of the BBs is 0, enforce lower skip: MIN( cbPattern-(2-1)-1 , cbPattern-(2-1)-2 )
                                            } else Gulliver = cbPattern-(2-1);
                                            i = i + Gulliver;
                                            //GlobalI++; // Comment it, it is only for stats.
                                }
                                return(NULL);
            // BMH order 4, needle should be >=8:
            } else { //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )
                                //countSTATIC = cbPattern-2-2;
                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                i=0;
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
                                // In line below we "hash" 4bytes to 2bytes i.e. 16bit table, how to compute TOTAL number of BBs, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox'
and Order=4 we have BBs = 11-4+1=8:
                                //"fast"
                                //"aste"
                                //"stes"
                                //"test"
                                //"est "
```

```
                                        //"st f"
                                        //"t fo"
                                        //" fox"
                                        //for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( *(unsigned short *)(pbPattern+j+0) + *(unsigned short *)(pbPattern+j+2) ) & ( (1<<16)-1 )]=1;
//Possible commenting of next line:
                                        for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( (*(uint32_t *)(pbPattern+j+0)>>16)+(*(uint32_t *)(pbPattern+j+0)&0xFFFF) ) & ( (1<<16)-1 )]=1;
                                        while (i <= cbTarget-cbPattern) {
                                                Gulliver = 1;
                                                if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]&0xFFFF) ) & ( (1<<16)-1 )] != 0 ) {
                                                        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = cbPattern-(2-1)-2-4; else {
                                                                // Order 4 [
                                                                // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                                // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:

                                                                //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                                //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                                //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                                //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                                //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                                //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                                //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                                //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                        count = cbPattern-4+1;
                                                                        while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                                count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                        if ( count <= 0 ) {
                                                                                if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                        }
                                                                        //else {
                                                                        //        if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+count-1]>>16)+(*(uint32_t *)&pbTarget[i+count-1]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = count; // 1 or
bigger, as it should

                                                                        //}
                                                                        // Order 4 ]
                                                        }
                                                } else Gulliver = cbPattern-(2-1)-2; // -2 because we check the 4 rightmost bytes not 2.
                                                i = i + Gulliver;
                                                //GlobalI++; // Comment it, it is only for stats.
                                        }
                                        return(NULL);
                } //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )

                } else { // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() [
                                        //countSTATIC = cbPattern-2-2;
                                        ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                        //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                        i=0;
                                        for (a=0; a < 1<<(HashTableSizeSekireiTchittoGritto-3); a++) {bm_Hasherezade_HASH[a]= 0;} // to-do: 'memset' if not optimized
                                        // cbPattern - Order + 1 i.e. number of BBs for 11 'fastest fox' 11-8+1=4: 'fastest ', 'astest f', 'stest fo', 'test fox'
                                        for (j=0; j < cbPattern-12+1; j++) {
                                                hash32 = (2166136261UL ^ *(uint32_t *)(pbPattern+j+0)) * 709607;
                                                hash32B = (2166136261UL ^ *(uint32_t *)(pbPattern+j+4)) * 709607;
                                                hash32C = (2166136261UL ^ *(uint32_t *)(pbPattern+j+8)) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                bm_Hasherezade_HASH[hash32>>3]= bm_Hasherezade_HASH[hash32>>3] | (1<<(hash32&0x7));
                                        }
                                        while (i <= cbTarget-cbPattern) {
                                                Gulliver = 1; // Assume minimal jump as initial value.
                                                // The goal: to jump when the rightmost 8bytes (Order 8 Horspool) of window do not look like any of Needle prefixes i.e. are not to be found. This maximum jump equals cbPattern-(Order-1) or 11-(8-1)=4 for
'fastest fox' - a small one but for Needle 31 bytes the jump equals 31-(8-1)=24
                                                //GlobalHashSectionExecution++; // Comment it, it is only for stats.
                                                        hash32 = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+0)) * 709607;
                                                        hash32B = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+4)) * 709607;
                                                        hash32C = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+8)) * 709607;
                                                        hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                        hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                        hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
```

```
                                                        if ( (bm_Hasherezade_HASH[hash32>>3] & (1<<(hash32&0x7))) ==0 )     Gulliver = cbPattern-(12-1);
                                                        else {
                                                        //if ( Gulliver == 1 ) { // Means the Building-Block order 8/12 is found somewhere i.e. NO MAXIMUM SKIP
                                                                                // Order 4 [
                                                                        // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                                        // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:

                                                                        //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                                        //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                                        //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                                        //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                                        //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                                        //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                                        //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                                        //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                                count = cbPattern-4+1;
                                                                                while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                                        count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                                if ( count <= 0 ) {
                                                                                        if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                                }
                                                                                // Order 4 ]
                                                        }
                                                i = i + Gulliver;
                                                //GlobalI++; // Comment it, it is only for stats.
                                        } // while (i <= cbTarget-cbPattern)
                                        return(NULL);
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() ]
                } // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
                        } //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
                } //if ( cbPattern<4 )
}
char * Railgun_Sekireigan_Wolfram_f (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
{
                char * pbTargetMax = pbTarget + cbTarget;
                register uint32_t ulHashPattern;
                register uint32_t ulHashTarget;
                signed long count;
                //signed long countSTATIC;

                unsigned char SINGLET;
                uint32_t Quadruplet2nd;
                uint32_t Quadruplet3rd;
                uint32_t Quadruplet4th;

                uint32_t AdvanceHopperGrass;

                uint32_t a, i, j;
//Global is next line already:
                //unsigned char bm_Horspool_Order2[256*256]; // BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
                uint32_t Gulliver; // or unsigned char or unsigned short

                //unsigned char bm_Hasherezade_HASH[1<<(HashTableSizeSekireiTchittoGritto-3)];
                uint32_t hash32;
                uint32_t hash32B;
                uint32_t hash32C;

                if (cbPattern > cbTarget) return(NULL);

                if ( cbPattern<4 ) {

                pbTarget = pbTarget+cbPattern;
                        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
                        if ( cbPattern==3 ) {
                                        for ( ;; ) {
                                                if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
                                                        if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
                                                }
                                                if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) {
```

```
                                        pbTarget++;
                                        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
                                }
                                pbTarget++;
                                if (pbTarget > pbTargetMax) return(NULL);
                        }
                } else {
                }
                for ( ;; ) {
                        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) ) return((pbTarget-2));
                        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
                        pbTarget++;
                        if (pbTarget > pbTargetMax) return(NULL);
                }

        } else {
                if (cbTarget<HaystackThresholdSekireiTchittoGritto) { // This value is arbitrary (don't know how exactly), it ensures (at least must) better performance than 'Boyer_Moore_Horspool'.

                pbTarget = pbTarget+cbPattern;
                ulHashPattern = *(uint32_t *)(pbPattern);
                SINGLET = ulHashPattern & 0xFF;
                Quadruplet2nd = SINGLET<<8;
                Quadruplet3rd = SINGLET<<16;
                Quadruplet4th = SINGLET<<24;
                for ( ;; ) {
                        AdvanceHopperGrass = 0;
                        ulHashTarget = *(uint32_t *)(pbTarget-cbPattern);
                        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
                                count = cbPattern-1;
                                while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                                        if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
                                        count--;
                                }
                                if ( count == 0) return((pbTarget-cbPattern));
                        } else { // The goal here: to avoid memory accesses by stressing the registers.
                                if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                        AdvanceHopperGrass++;
                                        if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                AdvanceHopperGrass++;
                                                if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                        }
                                }
                        }
                        AdvanceHopperGrass++;
                        pbTarget = pbTarget + AdvanceHopperGrass;
                        if (pbTarget > pbTargetMax) return(NULL);
                }

                } else { //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
        if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto ) {

        // BMH order 2:
        if ( cbPattern<=NeedleThreshold2vs4TchittoGritto ) {
                        //countSTATIC = cbPattern-2-2;
                        ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                        //ulHashTarget = *(uint32_t *)(pbPattern+cbPattern-4); // Last four bytes
                        i=0;
                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
                        //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
//Possible commenting of next line:
                        for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=1;
                        while (i <= cbTarget-cbPattern) {
                                Gulliver = 1; // 'Gulliver' is the skip
        // Few thoughts regarding an excellent Skip Performance etude:
        // Something "still" considered crazy: using BITwise order 3, not pseudo order 3, though!
        // 2^24 = 16MB BYTEwise or 2^(24-3) = 2MB BITwise, when searching big haystacks e.g. Wikipedia 42GB with Kazahana then this 2MB lookup table seems not so atrocious.
        // The code is like:
```

```
	// ulHashTarget = *(uint32_t *)&pbTarget[i+cbPattern-4]; // One memory access instead of 2
	// if ( bm_Horspool_Order3[ulHashTarget>>8] == 0 ) Gulliver = cbPattern-(3-1); else
	// if ( bm_Horspool_Order3[ulHashTarget&0xFFFFFF] == 0 ) Gulliver = cbPattern-(3-1)-1; else
	// {
	// ...
	// }
				if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]] != 0 ) {
					if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-2]] + bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1-1]] != 2 ) Gulliver = cbPattern-(2-1)-2; else {
						if ( *(uint32_t *)&pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for remainder) when going under 0 in loop below:
						// Order 4 [
					// Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
					// Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
					//0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
					//1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
					//2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
					//3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
					//4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
					//5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
					//6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
					//7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
						count = cbPattern-4+1;
						//count = count-4; // Double-beauty here of already being checked 'ulHashTarget' and not polluting/repeating the final lookup below.
						while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
							count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
						if ( count <= 0 ) {
							return(pbTarget+i);
						}
						//if ( count <= 0 ) {
						//	if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
						//}
						//else {
						//	if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+count-1]] == 0 ) Gulliver = count; // 1 or bigger, as it should
						//}
						// Order 4 ]
						}
					} // Means AT LEAST one of the BBs is 0, enforce lower skip: MIN( cbPattern-(2-1)-1 , cbPattern-(2-1)-2 )
				} else Gulliver = cbPattern-(2-1);
				i = i + Gulliver;
				//GlobalI++; // Comment it, it is only for stats.
			}
			return(NULL);
	// BMH order 4, needle should be >=8:
	} else { //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )
			//countSTATIC = cbPattern-2-2;
			ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
			//ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
			i=0;
			//for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
			//for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed
//Global is next line already:
			//for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
			// In line below we "hash" 4bytes to 2bytes i.e. 16bit table, how to compute TOTAL number of BBs, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox'
and Order=4 we have BBs = 11-4+1=8:
			//"fast"
			//"aste"
			//"stes"
			//"test"
			//"est "
			//"st f"
			//"t fo"
			//" fox"
			//for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( *(unsigned short *)(pbPattern+j+0) + *(unsigned short *)(pbPattern+j+2) ) & ( (1<<16)-1 )]=1;
//Possible commenting of next line:
			for (j=0; j < cbPattern-4+1; j++) bm_Horspool_Order2[( (*(uint32_t *)(pbPattern+j+0)>>16)+(*(uint32_t *)(pbPattern+j+0)&0xFFFF) ) & ( (1<<16)-1 )]=1;
			while (i <= cbTarget-cbPattern) {
				Gulliver = 1;
				if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2]&0xFFFF) ) & ( (1<<16)-1 )] != 0 ) {
					if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>16)+(*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = cbPattern-(2-1)-2-4; else {
							// Order 4 [
```

```
                                                                        // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                                        // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:

                                                                        //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                                        //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                                        //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                                        //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
                                                                        //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                                        //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                                        //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                                        //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                                    count = cbPattern-4+1;
                                                                                    while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                                                count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                                    if ( count <= 0 ) {
                                                                                                if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                                    }
                                                                                    //else {
                                                                                    //          if ( bm_Horspool_Order2[( (*(uint32_t *)&pbTarget[i+count-1]>>16)+(*(uint32_t *)&pbTarget[i+count-1]&0xFFFF) ) & ( (1<<16)-1 )] == 0 ) Gulliver = count; // 1 or
bigger, as it should

                                                                                    //}
                                                                                    // Order 4 ]
                                                                        }
                                                            } else Gulliver = cbPattern-(2-1)-2; // -2 because we check the 4 rightmost bytes not 2.
                                                            i = i + Gulliver;
                                                            //GlobalI++; // Comment it, it is only for stats.
                                                }
                                                return(NULL);
                                    } //if ( cbPattern<=NeedleThreshold2vs4TchittoGritto )

                                    } else { // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() [
                                                //countSTATIC = cbPattern-2-2;
                                                ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
                                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes
                                                i=0;
                                                for (a=0; a < 1<<(HashTableSizeSekireiTchittoGritto-3); a++) {bm_Hasherezade_HASH[a]= 0;} // to-do: 'memset' if not optimized
                                                // cbPattern - Order + 1 i.e. number of BBs for 11 'fastest fox' 11-8+1=4: 'fastest ', 'astest f', 'stest fo', 'test fox'
                                                for (j=0; j < cbPattern-12+1; j++) {
                                                            hash32 = (2166136261UL ^ *(uint32_t *)(pbPattern+j+0)) * 709607;
                                                            hash32B = (2166136261UL ^ *(uint32_t *)(pbPattern+j+4)) * 709607;
                                                            hash32C = (2166136261UL ^ *(uint32_t *)(pbPattern+j+8)) * 709607;
                                                            hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                            hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                            hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                            bm_Hasherezade_HASH[hash32>>3]= bm_Hasherezade_HASH[hash32>>3] | (1<<(hash32&0x7));
                                                }
                                                while (i <= cbTarget-cbPattern) {
                                                            Gulliver = 1; // Assume minimal jump as initial value.
                                                            // The goal: to jump when the rightmost 8bytes (Order 8 Horspool) of window do not look like any of Needle prefixes i.e. are not to be found. This maximum jump equals cbPattern-(Order-1) or 11-(8-1)=4 for
'fastest fox' - a small one but for Needle 31 bytes the jump equals 31-(8-1)=24
                                                            //GlobalHashSectionExecution++; // Comment it, it is only for stats.
                                                                        hash32 = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+0)) * 709607;
                                                                        hash32B = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+4)) * 709607;
                                                                        hash32C = (2166136261UL ^ *(uint32_t *)(pbTarget+i+cbPattern-12+8)) * 709607;
                                                                        hash32 = (hash32 ^ _rotl_KAZE(hash32C,5) ) * 709607;
                                                                        hash32 = (hash32 ^ _rotl_KAZE(hash32B,5) ) * 709607;
                                                                        hash32 = ( hash32 ^ (hash32 >> 16) ) & ( (1<<(HashTableSizeSekireiTchittoGritto))-1 );
                                                                        if ( (bm_Hasherezade_HASH[hash32>>3] & (1<<(hash32&0x7))) ==0 )     Gulliver = cbPattern-(12-1);
                                                                        else {
                                                                        //if ( Gulliver == 1 ) { // Means the Building-Block order 8/12 is found somewhere i.e. NO MAXIMUM SKIP
                                                                                    // Order 4 [
                                                                        // Let's try something "outrageous" like comparing with[out] overlap BBs 4bytes long instead of 1 byte back-to-back:
                                                                        // Inhere we are using order 4, 'cbPattern - Order + 1' is the number of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-
4+1=8:

                                                                        //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
                                                                        //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
                                                                        //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
                                                                        //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
```

```
                                                                        //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
                                                                        //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
                                                                        //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
                                                                        //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
                                                                                  count = cbPattern-4+1;
                                                                                  while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)(&pbTarget[i]+(count-1)) )
                                                                                            count = count-4; // - order, of course order 4 is much more SWEET&CHEAP - less loops
                                                                                  if ( count <= 0 ) {
                                                                                            if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) return(pbTarget+i);
                                                                                  }
                                                                                  // Order 4 ]
                                                                        }
                                                            i = i + Gulliver;
                                                            //GlobalI++; // Comment it, it is only for stats.
                                                  } // while (i <= cbTarget-cbPattern)
                                                  return(NULL);
// MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() MEMMEM() ]
            } // if ( cbPattern<=NeedleThresholdBIGSekireiTchittoGritto )
                        } //if (cbTarget<HaystackThresholdSekireiTchittoGritto)
            } //if ( cbPattern<4 )
}

// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm [
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
char * Railgun_Quadruplet_7_1 (char * pbTarget,
     char * pbPattern,
     unsigned long cbTarget,
     unsigned long cbPattern)
{
    char * pbTargetMax = pbTarget + cbTarget;
    register unsigned long  ulHashPattern;
    unsigned long ulHashTarget;
    //unsigned long count; //r.6+
    signed long count;
    //unsigned long countSTATIC; //r.6+
    signed long countSTATIC;
//  unsigned long countRemainder;

/*
    const unsigned char SINGLET = *(char *)(pbPattern);
    const unsigned long Quadruplet2nd = SINGLET<<8;
    const unsigned long Quadruplet3rd = SINGLET<<16;
    const unsigned long Quadruplet4th = SINGLET<<24;
*/
    unsigned char SINGLET;
    unsigned long Quadruplet2nd;
    unsigned long Quadruplet3rd;
    unsigned long Quadruplet4th;

    unsigned long  AdvanceHopperGrass;

    long i; //BMH needed
//Below array is already global:
    int a, j;
    //int a, j, bm_bc[ASIZE]; //BMH needed
    unsigned char ch; //BMH needed
    unsigned long chchchch; //BMH needed
//    unsigned char lastch, firstch; //BMH needed

    if (cbPattern > cbTarget)
        return(NULL);

// Doesn't work when cbPattern = 1
// The next IF-fragment works very well with cbPattern>1, OBVIOUSLY IT MUST BE UNROLLED(but crippled with less functionality) SINCE either cbPattern=2 or cbPattern=3!
if ( cbPattern<4) { // This IF makes me unhappy: it slows down from 390KB/clock to 367KB/clock for 'fast' pattern. This fragment(for 2..3 pattern lengths) is needed because I need a function different than strchr but sticking to strstr i.e. lengths
above 1 are to be handled.
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
//        countSTATIC = cbPattern-2;
```

```
if ( cbPattern==3) {
    for ( ;; )
    {
        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
         if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else {
}
    for ( ;; )
    {
        // The line below gives for 'cbPattern'>=1:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
        // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
/*
        if ( (ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1)) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned int)cbPattern) )
            return((long)(pbTarget-cbPattern));
*/

        // The fragment below gives for 'cbPattern'>=2:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
        // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock

/*
//For 2 and 3 [
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
//         count = countSTATIC;
         count = cbPattern-2;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
         while ( count && *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) { // Crippling i.e. only 2 and 3 chars are allowed!
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-cbPattern+1) ) pbTarget++;
//For 2 and 3 ]
*/


        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
            return((pbTarget-2));
        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;


        // The fragment below gives for 'cbPattern'>=2:
           // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
           // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
/*
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
         count = countSTATIC>>2;
         countRemainder = countSTATIC % 4;

         while ( count && *(unsigned long *)(pbPattern+1+((count-1)<<2)) == *(unsigned long *)(pbTarget-cbPattern+1+((count-1)<<2)) ) {
               count--;
         }
            //if (count == 0) {  // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skillessness'; 312KB/clock when 1+1x4+2+1 bytes pattern: 'underdog'), otherwise 368KB/clock.
         while ( countRemainder && *(char *)(pbPattern+1+(countSTATIC-countRemainder)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-countRemainder)) ) {
               countRemainder--;
         }
         //if ( countRemainder == 0) return((long)(pbTarget-cbPattern));
         if ( count+countRemainder == 0) return((long)(pbTarget-cbPattern));
         //}
        }
*/
```

```
        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if ( cbPattern<4)
if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.
{
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = *(unsigned long *)(pbPattern);
//        countSTATIC = cbPattern-1;

    //SINGLET = *(char *)(pbPattern);
    SINGLET = ulHashPattern & 0xFF;
    Quadruplet2nd = SINGLET<<8;
    Quadruplet3rd = SINGLET<<16;
    Quadruplet4th = SINGLET<<24;

    for ( ;; )
    {
            AdvanceHopperGrass = 0;
            ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
//         count = countSTATIC;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
//                if ( countSTATIC==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) AdvanceHopperGrass++;
//               count--;
//         }
         count = cbPattern-1;
         while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                 if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        } else { // The goal here: to avoid memory accesses by stressing the registers.
    if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
         AdvanceHopperGrass++;
         if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
              AdvanceHopperGrass++;
              if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
         }
    }
            }

            AdvanceHopperGrass++;

            pbTarget = pbTarget + AdvanceHopperGrass;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if (cbTarget<961)
        //countSTATIC = cbPattern-2; //r.6+
        //countSTATIC = cbPattern-2-3;
        //countSTATIC = cbPattern-2-2; // r.6+++ I suppose that the awful degradation comes from 2bytes more (from either 'if (countSTATIC<0) countSTATIC=0;' or 'count >0' fixes) which make the function unfittable in code cache lines?!
        //countSTATIC = cbPattern-2-3; // r.7- At last no recompared bytes in-between chars
        countSTATIC = cbPattern-2-2; // r.7
        ulHashPattern = *(unsigned long *)(pbPattern);

        //chPTR=(unsigned char *)&chchchch+3;
// Next line fixes the BUG from r.6++: but with awful speed degradation! So the bug is fixed in the definitions by setting 'countSTATIC = cbPattern-2-2;', bug appears only for patterns with lengths of 4, The setback is one unnecessary comparison for
5bytes patterns, stupidly such setback exists (from before) for 4bytes as well.
//if (countSTATIC<0) countSTATIC=0;
    /* Preprocessing */
//Below 2 lines are global already:
    //for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
    //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;

    /* Searching */
```

```
    //lastch=pbPattern[cbPattern-1];
    //firstch=pbPattern[0];
    i=0;
    while (i <= cbTarget-cbPattern) {
       //ch=pbTarget[i+cbPattern-1];
       //ch=pbTarget[i];
          //if ( pbTarget[i] == pbPattern[0] && *(unsigned long *)&pbTarget[i+cbPattern-1-3] == ulHashPattern) // No problema here since we have 4[+] long pattern here. Overlapping (1 byte recompared) when length=4, grmbl.
          if ( *(unsigned long *)&pbTarget[i] == ulHashPattern ) // The lesson I learned from r.7- now applied in r.7: instead of extracting 'ch' having higher address now the lower address is extracted first in order (hopefully, the test confirms it)
the next 32bytes (including 'ch') to be cached i.e. to comparison part is faster.
             {
         count = countSTATIC;
         while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e. no need of comparing in-between chars.
               count--;
         }
         if ( count == 0) return(pbTarget+i);
                }
       i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
//GlobalI++;
    }
    return(NULL);
} //if (cbTarget<961)
} //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm ]

// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm [
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
char * Railgun_Quadruplet_7_2 (char * pbTarget,
     char * pbPattern,
     unsigned long cbTarget,
     unsigned long cbPattern)
{
    char * pbTargetMax = pbTarget + cbTarget;
    register unsigned long  ulHashPattern;
    unsigned long ulHashTarget;
    //unsigned long count; //r.6+
    signed long count;
    //unsigned long countSTATIC; //r.6+
    signed long countSTATIC;
//  unsigned long countRemainder;

/*
    const unsigned char SINGLET = *(char *)(pbPattern);
    const unsigned long Quadruplet2nd = SINGLET<<8;
    const unsigned long Quadruplet3rd = SINGLET<<16;
    const unsigned long Quadruplet4th = SINGLET<<24;
*/
    unsigned char SINGLET;
    unsigned long Quadruplet2nd;
    unsigned long Quadruplet3rd;
    unsigned long Quadruplet4th;

    unsigned long  AdvanceHopperGrass;

    long i; //BMH needed
//Below array is already global:
    int a, j;
    //int a, j, bm_bc[ASIZE]; //BMH needed
    unsigned char ch; //BMH needed
    unsigned long chchchch; //BMH needed
//    unsigned char lastch, firstch; //BMH needed

    if (cbPattern > cbTarget)
        return(NULL);

// Doesn't work when cbPattern = 1
// The next IF-fragment works very well with cbPattern>1, OBVIOUSLY IT MUST BE UNROLLED(but crippled with less functionality) SINCE either cbPattern=2 or cbPattern=3!
if ( cbPattern<4) { // This IF makes me unhappy: it slows down from 390KB/clock to 367KB/clock for 'fast' pattern. This fragment(for 2..3 pattern lengths) is needed because I need a function different than strchr but sticking to strstr i.e. lengths
above 1 are to be handled.
```

```
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
//        countSTATIC = cbPattern-2;

if ( cbPattern==3) {
    for ( ;; )
    {
        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
         if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else {
}
    for ( ;; )
    {
        // The line below gives for 'cbPattern'>=1:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
        // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
/*
        if ( (ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1)) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned int)cbPattern) )
            return((long)(pbTarget-cbPattern));
*/

        // The fragment below gives for 'cbPattern'>=2:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
        // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock

/*
//For 2 and 3 [
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
//         count = countSTATIC;
         count = cbPattern-2;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
         while ( count && *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) { // Crippling i.e. only 2 and 3 chars are allowed!
              count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-cbPattern+1) ) pbTarget++;
//For 2 and 3 ]
*/


        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
            return((pbTarget-2));
        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;


        // The fragment below gives for 'cbPattern'>=2:
           // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
           // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
/*
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
         count = countSTATIC>>2;
         countRemainder = countSTATIC % 4;

         while ( count && *(unsigned long *)(pbPattern+1+((count-1)<<2)) == *(unsigned long *)(pbTarget-cbPattern+1+((count-1)<<2)) ) {
              count--;
         }
            //if (count == 0) {  // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skillessness'; 312KB/clock when 1+1x4+2+1 bytes pattern: 'underdog'), otherwise 368KB/clock.
         while ( countRemainder && *(char *)(pbPattern+1+(countSTATIC-countRemainder)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-countRemainder)) ) {
              countRemainder--;
         }
         //if ( countRemainder == 0) return((long)(pbTarget-cbPattern));
         if ( count+countRemainder == 0) return((long)(pbTarget-cbPattern));
```

```
         //}
        }
*/

        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if ( cbPattern<4)
if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.
{
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = *(unsigned long *)(pbPattern);
//        countSTATIC = cbPattern-1;

    //SINGLET = *(char *)(pbPattern);
    SINGLET = ulHashPattern & 0xFF;
    Quadruplet2nd = SINGLET<<8;
    Quadruplet3rd = SINGLET<<16;
    Quadruplet4th = SINGLET<<24;

    for ( ;; )
    {
            AdvanceHopperGrass = 0;
            ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
//         count = countSTATIC;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
//                if ( countSTATIC==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) AdvanceHopperGrass++;
//               count--;
//         }
         count = cbPattern-1;
         while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                 if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        } else { // The goal here: to avoid memory accesses by stressing the registers.
    if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
         AdvanceHopperGrass++;
         if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
             AdvanceHopperGrass++;
             if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
         }
    }
            }

            AdvanceHopperGrass++;

            pbTarget = pbTarget + AdvanceHopperGrass;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if (cbTarget<961)
        //countSTATIC = cbPattern-2; //r.6+
        //countSTATIC = cbPattern-2-3;
        //countSTATIC = cbPattern-2-2; // r.6+++ I suppose that the awful degradation comes from 2bytes more (from either 'if (countSTATIC<0) countSTATIC=0;' or 'count >0' fixes) which make the function unfittable in code cache lines?!
        //countSTATIC = cbPattern-2-3; // r.7- At last no recompared bytes in-between chars
        countSTATIC = cbPattern-2-2; // r.7
        ulHashPattern = *(unsigned long *)(pbPattern);

        //chPTR=(unsigned char *)&chchchch+3;
// Next line fixes the BUG from r.6++: but with awful speed degradation! So the bug is fixed in the definitions by setting 'countSTATIC = cbPattern-2-2;', bug appears only for patterns with lengths of 4, The setback is one unnecessary comparison for
5bytes patterns, stupidly such setback exists (from before) for 4bytes as well.
//if (countSTATIC<0) countSTATIC=0;
    /* Preprocessing */
//Below 2 lines are global already:
    //for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
```

```
    //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;

    /* Searching */
    //lastch=pbPattern[cbPattern-1];
    //firstch=pbPattern[0];
    i=0;
    while (i <= cbTarget-cbPattern) {
       //ch=pbTarget[i+cbPattern-1];
       //ch=pbTarget[i];
          //if ( pbTarget[i] == pbPattern[0] && *(unsigned long *)&pbTarget[i+cbPattern-1-3] == ulHashPattern) // No problema here since we have 4[+] long pattern here. Overlapping (1 byte recompared) when length=4, grmbl.
          if ( *(unsigned long *)&pbTarget[i] == ulHashPattern ) // The lesson I learned from r.7- now applied in r.7: instead of extracting 'ch' having higher address now the lower address is extracted first in order (hopefully, the test confirms it)
the next 32bytes (including 'ch') to be cached i.e. to comparison part is faster.
             {
         count = countSTATIC;
         while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e. no need of comparing in-between chars.
              count--;
         }
         if ( count == 0) return(pbTarget+i);
                }
       i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
//GlobalI++;
    }
    return(NULL);
} //if (cbTarget<961)
} //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm ]

// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm [
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
char * Railgun_Quadruplet_7_3 (char * pbTarget,
     char * pbPattern,
     unsigned long cbTarget,
     unsigned long cbPattern)
{
    char * pbTargetMax = pbTarget + cbTarget;
    register unsigned long  ulHashPattern;
    unsigned long ulHashTarget;
    //unsigned long count; //r.6+
    signed long count;
    //unsigned long countSTATIC; //r.6+
    signed long countSTATIC;
//  unsigned long countRemainder;

/*
    const unsigned char SINGLET = *(char *)(pbPattern);
    const unsigned long Quadruplet2nd = SINGLET<<8;
    const unsigned long Quadruplet3rd = SINGLET<<16;
    const unsigned long Quadruplet4th = SINGLET<<24;
*/
    unsigned char SINGLET;
    unsigned long Quadruplet2nd;
    unsigned long Quadruplet3rd;
    unsigned long Quadruplet4th;

    unsigned long  AdvanceHopperGrass;

    long i; //BMH needed
//Below array is already global:
    int a, j;
    //int a, j, bm_bc[ASIZE]; //BMH needed
    unsigned char ch; //BMH needed
    unsigned long chchchch; //BMH needed
//    unsigned char lastch, firstch; //BMH needed

    if (cbPattern > cbTarget)
        return(NULL);

// Doesn't work when cbPattern = 1
```

```
// The next IF-fragment works very well with cbPattern>1, OBVIOUSLY IT MUST BE UNROLLED(but crippled with less functionality) SINCE either cbPattern=2 or cbPattern=3!
if ( cbPattern<4) { // This IF makes me unhappy: it slows down from 390KB/clock to 367KB/clock for 'fast' pattern. This fragment(for 2..3 pattern lengths) is needed because I need a function different than strchr but sticking to strstr i.e. lengths
above 1 are to be handled.
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
//        countSTATIC = cbPattern-2;

if ( cbPattern==3) {
    for ( ;; )
    {
        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
         if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else {
}
    for ( ;; )
    {
        // The line below gives for 'cbPattern'>=1:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
        // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
/*
        if ( (ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1)) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned int)cbPattern) )
            return((long)(pbTarget-cbPattern));
*/

        // The fragment below gives for 'cbPattern'>=2:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
        // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock

/*
//For 2 and 3 [
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
//         count = countSTATIC;
         count = cbPattern-2;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
         while ( count && *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) { // Crippling i.e. only 2 and 3 chars are allowed!
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-cbPattern+1) ) pbTarget++;
//For 2 and 3 ]
*/


        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
            return((pbTarget-2));
        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;


        // The fragment below gives for 'cbPattern'>=2:
           // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
           // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
/*
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
         count = countSTATIC>>2;
         countRemainder = countSTATIC % 4;

         while ( count && *(unsigned long *)(pbPattern+1+((count-1)<<2)) == *(unsigned long *)(pbTarget-cbPattern+1+((count-1)<<2)) ) {
               count--;
         }
            //if (count == 0) {  // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skillessness'; 312KB/clock when 1+1x4+2+1 bytes pattern: 'underdog'), otherwise 368KB/clock.
         while ( countRemainder && *(char *)(pbPattern+1+(countSTATIC-countRemainder)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-countRemainder)) ) {
               countRemainder--;
```

```
         }
         //if ( countRemainder == 0) return((long)(pbTarget-cbPattern));
         if ( count+countRemainder == 0) return((long)(pbTarget-cbPattern));
         //}
        }
*/

        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if ( cbPattern<4)
if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.
{
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = *(unsigned long *)(pbPattern);
//        countSTATIC = cbPattern-1;

    //SINGLET = *(char *)(pbPattern);
    SINGLET = ulHashPattern & 0xFF;
    Quadruplet2nd = SINGLET<<8;
    Quadruplet3rd = SINGLET<<16;
    Quadruplet4th = SINGLET<<24;

    for ( ;; )
    {
            AdvanceHopperGrass = 0;
            ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
//         count = countSTATIC;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
//                if ( countSTATIC==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) AdvanceHopperGrass++;
//               count--;
//         }
         count = cbPattern-1;
         while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                  if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
              count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        } else { // The goal here: to avoid memory accesses by stressing the registers.
    if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
         AdvanceHopperGrass++;
         if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
              AdvanceHopperGrass++;
              if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
         }
    }
            }

            AdvanceHopperGrass++;

            pbTarget = pbTarget + AdvanceHopperGrass;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if (cbTarget<961)
        //countSTATIC = cbPattern-2; //r.6+
        //countSTATIC = cbPattern-2-3;
        //countSTATIC = cbPattern-2-2; // r.6+++ I suppose that the awful degradation comes from 2bytes more (from either 'if (countSTATIC<0) countSTATIC=0;' or 'count >0' fixes) which make the function unfittable in code cache lines?!
        //countSTATIC = cbPattern-2-3; // r.7- At last no recompared bytes in-between chars
        countSTATIC = cbPattern-2-2; // r.7
        ulHashPattern = *(unsigned long *)(pbPattern);

        //chPTR=(unsigned char *)&chchchch+3;
// Next line fixes the BUG from r.6++: but with awful speed degradation! So the bug is fixed in the definitions by setting 'countSTATIC = cbPattern-2-2;', bug appears only for patterns with lengths of 4, The setback is one unnecessary comparison for
5bytes patterns, stupidly such setback exists (from before) for 4bytes as well.
//if (countSTATIC<0) countSTATIC=0;
```

```
    /* Preprocessing */
//Below 2 lines are global already:
    //for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
    //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;

    /* Searching */
    //lastch=pbPattern[cbPattern-1];
    //firstch=pbPattern[0];
    i=0;
    while (i <= cbTarget-cbPattern) {
       //ch=pbTarget[i+cbPattern-1];
       //ch=pbTarget[i];
          //if ( pbTarget[i] == pbPattern[0] && *(unsigned long *)&pbTarget[i+cbPattern-1-3] == ulHashPattern) // No problema here since we have 4[+] long pattern here. Overlapping (1 byte recompared) when length=4, grmbl.
          if ( *(unsigned long *)&pbTarget[i] == ulHashPattern ) // The lesson I learned from r.7- now applied in r.7: instead of extracting 'ch' having higher address now the lower address is extracted first in order (hopefully, the test confirms it)
the next 32bytes (including 'ch') to be cached i.e. to comparison part is faster.
             {
         count = countSTATIC;
         while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e. no need of comparing in-between chars.
               count--;
         }
         if ( count == 0) return(pbTarget+i);
                 }
       i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
//GlobalI++;
    }
    return(NULL);
} //if (cbTarget<961)
} //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm ]

// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm [
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
char * Railgun_Quadruplet_7_4 (char * pbTarget,
     char * pbPattern,
     unsigned long cbTarget,
     unsigned long cbPattern)
{
    char * pbTargetMax = pbTarget + cbTarget;
    register unsigned long  ulHashPattern;
    unsigned long ulHashTarget;
    //unsigned long count; //r.6+
    signed long count;
    //unsigned long countSTATIC; //r.6+
    signed long countSTATIC;
//  unsigned long countRemainder;

/*
    const unsigned char SINGLET = *(char *)(pbPattern);
    const unsigned long Quadruplet2nd = SINGLET<<8;
    const unsigned long Quadruplet3rd = SINGLET<<16;
    const unsigned long Quadruplet4th = SINGLET<<24;
*/
    unsigned char SINGLET;
    unsigned long Quadruplet2nd;
    unsigned long Quadruplet3rd;
    unsigned long Quadruplet4th;

    unsigned long  AdvanceHopperGrass;

    long i; //BMH needed
//Below array is already global:
    int a, j;
    //int a, j, bm_bc[ASIZE]; //BMH needed
    unsigned char ch; //BMH needed
    unsigned long chchchch; //BMH needed
//    unsigned char lastch, firstch; //BMH needed

    if (cbPattern > cbTarget)
```

```
        return(NULL);

// Doesn't work when cbPattern = 1
// The next IF-fragment works very well with cbPattern>1, OBVIOUSLY IT MUST BE UNROLLED(but crippled with less functionality) SINCE either cbPattern=2 or cbPattern=3!
if ( cbPattern<4) { // This IF makes me unhappy: it slows down from 390KB/clock to 367KB/clock for 'fast' pattern. This fragment(for 2..3 pattern lengths) is needed because I need a function different than strchr but sticking to strstr i.e. lengths
above 1 are to be handled.
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
//        countSTATIC = cbPattern-2;

if ( cbPattern==3) {
    for ( ;; )
    {
        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
         if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else {
}
    for ( ;; )
    {
        // The line below gives for 'cbPattern'>=1:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
        // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
/*
        if ( (ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1)) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned int)cbPattern) )
            return((long)(pbTarget-cbPattern));
*/

        // The fragment below gives for 'cbPattern'>=2:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
        // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock

/*
//For 2 and 3 [
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
//        count = countSTATIC;
         count = cbPattern-2;
//        while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
         while ( count && *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) { // Crippling i.e. only 2 and 3 chars are allowed!
              count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-cbPattern+1) ) pbTarget++;
//For 2 and 3 ]
*/


        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
            return((pbTarget-2));
        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;


        // The fragment below gives for 'cbPattern'>=2:
           // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
           // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
/*
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
         count = countSTATIC>>2;
         countRemainder = countSTATIC % 4;

         while ( count && *(unsigned long *)(pbPattern+1+((count-1)<<2)) == *(unsigned long *)(pbTarget-cbPattern+1+((count-1)<<2)) ) {
              count--;
         }
```

```
            //if (count == 0) {  // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skillessness'; 312KB/clock when 1+1x4+2+1 bytes pattern: 'underdog'), otherwise 368KB/clock.
         while ( countRemainder && *(char *)(pbPattern+1+(countSTATIC-countRemainder)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-countRemainder)) ) {
               countRemainder--;
         }
         //if ( countRemainder == 0) return((long)(pbTarget-cbPattern));
         if ( count+countRemainder == 0) return((long)(pbTarget-cbPattern));
         //}
        }
*/

        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if ( cbPattern<4)
if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.
{
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = *(unsigned long *)(pbPattern);
//        countSTATIC = cbPattern-1;

    //SINGLET = *(char *)(pbPattern);
    SINGLET = ulHashPattern & 0xFF;
    Quadruplet2nd = SINGLET<<8;
    Quadruplet3rd = SINGLET<<16;
    Quadruplet4th = SINGLET<<24;

    for ( ;; )
    {
            AdvanceHopperGrass = 0;
            ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
//         count = countSTATIC;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
//                if ( countSTATIC==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) AdvanceHopperGrass++;
//               count--;
//         }
         count = cbPattern-1;
         while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                 if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        } else { // The goal here: to avoid memory accesses by stressing the registers.
    if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
         AdvanceHopperGrass++;
         if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
              AdvanceHopperGrass++;
              if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
         }
    }
            }

            AdvanceHopperGrass++;

            pbTarget = pbTarget + AdvanceHopperGrass;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if (cbTarget<961)
        //countSTATIC = cbPattern-2; //r.6+
        //countSTATIC = cbPattern-2-3;
        //countSTATIC = cbPattern-2-2; // r.6+++ I suppose that the awful degradation comes from 2bytes more (from either 'if (countSTATIC<0) countSTATIC=0;' or 'count >0' fixes) which make the function unfittable in code cache lines?!
        //countSTATIC = cbPattern-2-3; // r.7- At last no recompared bytes in-between chars
        countSTATIC = cbPattern-2-2; // r.7
        ulHashPattern = *(unsigned long *)(pbPattern);

        //chPTR=(unsigned char *)&chchchch+3;
```

```
// Next line fixes the BUG from r.6++: but with awful speed degradation! So the bug is fixed in the definitions by setting 'countSTATIC = cbPattern-2-2;', bug appears only for patterns with lengths of 4, The setback is one unnecessary comparison for
5bytes patterns, stupidly such setback exists (from before) for 4bytes as well.
//if (countSTATIC<0) countSTATIC=0;
    /* Preprocessing */
//Below 2 lines are global already:
    //for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
    //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;

    /* Searching */
    //lastch=pbPattern[cbPattern-1];
    //firstch=pbPattern[0];
    i=0;
    while (i <= cbTarget-cbPattern) {
       //ch=pbTarget[i+cbPattern-1];
       //ch=pbTarget[i];
          //if ( pbTarget[i] == pbPattern[0] && *(unsigned long *)&pbTarget[i+cbPattern-1-3] == ulHashPattern) // No problema here since we have 4[+] long pattern here. Overlapping (1 byte recompared) when length=4, grmbl.
          if ( *(unsigned long *)&pbTarget[i] == ulHashPattern ) // The lesson I learned from r.7- now applied in r.7: instead of extracting 'ch' having higher address now the lower address is extracted first in order (hopefully, the test confirms it)
the next 32bytes (including 'ch') to be cached i.e. to comparison part is faster.
             {
         count = countSTATIC;
         while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e. no need of comparing in-between chars.
               count--;
         }
         if ( count == 0) return(pbTarget+i);
                }
       i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
//GlobalI++;
    }
    return(NULL);
} //if (cbTarget<961)
} //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm ]

// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm [
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
char * Railgun_Quadruplet_7_5 (char * pbTarget,
     char * pbPattern,
     unsigned long cbTarget,
     unsigned long cbPattern)
{
    char * pbTargetMax = pbTarget + cbTarget;
    register unsigned long  ulHashPattern;
    unsigned long ulHashTarget;
    //unsigned long count; //r.6+
    signed long count;
    //unsigned long countSTATIC; //r.6+
    signed long countSTATIC;
//  unsigned long countRemainder;

/*
    const unsigned char SINGLET = *(char *)(pbPattern);
    const unsigned long Quadruplet2nd = SINGLET<<8;
    const unsigned long Quadruplet3rd = SINGLET<<16;
    const unsigned long Quadruplet4th = SINGLET<<24;
*/
    unsigned char SINGLET;
    unsigned long Quadruplet2nd;
    unsigned long Quadruplet3rd;
    unsigned long Quadruplet4th;

    unsigned long  AdvanceHopperGrass;

    long i; //BMH needed
//Below array is already global:
    int a, j;
    //int a, j, bm_bc[ASIZE]; //BMH needed
    unsigned char ch; //BMH needed
    unsigned long chchchch; //BMH needed
```

```
//    unsigned char lastch, firstch; //BMH needed

    if (cbPattern > cbTarget)
        return(NULL);

// Doesn't work when cbPattern = 1
// The next IF-fragment works very well with cbPattern>1, OBVIOUSLY IT MUST BE UNROLLED(but crippled with less functionality) SINCE either cbPattern=2 or cbPattern=3!
if ( cbPattern<4) { // This IF makes me unhappy: it slows down from 390KB/clock to 367KB/clock for 'fast' pattern. This fragment(for 2..3 pattern lengths) is needed because I need a function different than strchr but sticking to strstr i.e. lengths
above 1 are to be handled.
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
//        countSTATIC = cbPattern-2;

if ( cbPattern==3) {
    for ( ;; )
    {
        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
         if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else {
}
    for ( ;; )
    {
        // The line below gives for 'cbPattern'>=1:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
        // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
/*
        if ( (ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1)) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned int)cbPattern) )
            return((long)(pbTarget-cbPattern));
*/

        // The fragment below gives for 'cbPattern'>=2:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
        // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock

/*
//For 2 and 3 [
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
//         count = countSTATIC;
         count = cbPattern-2;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
         while ( count && *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) { // Crippling i.e. only 2 and 3 chars are allowed!
             count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-cbPattern+1) ) pbTarget++;
//For 2 and 3 ]
*/


        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
            return((pbTarget-2));
        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;


        // The fragment below gives for 'cbPattern'>=2:
           // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
           // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
/*
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
         count = countSTATIC>>2;
         countRemainder = countSTATIC % 4;
```

```
         while ( count && *(unsigned long *)(pbPattern+1+((count-1)<<2)) == *(unsigned long *)(pbTarget-cbPattern+1+((count-1)<<2)) ) {
               count--;
         }
            //if (count == 0) {  // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skillessness'; 312KB/clock when 1+1x4+2+1 bytes pattern: 'underdog'), otherwise 368KB/clock.
         while ( countRemainder && *(char *)(pbPattern+1+(countSTATIC-countRemainder)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-countRemainder)) ) {
               countRemainder--;
         }
         //if ( countRemainder == 0) return((long)(pbTarget-cbPattern));
         if ( count+countRemainder == 0) return((long)(pbTarget-cbPattern));
         //}
        }
*/

        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if ( cbPattern<4)
if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.
{
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = *(unsigned long *)(pbPattern);
//        countSTATIC = cbPattern-1;

    //SINGLET = *(char *)(pbPattern);
    SINGLET = ulHashPattern & 0xFF;
    Quadruplet2nd = SINGLET<<8;
    Quadruplet3rd = SINGLET<<16;
    Quadruplet4th = SINGLET<<24;

    for ( ;; )
    {
            AdvanceHopperGrass = 0;
            ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
//         count = countSTATIC;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
//                if ( countSTATIC==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) AdvanceHopperGrass++;
//               count--;
//         }
         count = cbPattern-1;
         while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                 if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        } else { // The goal here: to avoid memory accesses by stressing the registers.
    if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
         AdvanceHopperGrass++;
         if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
              AdvanceHopperGrass++;
              if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
         }
    }
            }

            AdvanceHopperGrass++;

            pbTarget = pbTarget + AdvanceHopperGrass;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if (cbTarget<961)
        //countSTATIC = cbPattern-2; //r.6+
        //countSTATIC = cbPattern-2-3;
        //countSTATIC = cbPattern-2-2; // r.6+++ I suppose that the awful degradation comes from 2bytes more (from either 'if (countSTATIC<0) countSTATIC=0;' or 'count >0' fixes) which make the function unfittable in code cache lines?!
        //countSTATIC = cbPattern-2-3; // r.7- At last no recompared bytes in-between chars
        countSTATIC = cbPattern-2-2; // r.7
```

```
        ulHashPattern = *(unsigned long *)(pbPattern);

        //chPTR=(unsigned char *)&chchchch+3;
// Next line fixes the BUG from r.6++: but with awful speed degradation! So the bug is fixed in the definitions by setting 'countSTATIC = cbPattern-2-2;', bug appears only for patterns with lengths of 4, The setback is one unnecessary comparison for
5bytes patterns, stupidly such setback exists (from before) for 4bytes as well.
//if (countSTATIC<0) countSTATIC=0;
    /* Preprocessing */
//Below 2 lines are global already:
    //for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
    //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;

    /* Searching */
    //lastch=pbPattern[cbPattern-1];
    //firstch=pbPattern[0];
    i=0;
    while (i <= cbTarget-cbPattern) {
       //ch=pbTarget[i+cbPattern-1];
       //ch=pbTarget[i];
          //if ( pbTarget[i] == pbPattern[0] && *(unsigned long *)&pbTarget[i+cbPattern-1-3] == ulHashPattern) // No problema here since we have 4[+] long pattern here. Overlapping (1 byte recompared) when length=4, grmbl.
          if ( *(unsigned long *)&pbTarget[i] == ulHashPattern ) // The lesson I learned from r.7- now applied in r.7: instead of extracting 'ch' having higher address now the lower address is extracted first in order (hopefully, the test confirms it)
the next 32bytes (including 'ch') to be cached i.e. to comparison part is faster.
             {
         count = countSTATIC;
         while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e. no need of comparing in-between chars.
               count--;
         }
         if ( count == 0) return(pbTarget+i);
                }
       i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
//GlobalI++;
    }
    return(NULL);
} //if (cbTarget<961)
} //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm ]

// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm [
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
char * Railgun_Quadruplet_7_6 (char * pbTarget,
     char * pbPattern,
     unsigned long cbTarget,
     unsigned long cbPattern)
{
    char * pbTargetMax = pbTarget + cbTarget;
    register unsigned long  ulHashPattern;
    unsigned long ulHashTarget;
    //unsigned long count; //r.6+
    signed long count;
    //unsigned long countSTATIC; //r.6+
    signed long countSTATIC;
//  unsigned long countRemainder;

/*
    const unsigned char SINGLET = *(char *)(pbPattern);
    const unsigned long Quadruplet2nd = SINGLET<<8;
    const unsigned long Quadruplet3rd = SINGLET<<16;
    const unsigned long Quadruplet4th = SINGLET<<24;
*/
    unsigned char SINGLET;
    unsigned long Quadruplet2nd;
    unsigned long Quadruplet3rd;
    unsigned long Quadruplet4th;

    unsigned long  AdvanceHopperGrass;

    long i; //BMH needed
//Below array is already global:
    int a, j;
```

```
    //int a, j, bm_bc[ASIZE]; //BMH needed
    unsigned char ch; //BMH needed
    unsigned long chchchch; //BMH needed
//    unsigned char lastch, firstch; //BMH needed

    if (cbPattern > cbTarget)
        return(NULL);

// Doesn't work when cbPattern = 1
// The next IF-fragment works very well with cbPattern>1, OBVIOUSLY IT MUST BE UNROLLED(but crippled with less functionality) SINCE either cbPattern=2 or cbPattern=3!
if ( cbPattern<4) { // This IF makes me unhappy: it slows down from 390KB/clock to 367KB/clock for 'fast' pattern. This fragment(for 2..3 pattern lengths) is needed because I need a function different than strchr but sticking to strstr i.e. lengths
above 1 are to be handled.
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
//        countSTATIC = cbPattern-2;

if ( cbPattern==3) {
    for ( ;; )
    {
        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
         if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else {
}
    for ( ;; )
    {
        // The line below gives for 'cbPattern'>=1:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
        // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
/*
        if ( (ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1)) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned int)cbPattern) )
            return((long)(pbTarget-cbPattern));
*/

        // The fragment below gives for 'cbPattern'>=2:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
        // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock

/*
//For 2 and 3 [
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
//         count = countSTATIC;
         count = cbPattern-2;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
         while ( count && *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) { // Crippling i.e. only 2 and 3 chars are allowed!
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-cbPattern+1) ) pbTarget++;
//For 2 and 3 ]
*/


        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
            return((pbTarget-2));
        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;


        // The fragment below gives for 'cbPattern'>=2:
           // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
           // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
/*
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
```

```
         count = countSTATIC>>2;
         countRemainder = countSTATIC % 4;

         while ( count && *(unsigned long *)(pbPattern+1+((count-1)<<2)) == *(unsigned long *)(pbTarget-cbPattern+1+((count-1)<<2)) ) {
               count--;
         }
            //if (count == 0) {  // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skillessness'; 312KB/clock when 1+1x4+2+1 bytes pattern: 'underdog'), otherwise 368KB/clock.
         while ( countRemainder && *(char *)(pbPattern+1+(countSTATIC-countRemainder)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-countRemainder)) ) {
               countRemainder--;
         }
         //if ( countRemainder == 0) return((long)(pbTarget-cbPattern));
         if ( count+countRemainder == 0) return((long)(pbTarget-cbPattern));
         //}
        }
*/

        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if ( cbPattern<4)
if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.
{
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = *(unsigned long *)(pbPattern);
//        countSTATIC = cbPattern-1;

    //SINGLET = *(char *)(pbPattern);
    SINGLET = ulHashPattern & 0xFF;
    Quadruplet2nd = SINGLET<<8;
    Quadruplet3rd = SINGLET<<16;
    Quadruplet4th = SINGLET<<24;

    for ( ;; )
    {
            AdvanceHopperGrass = 0;
            ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
//         count = countSTATIC;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
//                if ( countSTATIC==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) AdvanceHopperGrass++;
//               count--;
//         }
         count = cbPattern-1;
         while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                 if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        } else { // The goal here: to avoid memory accesses by stressing the registers.
    if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
         AdvanceHopperGrass++;
         if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
              AdvanceHopperGrass++;
              if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
         }
    }
            }

            AdvanceHopperGrass++;

            pbTarget = pbTarget + AdvanceHopperGrass;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if (cbTarget<961)
        //countSTATIC = cbPattern-2; //r.6+
        //countSTATIC = cbPattern-2-3;
```

```
        //countSTATIC = cbPattern-2-2; // r.6+++ I suppose that the awful degradation comes from 2bytes more (from either 'if (countSTATIC<0) countSTATIC=0;' or 'count >0' fixes) which make the function unfittable in code cache lines?!
        //countSTATIC = cbPattern-2-3; // r.7- At last no recompared bytes in-between chars
        countSTATIC = cbPattern-2-2; // r.7
        ulHashPattern = *(unsigned long *)(pbPattern);

        //chPTR=(unsigned char *)&chchchch+3;
// Next line fixes the BUG from r.6++: but with awful speed degradation! So the bug is fixed in the definitions by setting 'countSTATIC = cbPattern-2-2;', bug appears only for patterns with lengths of 4, The setback is one unnecessary comparison for
5bytes patterns, stupidly such setback exists (from before) for 4bytes as well.
//if (countSTATIC<0) countSTATIC=0;
    /* Preprocessing */
//Below 2 lines are global already:
    //for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
    //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;

    /* Searching */
    //lastch=pbPattern[cbPattern-1];
    //firstch=pbPattern[0];
    i=0;
    while (i <= cbTarget-cbPattern) {
       //ch=pbTarget[i+cbPattern-1];
       //ch=pbTarget[i];
          //if ( pbTarget[i] == pbPattern[0] && *(unsigned long *)&pbTarget[i+cbPattern-1-3] == ulHashPattern) // No problema here since we have 4[+] long pattern here. Overlapping (1 byte recompared) when length=4, grmbl.
          if ( *(unsigned long *)&pbTarget[i] == ulHashPattern ) // The lesson I learned from r.7- now applied in r.7: instead of extracting 'ch' having higher address now the lower address is extracted first in order (hopefully, the test confirms it)
the next 32bytes (including 'ch') to be cached i.e. to comparison part is faster.
             {
         count = countSTATIC;
         while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e. no need of comparing in-between chars.
               count--;
         }
         if ( count == 0) return(pbTarget+i);
                }
       i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
//GlobalI++;
    }
    return(NULL);
} //if (cbTarget<961)
} //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm ]

// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm [
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
char * Railgun_Quadruplet_7_7 (char * pbTarget,
     char * pbPattern,
     unsigned long cbTarget,
     unsigned long cbPattern)
{
    char * pbTargetMax = pbTarget + cbTarget;
    register unsigned long  ulHashPattern;
    unsigned long ulHashTarget;
    //unsigned long count; //r.6+
    signed long count;
    //unsigned long countSTATIC; //r.6+
    signed long countSTATIC;
//  unsigned long countRemainder;

/*
    const unsigned char SINGLET = *(char *)(pbPattern);
    const unsigned long Quadruplet2nd = SINGLET<<8;
    const unsigned long Quadruplet3rd = SINGLET<<16;
    const unsigned long Quadruplet4th = SINGLET<<24;
*/
    unsigned char SINGLET;
    unsigned long Quadruplet2nd;
    unsigned long Quadruplet3rd;
    unsigned long Quadruplet4th;

    unsigned long  AdvanceHopperGrass;
```

```
    long i; //BMH needed
//Below array is already global:
    int a, j;
    //int a, j, bm_bc[ASIZE]; //BMH needed
    unsigned char ch; //BMH needed
    unsigned long chchchch; //BMH needed
//    unsigned char lastch, firstch; //BMH needed

    if (cbPattern > cbTarget)
        return(NULL);

// Doesn't work when cbPattern = 1
// The next IF-fragment works very well with cbPattern>1, OBVIOUSLY IT MUST BE UNROLLED(but crippled with less functionality) SINCE either cbPattern=2 or cbPattern=3!
if ( cbPattern<4) { // This IF makes me unhappy: it slows down from 390KB/clock to 367KB/clock for 'fast' pattern. This fragment(for 2..3 pattern lengths) is needed because I need a function different than strchr but sticking to strstr i.e. lengths
above 1 are to be handled.
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
//        countSTATIC = cbPattern-2;

if ( cbPattern==3) {
    for ( ;; )
    {
        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
         if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else {
}
    for ( ;; )
    {
        // The line below gives for 'cbPattern'>=1:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
        // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
/*
        if ( (ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1)) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned int)cbPattern) )
            return((long)(pbTarget-cbPattern));
*/

        // The fragment below gives for 'cbPattern'>=2:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
        // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock

/*
//For 2 and 3 [
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
//         count = countSTATIC;
         count = cbPattern-2;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
         while ( count && *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) { // Crippling i.e. only 2 and 3 chars are allowed!
              count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-cbPattern+1) ) pbTarget++;
//For 2 and 3 ]
*/


        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
            return((pbTarget-2));
        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;


        // The fragment below gives for 'cbPattern'>=2:
            // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
```

```
          // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
/*
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
         count = countSTATIC>>2;
         countRemainder = countSTATIC % 4;

         while ( count && *(unsigned long *)(pbPattern+1+((count-1)<<2)) == *(unsigned long *)(pbTarget-cbPattern+1+((count-1)<<2)) ) {
               count--;
         }
            //if (count == 0) {  // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skillessness'; 312KB/clock when 1+1x4+2+1 bytes pattern: 'underdog'), otherwise 368KB/clock.
         while ( countRemainder && *(char *)(pbPattern+1+(countSTATIC-countRemainder)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-countRemainder)) ) {
               countRemainder--;
         }
         //if ( countRemainder == 0) return((long)(pbTarget-cbPattern));
         if ( count+countRemainder == 0) return((long)(pbTarget-cbPattern));
         //}
        }
*/

        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if ( cbPattern<4)
if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.
{
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = *(unsigned long *)(pbPattern);
//        countSTATIC = cbPattern-1;

    //SINGLET = *(char *)(pbPattern);
    SINGLET = ulHashPattern & 0xFF;
    Quadruplet2nd = SINGLET<<8;
    Quadruplet3rd = SINGLET<<16;
    Quadruplet4th = SINGLET<<24;

    for ( ;; )
    {
            AdvanceHopperGrass = 0;
            ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
//         count = countSTATIC;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
//                if ( countSTATIC==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) AdvanceHopperGrass++;
//               count--;
//         }
         count = cbPattern-1;
         while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                 if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        } else { // The goal here: to avoid memory accesses by stressing the registers.
    if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
         AdvanceHopperGrass++;
         if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
              AdvanceHopperGrass++;
              if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
         }
    }
            }

            AdvanceHopperGrass++;

            pbTarget = pbTarget + AdvanceHopperGrass;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
```

```
} else { //if (cbTarget<961)
        //countSTATIC = cbPattern-2; //r.6+
        //countSTATIC = cbPattern-2-3;
        //countSTATIC = cbPattern-2-2; // r.6+++ I suppose that the awful degradation comes from 2bytes more (from either 'if (countSTATIC<0) countSTATIC=0;' or 'count >0' fixes) which make the function unfittable in code cache lines?!
        //countSTATIC = cbPattern-2-3; // r.7- At last no recompared bytes in-between chars
        countSTATIC = cbPattern-2-2; // r.7
        ulHashPattern = *(unsigned long *)(pbPattern);

        //chPTR=(unsigned char *)&chchchch+3;
// Next line fixes the BUG from r.6++: but with awful speed degradation! So the bug is fixed in the definitions by setting 'countSTATIC = cbPattern-2-2;', bug appears only for patterns with lengths of 4, The setback is one unnecessary comparison for
5bytes patterns, stupidly such setback exists (from before) for 4bytes as well.
//if (countSTATIC<0) countSTATIC=0;
    /* Preprocessing */
//Below 2 lines are global already:
    //for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
    //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;

    /* Searching */
    //lastch=pbPattern[cbPattern-1];
    //firstch=pbPattern[0];
    i=0;
    while (i <= cbTarget-cbPattern) {
       //ch=pbTarget[i+cbPattern-1];
       //ch=pbTarget[i];
          //if ( pbTarget[i] == pbPattern[0] && *(unsigned long *)&pbTarget[i+cbPattern-1-3] == ulHashPattern) // No problema here since we have 4[+] long pattern here. Overlapping (1 byte recompared) when length=4, grmbl.
          if ( *(unsigned long *)&pbTarget[i] == ulHashPattern ) // The lesson I learned from r.7- now applied in r.7: instead of extracting 'ch' having higher address now the lower address is extracted first in order (hopefully, the test confirms it)
the next 32bytes (including 'ch') to be cached i.e. to comparison part is faster.
             {
         count = countSTATIC;
         while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e. no need of comparing in-between chars.
              count--;
         }
         if ( count == 0) return(pbTarget+i);
                 }
       i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
//GlobalI++;
    }
    return(NULL);
} //if (cbTarget<961)
} //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm ]

// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm [
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
char * Railgun_Quadruplet_7_8 (char * pbTarget,
     char * pbPattern,
     unsigned long cbTarget,
     unsigned long cbPattern)
{
    char * pbTargetMax = pbTarget + cbTarget;
    register unsigned long  ulHashPattern;
    unsigned long ulHashTarget;
    //unsigned long count; //r.6+
    signed long count;
    //unsigned long countSTATIC; //r.6+
    signed long countSTATIC;
//  unsigned long countRemainder;

/*
    const unsigned char SINGLET = *(char *)(pbPattern);
    const unsigned long Quadruplet2nd = SINGLET<<8;
    const unsigned long Quadruplet3rd = SINGLET<<16;
    const unsigned long Quadruplet4th = SINGLET<<24;
*/
    unsigned char SINGLET;
    unsigned long Quadruplet2nd;
    unsigned long Quadruplet3rd;
    unsigned long Quadruplet4th;
```

```
    unsigned long  AdvanceHopperGrass;

    long i; //BMH needed
//Below array is already global:
    int a, j;
    //int a, j, bm_bc[ASIZE]; //BMH needed
    unsigned char ch; //BMH needed
    unsigned long chchchch; //BMH needed
//    unsigned char lastch, firstch; //BMH needed

    if (cbPattern > cbTarget)
        return(NULL);

// Doesn't work when cbPattern = 1
// The next IF-fragment works very well with cbPattern>1, OBVIOUSLY IT MUST BE UNROLLED(but crippled with less functionality) SINCE either cbPattern=2 or cbPattern=3!
if ( cbPattern<4) { // This IF makes me unhappy: it slows down from 390KB/clock to 367KB/clock for 'fast' pattern. This fragment(for 2..3 pattern lengths) is needed because I need a function different than strchr but sticking to strstr i.e. lengths
above 1 are to be handled.
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
//        countSTATIC = cbPattern-2;

if ( cbPattern==3) {
    for ( ;; )
    {
        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
         if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else {
}
    for ( ;; )
    {
        // The line below gives for 'cbPattern'>=1:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
        // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
/*
        if ( (ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1)) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned int)cbPattern) )
            return((long)(pbTarget-cbPattern));
*/

        // The fragment below gives for 'cbPattern'>=2:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
        // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock

/*
//For 2 and 3 [
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
//         count = countSTATIC;
         count = cbPattern-2;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
         while ( count && *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) { // Crippling i.e. only 2 and 3 chars are allowed!
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-cbPattern+1) ) pbTarget++;
//For 2 and 3 ]
*/


        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
            return((pbTarget-2));
        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
```

```
        // The fragment below gives for 'cbPattern'>=2:
           // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
           // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
/*
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
         count = countSTATIC>>2;
         countRemainder = countSTATIC % 4;

         while ( count && *(unsigned long *)(pbPattern+1+((count-1)<<2)) == *(unsigned long *)(pbTarget-cbPattern+1+((count-1)<<2)) ) {
               count--;
         }
            //if (count == 0) {  // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skillessness'; 312KB/clock when 1+1x4+2+1 bytes pattern: 'underdog'), otherwise 368KB/clock.
         while ( countRemainder && *(char *)(pbPattern+1+(countSTATIC-countRemainder)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-countRemainder)) ) {
               countRemainder--;
         }
         //if ( countRemainder == 0) return((long)(pbTarget-cbPattern));
         if ( count+countRemainder == 0) return((long)(pbTarget-cbPattern));
         //}
        }
*/

        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if ( cbPattern<4)
if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.
{
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = *(unsigned long *)(pbPattern);
//        countSTATIC = cbPattern-1;

    //SINGLET = *(char *)(pbPattern);
    SINGLET = ulHashPattern & 0xFF;
    Quadruplet2nd = SINGLET<<8;
    Quadruplet3rd = SINGLET<<16;
    Quadruplet4th = SINGLET<<24;

    for ( ;; )
    {
           AdvanceHopperGrass = 0;
           ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
//         count = countSTATIC;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
//                if ( countSTATIC==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) AdvanceHopperGrass++;
//               count--;
//         }
         count = cbPattern-1;
         while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                 if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        } else { // The goal here: to avoid memory accesses by stressing the registers.
    if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
         AdvanceHopperGrass++;
         if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
              AdvanceHopperGrass++;
              if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
         }
    }
           }

           AdvanceHopperGrass++;

           pbTarget = pbTarget + AdvanceHopperGrass;
```

```
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if (cbTarget<961)
        //countSTATIC = cbPattern-2; //r.6+
        //countSTATIC = cbPattern-2-3;
        //countSTATIC = cbPattern-2-2; // r.6+++ I suppose that the awful degradation comes from 2bytes more (from either 'if (countSTATIC<0) countSTATIC=0;' or 'count >0' fixes) which make the function unfittable in code cache lines?!
        //countSTATIC = cbPattern-2-3; // r.7- At last no recompared bytes in-between chars
        countSTATIC = cbPattern-2-2; // r.7
        ulHashPattern = *(unsigned long *)(pbPattern);

        //chPTR=(unsigned char *)&chchchch+3;
// Next line fixes the BUG from r.6++: but with awful speed degradation! So the bug is fixed in the definitions by setting 'countSTATIC = cbPattern-2-2;', bug appears only for patterns with lengths of 4, The setback is one unnecessary comparison for
5bytes patterns, stupidly such setback exists (from before) for 4bytes as well.
//if (countSTATIC<0) countSTATIC=0;
    /* Preprocessing */
//Below 2 lines are global already:
    //for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
    //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;

    /* Searching */
    //lastch=pbPattern[cbPattern-1];
    //firstch=pbPattern[0];
    i=0;
    while (i <= cbTarget-cbPattern) {
       //ch=pbTarget[i+cbPattern-1];
       //ch=pbTarget[i];
          //if ( pbTarget[i] == pbPattern[0] && *(unsigned long *)&pbTarget[i+cbPattern-1-3] == ulHashPattern) // No problema here since we have 4[+] long pattern here. Overlapping (1 byte recompared) when length=4, grmbl.
          if ( *(unsigned long *)&pbTarget[i] == ulHashPattern ) // The lesson I learned from r.7- now applied in r.7: instead of extracting 'ch' having higher address now the lower address is extracted first in order (hopefully, the test confirms it)
the next 32bytes (including 'ch') to be cached i.e. to comparison part is faster.
             {
          count = countSTATIC;
          while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e. no need of comparing in-between chars.
                count--;
          }
          if ( count == 0) return(pbTarget+i);
                }
       i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
//GlobalI++;
    }
    return(NULL);
} //if (cbTarget<961)
} //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm ]

// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm [
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
char * Railgun_Quadruplet_7_9 (char * pbTarget,
     char * pbPattern,
     unsigned long cbTarget,
     unsigned long cbPattern)
{
    char * pbTargetMax = pbTarget + cbTarget;
    register unsigned long  ulHashPattern;
    unsigned long ulHashTarget;
    //unsigned long count; //r.6+
    signed long count;
    //unsigned long countSTATIC; //r.6+
    signed long countSTATIC;
//  unsigned long countRemainder;

/*
    const unsigned char SINGLET = *(char *)(pbPattern);
    const unsigned long Quadruplet2nd = SINGLET<<8;
    const unsigned long Quadruplet3rd = SINGLET<<16;
    const unsigned long Quadruplet4th = SINGLET<<24;
*/
    unsigned char SINGLET;
```

```
    unsigned long Quadruplet2nd;
    unsigned long Quadruplet3rd;
    unsigned long Quadruplet4th;

    unsigned long  AdvanceHopperGrass;

    long i; //BMH needed
//Below array is already global:
    int a, j;
    //int a, j, bm_bc[ASIZE]; //BMH needed
    unsigned char ch; //BMH needed
    unsigned long chchchch; //BMH needed
//    unsigned char lastch, firstch; //BMH needed

    if (cbPattern > cbTarget)
        return(NULL);

// Doesn't work when cbPattern = 1
// The next IF-fragment works very well with cbPattern>1, OBVIOUSLY IT MUST BE UNROLLED(but crippled with less functionality) SINCE either cbPattern=2 or cbPattern=3!
if ( cbPattern<4) { // This IF makes me unhappy: it slows down from 390KB/clock to 367KB/clock for 'fast' pattern. This fragment(for 2..3 pattern lengths) is needed because I need a function different than strchr but sticking to strstr i.e. lengths
above 1 are to be handled.
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
//        countSTATIC = cbPattern-2;

if ( cbPattern==3) {
    for ( ;; )
    {
        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
         if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else {
}
    for ( ;; )
    {
        // The line below gives for 'cbPattern'>=1:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
        // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
/*
        if ( (ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1)) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned int)cbPattern) )
            return((long)(pbTarget-cbPattern));
*/

        // The fragment below gives for 'cbPattern'>=2:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
        // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock

/*
//For 2 and 3 [
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
//         count = countSTATIC;
         count = cbPattern-2;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
         while ( count && *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) { // Crippling i.e. only 2 and 3 chars are allowed!
              count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-cbPattern+1) ) pbTarget++;
//For 2 and 3 ]
*/


        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
```

```
            return((pbTarget-2));
        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;


        // The fragment below gives for 'cbPattern'>=2:
           // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
           // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
/*
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
         count = countSTATIC>>2;
         countRemainder = countSTATIC % 4;

         while ( count && *(unsigned long *)(pbPattern+1+((count-1)<<2)) == *(unsigned long *)(pbTarget-cbPattern+1+((count-1)<<2)) ) {
               count--;
         }
            //if (count == 0) {  // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skillessness'; 312KB/clock when 1+1x4+2+1 bytes pattern: 'underdog'), otherwise 368KB/clock.
         while ( countRemainder && *(char *)(pbPattern+1+(countSTATIC-countRemainder)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-countRemainder)) ) {
               countRemainder--;
         }
         //if ( countRemainder == 0) return((long)(pbTarget-cbPattern));
         if ( count+countRemainder == 0) return((long)(pbTarget-cbPattern));
         //}
        }
*/

        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if ( cbPattern<4)
if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.
{
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = *(unsigned long *)(pbPattern);
//        countSTATIC = cbPattern-1;

    //SINGLET = *(char *)(pbPattern);
    SINGLET = ulHashPattern & 0xFF;
    Quadruplet2nd = SINGLET<<8;
    Quadruplet3rd = SINGLET<<16;
    Quadruplet4th = SINGLET<<24;

    for ( ;; )
    {
           AdvanceHopperGrass = 0;
           ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
//         count = countSTATIC;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
//                if ( countSTATIC==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) AdvanceHopperGrass++;
//               count--;
//         }
         count = cbPattern-1;
         while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                 if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        } else { // The goal here: to avoid memory accesses by stressing the registers.
    if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
         AdvanceHopperGrass++;
         if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
              AdvanceHopperGrass++;
              if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
         }
    }
           }
```

```
            AdvanceHopperGrass++;

            pbTarget = pbTarget + AdvanceHopperGrass;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if (cbTarget<961)
        //countSTATIC = cbPattern-2; //r.6+
        //countSTATIC = cbPattern-2-3;
        //countSTATIC = cbPattern-2-2; // r.6+++ I suppose that the awful degradation comes from 2bytes more (from either 'if (countSTATIC<0) countSTATIC=0;' or 'count >0' fixes) which make the function unfittable in code cache lines?!
        //countSTATIC = cbPattern-2-3; // r.7- At last no recompared bytes in-between chars
        countSTATIC = cbPattern-2-2; // r.7
        ulHashPattern = *(unsigned long *)(pbPattern);

        //chPTR=(unsigned char *)&chchchch+3;
// Next line fixes the BUG from r.6++: but with awful speed degradation! So the bug is fixed in the definitions by setting 'countSTATIC = cbPattern-2-2;', bug appears only for patterns with lengths of 4, The setback is one unnecessary comparison for
5bytes patterns, stupidly such setback exists (from before) for 4bytes as well.
//if (countSTATIC<0) countSTATIC=0;
    /* Preprocessing */
//Below 2 lines are global already:
    //for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
    //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;

    /* Searching */
    //lastch=pbPattern[cbPattern-1];
    //firstch=pbPattern[0];
    i=0;
    while (i <= cbTarget-cbPattern) {
       //ch=pbTarget[i+cbPattern-1];
       //ch=pbTarget[i];
          //if ( pbTarget[i] == pbPattern[0] && *(unsigned long *)&pbTarget[i+cbPattern-1-3] == ulHashPattern) // No problema here since we have 4[+] long pattern here. Overlapping (1 byte recompared) when length=4, grmbl.
          if ( *(unsigned long *)&pbTarget[i] == ulHashPattern ) // The lesson I learned from r.7- now applied in r.7: instead of extracting 'ch' having higher address now the lower address is extracted first in order (hopefully, the test confirms it)
the next 32bytes (including 'ch') to be cached i.e. to comparison part is faster.
             {
          count = countSTATIC;
          while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e. no need of comparing in-between chars.
               count--;
          }
          if ( count == 0) return(pbTarget+i);
                }
       i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
//GlobalI++;
    }
    return(NULL);
} //if (cbTarget<961)
} //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm ]

// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm [
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
char * Railgun_Quadruplet_7_0 (char * pbTarget,
     char * pbPattern,
     unsigned long cbTarget,
     unsigned long cbPattern)
{
    char * pbTargetMax = pbTarget + cbTarget;
    register unsigned long  ulHashPattern;
    unsigned long ulHashTarget;
    //unsigned long count; //r.6+
    signed long count;
    //unsigned long countSTATIC; //r.6+
    signed long countSTATIC;
//  unsigned long countRemainder;

/*
    const unsigned char SINGLET = *(char *)(pbPattern);
    const unsigned long Quadruplet2nd = SINGLET<<8;
    const unsigned long Quadruplet3rd = SINGLET<<16;
```

```
    const unsigned long Quadruplet4th = SINGLET<<24;
*/
    unsigned char SINGLET;
    unsigned long Quadruplet2nd;
    unsigned long Quadruplet3rd;
    unsigned long Quadruplet4th;

    unsigned long  AdvanceHopperGrass;

    long i; //BMH needed
//Below array is already global:
    int a, j;
    //int a, j, bm_bc[ASIZE]; //BMH needed
    unsigned char ch; //BMH needed
    unsigned long chchchch; //BMH needed
//    unsigned char lastch, firstch; //BMH needed

    if (cbPattern > cbTarget)
        return(NULL);

// Doesn't work when cbPattern = 1
// The next IF-fragment works very well with cbPattern>1, OBVIOUSLY IT MUST BE UNROLLED(but crippled with less functionality) SINCE either cbPattern=2 or cbPattern=3!
if ( cbPattern<4) { // This IF makes me unhappy: it slows down from 390KB/clock to 367KB/clock for 'fast' pattern. This fragment(for 2..3 pattern lengths) is needed because I need a function different than strchr but sticking to strstr i.e. lengths
above 1 are to be handled.
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
//        countSTATIC = cbPattern-2;

if ( cbPattern==3) {
    for ( ;; )
    {
        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
         if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else {
}
    for ( ;; )
    {
        // The line below gives for 'cbPattern'>=1:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
        // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
/*
        if ( (ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1)) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned int)cbPattern) )
            return((long)(pbTarget-cbPattern));
*/

        // The fragment below gives for 'cbPattern'>=2:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
        // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock

/*
//For 2 and 3 [
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
//         count = countSTATIC;
         count = cbPattern-2;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
         while ( count && *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) { // Crippling i.e. only 2 and 3 chars are allowed!
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-cbPattern+1) ) pbTarget++;
//For 2 and 3 ]
*/
```

```
        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
            return((pbTarget-2));
        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;


        // The fragment below gives for 'cbPattern'>=2:
           // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
           // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
/*
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
         count = countSTATIC>>2;
         countRemainder = countSTATIC % 4;

         while ( count && *(unsigned long *)(pbPattern+1+((count-1)<<2)) == *(unsigned long *)(pbTarget-cbPattern+1+((count-1)<<2)) ) {
               count--;
         }
            //if (count == 0) {  // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skillessness'; 312KB/clock when 1+1x4+2+1 bytes pattern: 'underdog'), otherwise 368KB/clock.
         while ( countRemainder && *(char *)(pbPattern+1+(countSTATIC-countRemainder)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-countRemainder)) ) {
               countRemainder--;
         }
         //if ( countRemainder == 0) return((long)(pbTarget-cbPattern));
         if ( count+countRemainder == 0) return((long)(pbTarget-cbPattern));
         //}
        }
*/

        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if ( cbPattern<4)
if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.
{
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = *(unsigned long *)(pbPattern);
//        countSTATIC = cbPattern-1;

    //SINGLET = *(char *)(pbPattern);
    SINGLET = ulHashPattern & 0xFF;
    Quadruplet2nd = SINGLET<<8;
    Quadruplet3rd = SINGLET<<16;
    Quadruplet4th = SINGLET<<24;

    for ( ;; )
    {
            AdvanceHopperGrass = 0;
            ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
//         count = countSTATIC;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
//                if ( countSTATIC==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) AdvanceHopperGrass++;
//               count--;
//         }
         count = cbPattern-1;
         while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                 if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        } else { // The goal here: to avoid memory accesses by stressing the registers.
    if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
         AdvanceHopperGrass++;
         if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
              AdvanceHopperGrass++;
              if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
         }
```

```
    }
            }

            AdvanceHopperGrass++;

            pbTarget = pbTarget + AdvanceHopperGrass;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if (cbTarget<961)
        //countSTATIC = cbPattern-2; //r.6+
        //countSTATIC = cbPattern-2-3;
        //countSTATIC = cbPattern-2-2; // r.6+++ I suppose that the awful degradation comes from 2bytes more (from either 'if (countSTATIC<0) countSTATIC=0;' or 'count >0' fixes) which make the function unfittable in code cache lines?!
        //countSTATIC = cbPattern-2-3; // r.7- At last no recompared bytes in-between chars
        countSTATIC = cbPattern-2-2; // r.7
        ulHashPattern = *(unsigned long *)(pbPattern);

        //chPTR=(unsigned char *)&chchchch+3;
// Next line fixes the BUG from r.6++: but with awful speed degradation! So the bug is fixed in the definitions by setting 'countSTATIC = cbPattern-2-2;', bug appears only for patterns with lengths of 4, The setback is one unnecessary comparison for
5bytes patterns, stupidly such setback exists (from before) for 4bytes as well.
//if (countSTATIC<0) countSTATIC=0;
    /* Preprocessing */
//Below 2 lines are global already:
    //for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
    //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;

    /* Searching */
    //lastch=pbPattern[cbPattern-1];
    //firstch=pbPattern[0];
    i=0;
    while (i <= cbTarget-cbPattern) {
       //ch=pbTarget[i+cbPattern-1];
       //ch=pbTarget[i];
          //if ( pbTarget[i] == pbPattern[0] && *(unsigned long *)&pbTarget[i+cbPattern-1-3] == ulHashPattern) // No problema here since we have 4[+] long pattern here. Overlapping (1 byte recompared) when length=4, grmbl.
          if ( *(unsigned long *)&pbTarget[i] == ulHashPattern ) // The lesson I learned from r.7- now applied in r.7: instead of extracting 'ch' having higher address now the lower address is extracted first in order (hopefully, the test confirms it)
the next 32bytes (including 'ch') to be cached i.e. to comparison part is faster.
             {
         count = countSTATIC;
         while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e. no need of comparing in-between chars.
               count--;
         }
         if ( count == 0) return(pbTarget+i);
                 }
       i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
//GlobalI++;
    }
    return(NULL);
} //if (cbTarget<961)
} //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm ]

// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm [
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
char * Railgun_Quadruplet_7_a (char * pbTarget,
     char * pbPattern,
     unsigned long cbTarget,
     unsigned long cbPattern)
{
    char * pbTargetMax = pbTarget + cbTarget;
    register unsigned long  ulHashPattern;
    unsigned long ulHashTarget;
    //unsigned long count; //r.6+
    signed long count;
    //unsigned long countSTATIC; //r.6+
    signed long countSTATIC;
//  unsigned long countRemainder;

/*
```

```
    const unsigned char SINGLET = *(char *)(pbPattern);
    const unsigned long Quadruplet2nd = SINGLET<<8;
    const unsigned long Quadruplet3rd = SINGLET<<16;
    const unsigned long Quadruplet4th = SINGLET<<24;
*/
    unsigned char SINGLET;
    unsigned long Quadruplet2nd;
    unsigned long Quadruplet3rd;
    unsigned long Quadruplet4th;

    unsigned long  AdvanceHopperGrass;

    long i; //BMH needed
//Below array is already global:
    int a, j;
    //int a, j, bm_bc[ASIZE]; //BMH needed
    unsigned char ch; //BMH needed
    unsigned long chchchch; //BMH needed
//    unsigned char lastch, firstch; //BMH needed

    if (cbPattern > cbTarget)
        return(NULL);

// Doesn't work when cbPattern = 1
// The next IF-fragment works very well with cbPattern>1, OBVIOUSLY IT MUST BE UNROLLED(but crippled with less functionality) SINCE either cbPattern=2 or cbPattern=3!
if ( cbPattern<4) { // This IF makes me unhappy: it slows down from 390KB/clock to 367KB/clock for 'fast' pattern. This fragment(for 2..3 pattern lengths) is needed because I need a function different than strchr but sticking to strstr i.e. lengths
above 1 are to be handled.
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
//        countSTATIC = cbPattern-2;

if ( cbPattern==3) {
    for ( ;; )
    {
        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
         if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else {
}
    for ( ;; )
    {
        // The line below gives for 'cbPattern'>=1:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
        // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
/*
        if ( (ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1)) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned int)cbPattern) )
            return((long)(pbTarget-cbPattern));
*/

        // The fragment below gives for 'cbPattern'>=2:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
        // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock

/*
//For 2 and 3 [
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
//         count = countSTATIC;
         count = cbPattern-2;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
         while ( count && *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) { // Crippling i.e. only 2 and 3 chars are allowed!
              count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        }
```

```
        if ( (char)(ulHashPattern>>8) != *(pbTarget-cbPattern+1) ) pbTarget++;
//For 2 and 3 ]
*/


        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
            return((pbTarget-2));
        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;


        // The fragment below gives for 'cbPattern'>=2:
           // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
           // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
/*
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
         count = countSTATIC>>2;
         countRemainder = countSTATIC % 4;

         while ( count && *(unsigned long *)(pbPattern+1+((count-1)<<2)) == *(unsigned long *)(pbTarget-cbPattern+1+((count-1)<<2)) ) {
               count--;
         }
            //if (count == 0) {  // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skillessness'; 312KB/clock when 1+1x4+2+1 bytes pattern: 'underdog'), otherwise 368KB/clock.
         while ( countRemainder && *(char *)(pbPattern+1+(countSTATIC-countRemainder)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-countRemainder)) ) {
               countRemainder--;
         }
         //if ( countRemainder == 0) return((long)(pbTarget-cbPattern));
         if ( count+countRemainder == 0) return((long)(pbTarget-cbPattern));
         //}
        }
*/

        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if ( cbPattern<4)
if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.
{
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = *(unsigned long *)(pbPattern);
//        countSTATIC = cbPattern-1;

    //SINGLET = *(char *)(pbPattern);
    SINGLET = ulHashPattern & 0xFF;
    Quadruplet2nd = SINGLET<<8;
    Quadruplet3rd = SINGLET<<16;
    Quadruplet4th = SINGLET<<24;

    for ( ;; )
    {
            AdvanceHopperGrass = 0;
            ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
//         count = countSTATIC;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
//                if ( countSTATIC==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) AdvanceHopperGrass++;
//               count--;
//         }
         count = cbPattern-1;
         while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                 if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        } else { // The goal here: to avoid memory accesses by stressing the registers.
    if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
         AdvanceHopperGrass++;
         if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
```

```
            AdvanceHopperGrass++;
            if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
        }
    }
          }

          AdvanceHopperGrass++;

          pbTarget = pbTarget + AdvanceHopperGrass;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if (cbTarget<961)
        //countSTATIC = cbPattern-2; //r.6+
        //countSTATIC = cbPattern-2-3;
        //countSTATIC = cbPattern-2-2; // r.6+++ I suppose that the awful degradation comes from 2bytes more (from either 'if (countSTATIC<0) countSTATIC=0;' or 'count >0' fixes) which make the function unfittable in code cache lines?!
        //countSTATIC = cbPattern-2-3; // r.7- At last no recompared bytes in-between chars
        countSTATIC = cbPattern-2-2; // r.7
        ulHashPattern = *(unsigned long *)(pbPattern);

        //chPTR=(unsigned char *)&chchchch+3;
// Next line fixes the BUG from r.6++: but with awful speed degradation! So the bug is fixed in the definitions by setting 'countSTATIC = cbPattern-2-2;', bug appears only for patterns with lengths of 4, The setback is one unnecessary comparison for
5bytes patterns, stupidly such setback exists (from before) for 4bytes as well.
//if (countSTATIC<0) countSTATIC=0;
    /* Preprocessing */
//Below 2 lines are global already:
    //for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
    //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;

    /* Searching */
    //lastch=pbPattern[cbPattern-1];
    //firstch=pbPattern[0];
    i=0;
    while (i <= cbTarget-cbPattern) {
       //ch=pbTarget[i+cbPattern-1];
       //ch=pbTarget[i];
          //if ( pbTarget[i] == pbPattern[0] && *(unsigned long *)&pbTarget[i+cbPattern-1-3] == ulHashPattern) // No problema here since we have 4[+] long pattern here. Overlapping (1 byte recompared) when length=4, grmbl.
          if ( *(unsigned long *)&pbTarget[i] == ulHashPattern ) // The lesson I learned from r.7- now applied in r.7: instead of extracting 'ch' having higher address now the lower address is extracted first in order (hopefully, the test confirms it)
the next 32bytes (including 'ch') to be cached i.e. to comparison part is faster.
             {
         count = countSTATIC;
         while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e. no need of comparing in-between chars.
               count--;
         }
         if ( count == 0) return(pbTarget+i);
                }
       i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
//GlobalI++;
    }
    return(NULL);
} //if (cbTarget<961)
} //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm ]

// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm [
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
char * Railgun_Quadruplet_7_b (char * pbTarget,
     char * pbPattern,
     unsigned long cbTarget,
     unsigned long cbPattern)
{
    char * pbTargetMax = pbTarget + cbTarget;
    register unsigned long  ulHashPattern;
    unsigned long ulHashTarget;
    //unsigned long count; //r.6+
    signed long count;
    //unsigned long countSTATIC; //r.6+
    signed long countSTATIC;
```

```
//  unsigned long countRemainder;

/*
    const unsigned char SINGLET = *(char *)(pbPattern);
    const unsigned long Quadruplet2nd = SINGLET<<8;
    const unsigned long Quadruplet3rd = SINGLET<<16;
    const unsigned long Quadruplet4th = SINGLET<<24;
*/
    unsigned char SINGLET;
    unsigned long Quadruplet2nd;
    unsigned long Quadruplet3rd;
    unsigned long Quadruplet4th;

    unsigned long  AdvanceHopperGrass;

    long i; //BMH needed
//Below array is already global:
    int a, j;
    //int a, j, bm_bc[ASIZE]; //BMH needed
    unsigned char ch; //BMH needed
    unsigned long chchchch; //BMH needed
//    unsigned char lastch, firstch; //BMH needed

    if (cbPattern > cbTarget)
        return(NULL);

// Doesn't work when cbPattern = 1
// The next IF-fragment works very well with cbPattern>1, OBVIOUSLY IT MUST BE UNROLLED(but crippled with less functionality) SINCE either cbPattern=2 or cbPattern=3!
if ( cbPattern<4) { // This IF makes me unhappy: it slows down from 390KB/clock to 367KB/clock for 'fast' pattern. This fragment(for 2..3 pattern lengths) is needed because I need a function different than strchr but sticking to strstr i.e. lengths
above 1 are to be handled.
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
//        countSTATIC = cbPattern-2;

if ( cbPattern==3) {
    for ( ;; )
    {
        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
         if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else {
}
    for ( ;; )
    {
        // The line below gives for 'cbPattern'>=1:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
        // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
/*
        if ( (ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1)) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned int)cbPattern) )
            return((long)(pbTarget-cbPattern));
*/

        // The fragment below gives for 'cbPattern'>=2:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
        // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock

/*
//For 2 and 3 [
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
//         count = countSTATIC;
         count = cbPattern-2;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
         while ( count && *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) { // Crippling i.e. only 2 and 3 chars are allowed!
             count--;
```

```
         }
         if ( count == 0) return((pbTarget-cbPattern));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-cbPattern+1) ) pbTarget++;
//For 2 and 3 ]
*/


        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
            return((pbTarget-2));
        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;


        // The fragment below gives for 'cbPattern'>=2:
           // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
           // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
/*
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
         count = countSTATIC>>2;
         countRemainder = countSTATIC % 4;

         while ( count && *(unsigned long *)(pbPattern+1+((count-1)<<2)) == *(unsigned long *)(pbTarget-cbPattern+1+((count-1)<<2)) ) {
               count--;
         }
            //if (count == 0) {  // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skillessness'; 312KB/clock when 1+1x4+2+1 bytes pattern: 'underdog'), otherwise 368KB/clock.
         while ( countRemainder && *(char *)(pbPattern+1+(countSTATIC-countRemainder)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-countRemainder)) ) {
               countRemainder--;
         }
         //if ( countRemainder == 0) return((long)(pbTarget-cbPattern));
         if ( count+countRemainder == 0) return((long)(pbTarget-cbPattern));
         //}
        }
*/

        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if ( cbPattern<4)
if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.
{
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = *(unsigned long *)(pbPattern);
//        countSTATIC = cbPattern-1;

    //SINGLET = *(char *)(pbPattern);
    SINGLET = ulHashPattern & 0xFF;
    Quadruplet2nd = SINGLET<<8;
    Quadruplet3rd = SINGLET<<16;
    Quadruplet4th = SINGLET<<24;

    for ( ;; )
    {
           AdvanceHopperGrass = 0;
           ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
//         count = countSTATIC;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
//                if ( countSTATIC==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) AdvanceHopperGrass++;
//               count--;
//         }
         count = cbPattern-1;
         while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        } else { // The goal here: to avoid memory accesses by stressing the registers.
```

```
    if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
         AdvanceHopperGrass++;
         if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
              AdvanceHopperGrass++;
              if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
         }
    }
            }

            AdvanceHopperGrass++;

            pbTarget = pbTarget + AdvanceHopperGrass;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if (cbTarget<961)
        //countSTATIC = cbPattern-2; //r.6+
        //countSTATIC = cbPattern-2-3;
        //countSTATIC = cbPattern-2-2; // r.6+++ I suppose that the awful degradation comes from 2bytes more (from either 'if (countSTATIC<0) countSTATIC=0;' or 'count >0' fixes) which make the function unfittable in code cache lines?!
        //countSTATIC = cbPattern-2-3; // r.7- At last no recompared bytes in-between chars
        countSTATIC = cbPattern-2-2; // r.7
        ulHashPattern = *(unsigned long *)(pbPattern);

        //chPTR=(unsigned char *)&chchchch+3;
// Next line fixes the BUG from r.6++: but with awful speed degradation! So the bug is fixed in the definitions by setting 'countSTATIC = cbPattern-2-2;', bug appears only for patterns with lengths of 4, The setback is one unnecessary comparison for
5bytes patterns, stupidly such setback exists (from before) for 4bytes as well.
//if (countSTATIC<0) countSTATIC=0;
    /* Preprocessing */
//Below 2 lines are global already:
    //for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
    //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;

    /* Searching */
    //lastch=pbPattern[cbPattern-1];
    //firstch=pbPattern[0];
    i=0;
    while (i <= cbTarget-cbPattern) {
       //ch=pbTarget[i+cbPattern-1];
       //ch=pbTarget[i];
          //if ( pbTarget[i] == pbPattern[0] && *(unsigned long *)&pbTarget[i+cbPattern-1-3] == ulHashPattern) // No problema here since we have 4[+] long pattern here. Overlapping (1 byte recompared) when length=4, grmbl.
          if ( *(unsigned long *)&pbTarget[i] == ulHashPattern ) // The lesson I learned from r.7- now applied in r.7: instead of extracting 'ch' having higher address now the lower address is extracted first in order (hopefully, the test confirms it)
the next 32bytes (including 'ch') to be cached i.e. to comparison part is faster.
             {
         count = countSTATIC;
         while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e. no need of comparing in-between chars.
               count--;
         }
         if ( count == 0) return(pbTarget+i);
                }
       i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
//GlobalI++;
    }
    return(NULL);
} //if (cbTarget<961)
} //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm ]

// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm [
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
char * Railgun_Quadruplet_7_c (char * pbTarget,
     char * pbPattern,
     unsigned long cbTarget,
     unsigned long cbPattern)
{
    char * pbTargetMax = pbTarget + cbTarget;
    register unsigned long  ulHashPattern;
    unsigned long ulHashTarget;
    //unsigned long count; //r.6+
```

```
    signed long count;
    //unsigned long countSTATIC; //r.6+
    signed long countSTATIC;
//  unsigned long countRemainder;

/*
    const unsigned char SINGLET = *(char *)(pbPattern);
    const unsigned long Quadruplet2nd = SINGLET<<8;
    const unsigned long Quadruplet3rd = SINGLET<<16;
    const unsigned long Quadruplet4th = SINGLET<<24;
*/
    unsigned char SINGLET;
    unsigned long Quadruplet2nd;
    unsigned long Quadruplet3rd;
    unsigned long Quadruplet4th;

    unsigned long  AdvanceHopperGrass;

    long i; //BMH needed
//Below array is already global:
    int a, j;
    //int a, j, bm_bc[ASIZE]; //BMH needed
    unsigned char ch; //BMH needed
    unsigned long chchchch; //BMH needed
//    unsigned char lastch, firstch; //BMH needed

    if (cbPattern > cbTarget)
        return(NULL);

// Doesn't work when cbPattern = 1
// The next IF-fragment works very well with cbPattern>1, OBVIOUSLY IT MUST BE UNROLLED(but crippled with less functionality) SINCE either cbPattern=2 or cbPattern=3!
if ( cbPattern<4) { // This IF makes me unhappy: it slows down from 390KB/clock to 367KB/clock for 'fast' pattern. This fragment(for 2..3 pattern lengths) is needed because I need a function different than strchr but sticking to strstr i.e. lengths
above 1 are to be handled.
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
//        countSTATIC = cbPattern-2;

if ( cbPattern==3) {
    for ( ;; )
    {
        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
         if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else {
}
    for ( ;; )
    {
        // The line below gives for 'cbPattern'>=1:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
        // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
/*
        if ( (ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1)) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned int)cbPattern) )
            return((long)(pbTarget-cbPattern));
*/

        // The fragment below gives for 'cbPattern'>=2:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
        // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock

/*
//For 2 and 3 [
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
//         count = countSTATIC;
         count = cbPattern-2;
```

```
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
         while ( count && *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) { // Crippling i.e. only 2 and 3 chars are allowed!
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-cbPattern+1) ) pbTarget++;
//For 2 and 3 ]
*/


        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
            return((pbTarget-2));
        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;


        // The fragment below gives for 'cbPattern'>=2:
           // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
           // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
/*
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
         count = countSTATIC>>2;
         countRemainder = countSTATIC % 4;

         while ( count && *(unsigned long *)(pbPattern+1+((count-1)<<2)) == *(unsigned long *)(pbTarget-cbPattern+1+((count-1)<<2)) ) {
               count--;
         }
            //if (count == 0) {  // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skillessness'; 312KB/clock when 1+1x4+2+1 bytes pattern: 'underdog'), otherwise 368KB/clock.
         while ( countRemainder && *(char *)(pbPattern+1+(countSTATIC-countRemainder)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-countRemainder)) ) {
               countRemainder--;
         }
         //if ( countRemainder == 0) return((long)(pbTarget-cbPattern));
         if ( count+countRemainder == 0) return((long)(pbTarget-cbPattern));
         //}
        }
*/

        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if ( cbPattern<4)
if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.
{
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = *(unsigned long *)(pbPattern);
//        countSTATIC = cbPattern-1;

    //SINGLET = *(char *)(pbPattern);
    SINGLET = ulHashPattern & 0xFF;
    Quadruplet2nd = SINGLET<<8;
    Quadruplet3rd = SINGLET<<16;
    Quadruplet4th = SINGLET<<24;

    for ( ;; )
    {
            AdvanceHopperGrass = 0;
            ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
//         count = countSTATIC;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
//                if ( countSTATIC==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) AdvanceHopperGrass++;
//               count--;
//         }
         count = cbPattern-1;
         while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                 if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
               count--;
```

```
         }
         if ( count == 0) return((pbTarget-cbPattern));
        } else { // The goal here: to avoid memory accesses by stressing the registers.
    if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
         AdvanceHopperGrass++;
         if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
              AdvanceHopperGrass++;
              if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
         }
    }
           }

           AdvanceHopperGrass++;

           pbTarget = pbTarget + AdvanceHopperGrass;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if (cbTarget<961)
        //countSTATIC = cbPattern-2; //r.6+
        //countSTATIC = cbPattern-2-3;
        //countSTATIC = cbPattern-2-2; // r.6+++ I suppose that the awful degradation comes from 2bytes more (from either 'if (countSTATIC<0) countSTATIC=0;' or 'count >0' fixes) which make the function unfittable in code cache lines?!
        //countSTATIC = cbPattern-2-3; // r.7- At last no recompared bytes in-between chars
        countSTATIC = cbPattern-2-2; // r.7
        ulHashPattern = *(unsigned long *)(pbPattern);

        //chPTR=(unsigned char *)&chchchch+3;
// Next line fixes the BUG from r.6++: but with awful speed degradation! So the bug is fixed in the definitions by setting 'countSTATIC = cbPattern-2-2;', bug appears only for patterns with lengths of 4, The setback is one unnecessary comparison for
5bytes patterns, stupidly such setback exists (from before) for 4bytes as well.
//if (countSTATIC<0) countSTATIC=0;
    /* Preprocessing */
//Below 2 lines are global already:
    //for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
    //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;

    /* Searching */
    //lastch=pbPattern[cbPattern-1];
    //firstch=pbPattern[0];
    i=0;
    while (i <= cbTarget-cbPattern) {
       //ch=pbTarget[i+cbPattern-1];
       //ch=pbTarget[i];
          //if ( pbTarget[i] == pbPattern[0] && *(unsigned long *)&pbTarget[i+cbPattern-1-3] == ulHashPattern) // No problema here since we have 4[+] long pattern here. Overlapping (1 byte recompared) when length=4, grmbl.
          if ( *(unsigned long *)&pbTarget[i] == ulHashPattern ) // The lesson I learned from r.7- now applied in r.7: instead of extracting 'ch' having higher address now the lower address is extracted first in order (hopefully, the test confirms it)
the next 32bytes (including 'ch') to be cached i.e. to comparison part is faster.
             {
         count = countSTATIC;
         while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e. no need of comparing in-between chars.
               count--;
         }
         if ( count == 0) return(pbTarget+i);
                }
       i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
//GlobalI++;
    }
    return(NULL);
} //if (cbTarget<961)
} //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm ]

// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm [
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
char * Railgun_Quadruplet_7_d (char * pbTarget,
     char * pbPattern,
     unsigned long cbTarget,
     unsigned long cbPattern)
{
    char * pbTargetMax = pbTarget + cbTarget;
```

```
    register unsigned long  ulHashPattern;
    unsigned long ulHashTarget;
    //unsigned long count; //r.6+
    signed long count;
    //unsigned long countSTATIC; //r.6+
    signed long countSTATIC;
//  unsigned long countRemainder;

/*
    const unsigned char SINGLET = *(char *)(pbPattern);
    const unsigned long Quadruplet2nd = SINGLET<<8;
    const unsigned long Quadruplet3rd = SINGLET<<16;
    const unsigned long Quadruplet4th = SINGLET<<24;
*/
    unsigned char SINGLET;
    unsigned long Quadruplet2nd;
    unsigned long Quadruplet3rd;
    unsigned long Quadruplet4th;

    unsigned long  AdvanceHopperGrass;

    long i; //BMH needed
//Below array is already global:
    int a, j;
    //int a, j, bm_bc[ASIZE]; //BMH needed
    unsigned char ch; //BMH needed
    unsigned long chchchch; //BMH needed
//    unsigned char lastch, firstch; //BMH needed

    if (cbPattern > cbTarget)
        return(NULL);

// Doesn't work when cbPattern = 1
// The next IF-fragment works very well with cbPattern>1, OBVIOUSLY IT MUST BE UNROLLED(but crippled with less functionality) SINCE either cbPattern=2 or cbPattern=3!
if ( cbPattern<4) { // This IF makes me unhappy: it slows down from 390KB/clock to 367KB/clock for 'fast' pattern. This fragment(for 2..3 pattern lengths) is needed because I need a function different than strchr but sticking to strstr i.e. lengths
above 1 are to be handled.
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
//        countSTATIC = cbPattern-2;

if ( cbPattern==3) {
    for ( ;; )
    {
        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
         if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else {
}
    for ( ;; )
    {
        // The line below gives for 'cbPattern'>=1:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
        // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
/*
        if ( (ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1)) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned int)cbPattern) )
            return((long)(pbTarget-cbPattern));
*/

        // The fragment below gives for 'cbPattern'>=2:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
        // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock

/*
//For 2 and 3 [
```

```
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
//         count = countSTATIC;
         count = cbPattern-2;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
         while ( count && *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) { // Crippling i.e. only 2 and 3 chars are allowed!
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-cbPattern+1) ) pbTarget++;
//For 2 and 3 ]
*/


        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
            return((pbTarget-2));
        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;


        // The fragment below gives for 'cbPattern'>=2:
           // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
           // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
/*
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
         count = countSTATIC>>2;
         countRemainder = countSTATIC % 4;

         while ( count && *(unsigned long *)(pbPattern+1+((count-1)<<2)) == *(unsigned long *)(pbTarget-cbPattern+1+((count-1)<<2)) ) {
               count--;
         }
            //if (count == 0) {  // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skillessness'; 312KB/clock when 1+1x4+2+1 bytes pattern: 'underdog'), otherwise 368KB/clock.
         while ( countRemainder && *(char *)(pbPattern+1+(countSTATIC-countRemainder)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-countRemainder)) ) {
               countRemainder--;
         }
         //if ( countRemainder == 0) return((long)(pbTarget-cbPattern));
         if ( count+countRemainder == 0) return((long)(pbTarget-cbPattern));
         //}
        }
*/

        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if ( cbPattern<4)
if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.
{
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = *(unsigned long *)(pbPattern);
//        countSTATIC = cbPattern-1;

    //SINGLET = *(char *)(pbPattern);
    SINGLET = ulHashPattern & 0xFF;
    Quadruplet2nd = SINGLET<<8;
    Quadruplet3rd = SINGLET<<16;
    Quadruplet4th = SINGLET<<24;

    for ( ;; )
    {
            AdvanceHopperGrass = 0;
            ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
//         count = countSTATIC;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
//                if ( countSTATIC==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) AdvanceHopperGrass++;
//               count--;
//         }
         count = cbPattern-1;
```

```
         while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                  if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        } else { // The goal here: to avoid memory accesses by stressing the registers.
    if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
         AdvanceHopperGrass++;
         if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
              AdvanceHopperGrass++;
              if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
         }
    }
           }

           AdvanceHopperGrass++;

           pbTarget = pbTarget + AdvanceHopperGrass;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if (cbTarget<961)
        //countSTATIC = cbPattern-2; //r.6+
        //countSTATIC = cbPattern-2-3;
        //countSTATIC = cbPattern-2-2; // r.6+++ I suppose that the awful degradation comes from 2bytes more (from either 'if (countSTATIC<0) countSTATIC=0;' or 'count >0' fixes) which make the function unfittable in code cache lines?!
        //countSTATIC = cbPattern-2-3; // r.7- At last no recompared bytes in-between chars
        countSTATIC = cbPattern-2-2; // r.7
        ulHashPattern = *(unsigned long *)(pbPattern);

        //chPTR=(unsigned char *)&chchchch+3;
// Next line fixes the BUG from r.6++: but with awful speed degradation! So the bug is fixed in the definitions by setting 'countSTATIC = cbPattern-2-2;', bug appears only for patterns with lengths of 4, The setback is one unnecessary comparison for
5bytes patterns, stupidly such setback exists (from before) for 4bytes as well.
//if (countSTATIC<0) countSTATIC=0;
    /* Preprocessing */
//Below 2 lines are global already:
    //for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
    //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;

    /* Searching */
    //lastch=pbPattern[cbPattern-1];
    //firstch=pbPattern[0];
    i=0;
    while (i <= cbTarget-cbPattern) {
       //ch=pbTarget[i+cbPattern-1];
       //ch=pbTarget[i];
          //if ( pbTarget[i] == pbPattern[0] && *(unsigned long *)&pbTarget[i+cbPattern-1-3] == ulHashPattern) // No problema here since we have 4[+] long pattern here. Overlapping (1 byte recompared) when length=4, grmbl.
          if ( *(unsigned long *)&pbTarget[i] == ulHashPattern ) // The lesson I learned from r.7- now applied in r.7: instead of extracting 'ch' having higher address now the lower address is extracted first in order (hopefully, the test confirms it)
the next 32bytes (including 'ch') to be cached i.e. to comparison part is faster.
             {
         count = countSTATIC;
         while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e. no need of comparing in-between chars.
               count--;
         }
         if ( count == 0) return(pbTarget+i);
                 }
       i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
//GlobalI++;
    }
    return(NULL);
} //if (cbTarget<961)
} //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm ]

// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm [
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
char * Railgun_Quadruplet_7_e (char * pbTarget,
     char * pbPattern,
     unsigned long cbTarget,
```

```
     unsigned long cbPattern)
{
    char * pbTargetMax = pbTarget + cbTarget;
    register unsigned long  ulHashPattern;
    unsigned long ulHashTarget;
    //unsigned long count; //r.6+
    signed long count;
    //unsigned long countSTATIC; //r.6+
    signed long countSTATIC;
//  unsigned long countRemainder;

/*
    const unsigned char SINGLET = *(char *)(pbPattern);
    const unsigned long Quadruplet2nd = SINGLET<<8;
    const unsigned long Quadruplet3rd = SINGLET<<16;
    const unsigned long Quadruplet4th = SINGLET<<24;
*/
    unsigned char SINGLET;
    unsigned long Quadruplet2nd;
    unsigned long Quadruplet3rd;
    unsigned long Quadruplet4th;

    unsigned long  AdvanceHopperGrass;

    long i; //BMH needed
//Below array is already global:
    int a, j;
    //int a, j, bm_bc[ASIZE]; //BMH needed
    unsigned char ch; //BMH needed
    unsigned long chchchch; //BMH needed
//    unsigned char lastch, firstch; //BMH needed

    if (cbPattern > cbTarget)
        return(NULL);

// Doesn't work when cbPattern = 1
// The next IF-fragment works very well with cbPattern>1, OBVIOUSLY IT MUST BE UNROLLED(but crippled with less functionality) SINCE either cbPattern=2 or cbPattern=3!
if ( cbPattern<4) { // This IF makes me unhappy: it slows down from 390KB/clock to 367KB/clock for 'fast' pattern. This fragment(for 2..3 pattern lengths) is needed because I need a function different than strchr but sticking to strstr i.e. lengths
above 1 are to be handled.
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
//        countSTATIC = cbPattern-2;

if ( cbPattern==3) {
    for ( ;; )
    {
        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
         if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else {
}
    for ( ;; )
    {
        // The line below gives for 'cbPattern'>=1:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
        // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
/*
        if ( (ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1)) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned int)cbPattern) )
            return((long)(pbTarget-cbPattern));
*/

        // The fragment below gives for 'cbPattern'>=2:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
        // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock
```

```
/*
//For 2 and 3 [
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
//         count = countSTATIC;
         count = cbPattern-2;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
         while ( count && *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) { // Crippling i.e. only 2 and 3 chars are allowed!
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-cbPattern+1) ) pbTarget++;
//For 2 and 3 ]
*/


        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
            return((pbTarget-2));
        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;


        // The fragment below gives for 'cbPattern'>=2:
           // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
           // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
/*
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
         count = countSTATIC>>2;
         countRemainder = countSTATIC % 4;

         while ( count && *(unsigned long *)(pbPattern+1+((count-1)<<2)) == *(unsigned long *)(pbTarget-cbPattern+1+((count-1)<<2)) ) {
               count--;
         }
            //if (count == 0) {  // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skillessness'; 312KB/clock when 1+1x4+2+1 bytes pattern: 'underdog'), otherwise 368KB/clock.
         while ( countRemainder && *(char *)(pbPattern+1+(countSTATIC-countRemainder)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-countRemainder)) ) {
               countRemainder--;
         }
         //if ( countRemainder == 0) return((long)(pbTarget-cbPattern));
         if ( count+countRemainder == 0) return((long)(pbTarget-cbPattern));
         //}
        }
*/

        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if ( cbPattern<4)
if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.
{
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = *(unsigned long *)(pbPattern);
//        countSTATIC = cbPattern-1;

    //SINGLET = *(char *)(pbPattern);
    SINGLET = ulHashPattern & 0xFF;
    Quadruplet2nd = SINGLET<<8;
    Quadruplet3rd = SINGLET<<16;
    Quadruplet4th = SINGLET<<24;

    for ( ;; )
    {
            AdvanceHopperGrass = 0;
            ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
//         count = countSTATIC;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
//                if ( countSTATIC==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) AdvanceHopperGrass++;
```

```
//              count--;
//         }
         count = cbPattern-1;
         while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                  if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
               count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        } else { // The goal here: to avoid memory accesses by stressing the registers.
    if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
         AdvanceHopperGrass++;
         if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
              AdvanceHopperGrass++;
              if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
         }
    }
           }

           AdvanceHopperGrass++;

           pbTarget = pbTarget + AdvanceHopperGrass;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if (cbTarget<961)
        //countSTATIC = cbPattern-2; //r.6+
        //countSTATIC = cbPattern-2-3;
        //countSTATIC = cbPattern-2-2; // r.6+++ I suppose that the awful degradation comes from 2bytes more (from either 'if (countSTATIC<0) countSTATIC=0;' or 'count >0' fixes) which make the function unfittable in code cache lines?!
        //countSTATIC = cbPattern-2-3; // r.7- At last no recompared bytes in-between chars
        countSTATIC = cbPattern-2-2; // r.7
        ulHashPattern = *(unsigned long *)(pbPattern);

        //chPTR=(unsigned char *)&chchchch+3;
// Next line fixes the BUG from r.6++: but with awful speed degradation! So the bug is fixed in the definitions by setting 'countSTATIC = cbPattern-2-2;', bug appears only for patterns with lengths of 4, The setback is one unnecessary comparison for
5bytes patterns, stupidly such setback exists (from before) for 4bytes as well.
//if (countSTATIC<0) countSTATIC=0;
    /* Preprocessing */
//Below 2 lines are global already:
    //for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
    //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;

    /* Searching */
    //lastch=pbPattern[cbPattern-1];
    //firstch=pbPattern[0];
    i=0;
    while (i <= cbTarget-cbPattern) {
       //ch=pbTarget[i+cbPattern-1];
       //ch=pbTarget[i];
          //if ( pbTarget[i] == pbPattern[0] && *(unsigned long *)&pbTarget[i+cbPattern-1-3] == ulHashPattern) // No problema here since we have 4[+] long pattern here. Overlapping (1 byte recompared) when length=4, grmbl.
          if ( *(unsigned long *)&pbTarget[i] == ulHashPattern ) // The lesson I learned from r.7- now applied in r.7: instead of extracting 'ch' having higher address now the lower address is extracted first in order (hopefully, the test confirms it)
the next 32bytes (including 'ch') to be cached i.e. to comparison part is faster.
             {
         count = countSTATIC;
         while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e. no need of comparing in-between chars.
               count--;
         }
         if ( count == 0) return(pbTarget+i);
                }
       i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
//GlobalI++;
    }
    return(NULL);
} //if (cbTarget<961)
} //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm ]

// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm [
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
```

```
char * Railgun_Quadruplet_7_f (char * pbTarget,
     char * pbPattern,
     unsigned long cbTarget,
     unsigned long cbPattern)
{
    char * pbTargetMax = pbTarget + cbTarget;
    register unsigned long  ulHashPattern;
    unsigned long ulHashTarget;
    //unsigned long count; //r.6+
    signed long count;
    //unsigned long countSTATIC; //r.6+
    signed long countSTATIC;
//  unsigned long countRemainder;

/*
    const unsigned char SINGLET = *(char *)(pbPattern);
    const unsigned long Quadruplet2nd = SINGLET<<8;
    const unsigned long Quadruplet3rd = SINGLET<<16;
    const unsigned long Quadruplet4th = SINGLET<<24;
*/
    unsigned char SINGLET;
    unsigned long Quadruplet2nd;
    unsigned long Quadruplet3rd;
    unsigned long Quadruplet4th;

    unsigned long  AdvanceHopperGrass;

    long i; //BMH needed
//Below array is already global:
    int a, j;
    //int a, j, bm_bc[ASIZE]; //BMH needed
    unsigned char ch; //BMH needed
    unsigned long chchchch; //BMH needed
//    unsigned char lastch, firstch; //BMH needed

    if (cbPattern > cbTarget)
        return(NULL);

// Doesn't work when cbPattern = 1
// The next IF-fragment works very well with cbPattern>1, OBVIOUSLY IT MUST BE UNROLLED(but crippled with less functionality) SINCE either cbPattern=2 or cbPattern=3!
if ( cbPattern<4) { // This IF makes me unhappy: it slows down from 390KB/clock to 367KB/clock for 'fast' pattern. This fragment(for 2..3 pattern lengths) is needed because I need a function different than strchr but sticking to strstr i.e. lengths
above 1 are to be handled.
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
//        countSTATIC = cbPattern-2;

if ( cbPattern==3) {
    for ( ;; )
    {
        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
         if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else {
}
    for ( ;; )
    {
        // The line below gives for 'cbPattern'>=1:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
        // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
/*
        if ( (ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1)) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned int)cbPattern) )
            return((long)(pbTarget-cbPattern));
*/
```

```
        // The fragment below gives for 'cbPattern'>=2:
        // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
        // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock

/*
//For 2 and 3 [
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
//         count = countSTATIC;
         count = cbPattern-2;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
         while ( count && *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) { // Crippling i.e. only 2 and 3 chars are allowed!
              count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        }
        if ( (char)(ulHashPattern>>8) != *(pbTarget-cbPattern+1) ) pbTarget++;
//For 2 and 3 ]
*/


        if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
            return((pbTarget-2));
        if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;


        // The fragment below gives for 'cbPattern'>=2:
           // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
           // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
/*
        if ( ulHashPattern == ( (*(char *)(pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
         count = countSTATIC>>2;
         countRemainder = countSTATIC % 4;

         while ( count && *(unsigned long *)(pbPattern+1+((count-1)<<2)) == *(unsigned long *)(pbTarget-cbPattern+1+((count-1)<<2)) ) {
              count--;
         }
            //if (count == 0) {  // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skillessness'; 312KB/clock when 1+1x4+2+1 bytes pattern: 'underdog'), otherwise 368KB/clock.
         while ( countRemainder && *(char *)(pbPattern+1+(countSTATIC-countRemainder)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-countRemainder)) ) {
              countRemainder--;
         }
         //if ( countRemainder == 0) return((long)(pbTarget-cbPattern));
         if ( count+countRemainder == 0) return((long)(pbTarget-cbPattern));
         //}
        }
*/

        pbTarget++;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if ( cbPattern<4)
if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.
{
        pbTarget = pbTarget+cbPattern;
        ulHashPattern = *(unsigned long *)(pbPattern);
//        countSTATIC = cbPattern-1;

    //SINGLET = *(char *)(pbPattern);
    SINGLET = ulHashPattern & 0xFF;
    Quadruplet2nd = SINGLET<<8;
    Quadruplet3rd = SINGLET<<16;
    Quadruplet4th = SINGLET<<24;

    for ( ;; )
    {
            AdvanceHopperGrass = 0;
            ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
```

```
//         count = countSTATIC;
//         while ( count && *(char *)(pbPattern+1+(countSTATIC-count)) == *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) {
//                if ( countSTATIC==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-cbPattern+1+(countSTATIC-count)) ) AdvanceHopperGrass++;
//               count--;
//         }
         count = cbPattern-1;
         while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                 if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
              count--;
         }
         if ( count == 0) return((pbTarget-cbPattern));
        } else { // The goal here: to avoid memory accesses by stressing the registers.
    if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
         AdvanceHopperGrass++;
         if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
              AdvanceHopperGrass++;
              if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
         }
    }
           }

           AdvanceHopperGrass++;

           pbTarget = pbTarget + AdvanceHopperGrass;
        if (pbTarget > pbTargetMax)
            return(NULL);
    }
} else { //if (cbTarget<961)
        //countSTATIC = cbPattern-2; //r.6+
        //countSTATIC = cbPattern-2-3;
        //countSTATIC = cbPattern-2-2; // r.6+++ I suppose that the awful degradation comes from 2bytes more (from either 'if (countSTATIC<0) countSTATIC=0;' or 'count >0' fixes) which make the function unfittable in code cache lines?!
        //countSTATIC = cbPattern-2-3; // r.7- At last no recompared bytes in-between chars
        countSTATIC = cbPattern-2-2; // r.7
        ulHashPattern = *(unsigned long *)(pbPattern);

        //chPTR=(unsigned char *)&chchchch+3;
// Next line fixes the BUG from r.6++: but with awful speed degradation! So the bug is fixed in the definitions by setting 'countSTATIC = cbPattern-2-2;', bug appears only for patterns with lengths of 4, The setback is one unnecessary comparison for
5bytes patterns, stupidly such setback exists (from before) for 4bytes as well.
//if (countSTATIC<0) countSTATIC=0;
    /* Preprocessing */
//Below 2 lines are global already:
    //for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
    //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;

    /* Searching */
    //lastch=pbPattern[cbPattern-1];
    //firstch=pbPattern[0];
    i=0;
    while (i <= cbTarget-cbPattern) {
       //ch=pbTarget[i+cbPattern-1];
       //ch=pbTarget[i];
          //if ( pbTarget[i] == pbPattern[0] && *(unsigned long *)&pbTarget[i+cbPattern-1-3] == ulHashPattern) // No problema here since we have 4[+] long pattern here. Overlapping (1 byte recompared) when length=4, grmbl.
          if ( *(unsigned long *)&pbTarget[i] == ulHashPattern ) // The lesson I learned from r.7- now applied in r.7: instead of extracting 'ch' having higher address now the lower address is extracted first in order (hopefully, the test confirms it)
the next 32bytes (including 'ch') to be cached i.e. to comparison part is faster.
             {
         count = countSTATIC;
         while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e. no need of comparing in-between chars.
              count--;
         }
         if ( count == 0) return(pbTarget+i);
                }
       i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
//GlobalI++;
    }
    return(NULL);
} //if (cbTarget<961)
} //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm ]
```

```
// Revision: 2, 2012-Jan-30, the main disadvantage: the preprocessing overhead.
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
//
char * Railgun_Quadruplet_7Gulliver_1 (char * pbTarget, char * pbPattern, unsigned long cbTarget, unsigned long cbPattern)
{
	char * pbTargetMax = pbTarget + cbTarget;
	register unsigned long ulHashPattern;
	register unsigned long ulHashTarget;
	signed long count;
	signed long countSTATIC;

	unsigned char SINGLET;
	unsigned long Quadruplet2nd;
	unsigned long Quadruplet3rd;
	unsigned long Quadruplet4th;

	unsigned long  AdvanceHopperGrass;

	long i; //BMH needed
	int a, j;
//Global are next 2 lines already:
	//unsigned int bm_bc[256]; //BMH needed
	//unsigned int bm_bc2nd[256]; //BMS needed
//Global is next line already:
	//unsigned char bm_Horspool_Order2[256*256]; //BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
	unsigned long Gulliver; // or unsigned char or unsigned short

	if (cbPattern > cbTarget)
		return(NULL);

	if ( cbPattern<4) {
		pbTarget = pbTarget+cbPattern;
		ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));

		if ( cbPattern==3) {
			for ( ;; ) {
				if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
					if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
				}
				if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
				pbTarget++;
				if (pbTarget > pbTargetMax)
					return(NULL);
			}
		} else {
		}
		for ( ;; ) {
			if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
				return((pbTarget-2));
			if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
			pbTarget++;
			if (pbTarget > pbTargetMax)
				return(NULL);
		}
	} else { //if ( cbPattern<4)
		if (cbTarget<961) { // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.

			pbTarget = pbTarget+cbPattern;
			ulHashPattern = *(unsigned long *)(pbPattern);

			SINGLET = ulHashPattern & 0xFF;
			Quadruplet2nd = SINGLET<<8;
			Quadruplet3rd = SINGLET<<16;
			Quadruplet4th = SINGLET<<24;

			for ( ;; ) {
				AdvanceHopperGrass = 0;
```

```
                                        ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

                                    if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
                                                count = cbPattern-1;
                                                while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                                                            if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
                                                            count--;
                                                }
                                                if ( count == 0) return((pbTarget-cbPattern));
                                    } else { // The goal here: to avoid memory accesses by stressing the registers.
                                                if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                                            AdvanceHopperGrass++;
                                                            if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                                        AdvanceHopperGrass++;
                                                                        if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                                            }
                                                }
                                        }

                                        AdvanceHopperGrass++;

                                        pbTarget = pbTarget + AdvanceHopperGrass;
                                        if (pbTarget > pbTargetMax)
                                                return(NULL);
                                }

                        } else { //if (cbTarget<961)
                                countSTATIC = cbPattern-2-2;

//Global are next 3 lines already:
                                //for (a=0; a < 256; a++) {bm_bc[a]=cbPattern; bm_bc2nd[a]=cbPattern+1;}
                                //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
                                //for (j=0; j < cbPattern; j++) bm_bc2nd[pbPattern[j]]=cbPattern-j;

                                // Elsiane r.2  [
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // 'memset' if not optimized

                                // alfalfa 7 long 6 BBs (al lf fa al lf fa) 3 distinct BBs (al lf fa)
                                // fast 4 0-1-2 fa as st
//Global is next line already:
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed

                                // Elsiane r.2 ]

                                ulHashPattern = *(unsigned long *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes

                                i=0;
            if ( cbPattern>10) { // r.2
                                while (i <= cbTarget-cbPattern-1) { // -1 because Sunday is used
                                        Gulliver = bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]];

                                        if ( Gulliver == cbPattern-2 ) { // CASE #1: means the pair (char order 2) is found
                                                if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                            count = countSTATIC; // Last two chars already matched, to be fixed with -2
                                                            while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                        count--;
                                                            if ( count == 0) return(pbTarget+i);
                                                }
                                                //i = i + 1; // r.1, obviuosly this is the worst skip so turning to 'SunHorse': lines below
// r.2 [
if ( bm_bc[(unsigned char)pbTarget[i+cbPattern-1]] < bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]] )
        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
else
        i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
// r.2 ]
                                        } else if ( Gulliver == cbPattern-1 ) // CASE #2: means the pair (char order 2) is not found
                                                i = i + Gulliver; // the pair is not found, skip the whole pattern and fall back (Order-1) chars i.e. one char for Order 2
```

```
                                        else
                                                i = i +  cbPattern - Gulliver - 2; // CASE #3: the pair is found and not as suffix i.e. rightmost position

// 32323218 Order 1 Horspool
// fa af fa af fa as st Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// fafafast
//   fafafast +2 Order 1 'a' vs 't'
//   fafafast +2 = (cbPattern-Gulliver-2 = 8-4-2 = 2) Order 2 'fa' vs 'st' i.e. CASE #3

// 76543218 Order 1 Horspool
// lo on ng gp pa ac ce Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// longpace
//   longpace +2 Order 1 'a' vs 'e'
//        longpace +7 = (cbPattern-1 = 8-1 = 7) Order 2 'fa' vs 'ce' i.e. CASE #2

                                        //GlobalI++;
                                }
            } else { // r.2
                                while (i <= cbTarget-cbPattern-1) {
                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                count = countSTATIC;
                                                while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e.
no need of comparing in-between chars.
                                                        count--;
                                                }
                                                if ( count == 0) return(pbTarget+i);
                                        }
                                        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
                                        //GlobalI++;
                                }
            } // r.2

                                if (i == cbTarget-cbPattern) {
                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                count = countSTATIC;
                                                while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                        count--;
                                                if ( count == 0) return(pbTarget+i);
                                        }
                                        //GlobalI++;
                                }

                                return(NULL);
                        } //if (cbTarget<961)
            } //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Sunday-Horspool algorithm ]

// Revision: 2, 2012-Jan-30, the main disadvantage: the preprocessing overhead.
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
//
char * Railgun_Quadruplet_7Gulliver_2 (char * pbTarget, char * pbPattern, unsigned long cbTarget, unsigned long cbPattern)
{
            char * pbTargetMax = pbTarget + cbTarget;
            register unsigned long ulHashPattern;
            register unsigned long ulHashTarget;
            signed long count;
            signed long countSTATIC;

            unsigned char SINGLET;
            unsigned long Quadruplet2nd;
            unsigned long Quadruplet3rd;
            unsigned long Quadruplet4th;

            unsigned long  AdvanceHopperGrass;
```

```
	long i; //BMH needed
	int a, j;
//Global are next 2 lines already:
	//unsigned int bm_bc[256]; //BMH needed
	//unsigned int bm_bc2nd[256]; //BMS needed
//Global is next line already:
	//unsigned char bm_Horspool_Order2[256*256]; //BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
	unsigned long Gulliver; // or unsigned char or unsigned short

	if (cbPattern > cbTarget)
		return(NULL);

	if ( cbPattern<4) {
		pbTarget = pbTarget+cbPattern;
		ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));

		if ( cbPattern==3) {
			for ( ;; ) {
				if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
					if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
				}
				if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
				pbTarget++;
				if (pbTarget > pbTargetMax)
					return(NULL);
			}
		} else {
		}
		for ( ;; ) {
			if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
				return((pbTarget-2));
			if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
			pbTarget++;
			if (pbTarget > pbTargetMax)
				return(NULL);
		}
	} else { //if ( cbPattern<4)
		if (cbTarget<961) { // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.

			pbTarget = pbTarget+cbPattern;
			ulHashPattern = *(unsigned long *)(pbPattern);

			SINGLET = ulHashPattern & 0xFF;
			Quadruplet2nd = SINGLET<<8;
			Quadruplet3rd = SINGLET<<16;
			Quadruplet4th = SINGLET<<24;

			for ( ;; ) {
				AdvanceHopperGrass = 0;
				ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

			if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
				count = cbPattern-1;
				while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
					if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
					count--;
				}
				if ( count == 0) return((pbTarget-cbPattern));
			} else { // The goal here: to avoid memory accesses by stressing the registers.
				if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
					AdvanceHopperGrass++;
					if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
						AdvanceHopperGrass++;
						if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
					}
				}
			}
```

```
                                        AdvanceHopperGrass++;

                                        pbTarget = pbTarget + AdvanceHopperGrass;
                                        if (pbTarget > pbTargetMax)
                                                return(NULL);
                                }

                        } else { //if (cbTarget<961)
                                countSTATIC = cbPattern-2-2;

//Global are next 3 lines already:
                                //for (a=0; a < 256; a++) {bm_bc[a]=cbPattern; bm_bc2nd[a]=cbPattern+1;}
                                //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
                                //for (j=0; j < cbPattern; j++) bm_bc2nd[pbPattern[j]]=cbPattern-j;

                                // Elsiane r.2  [
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // 'memset' if not optimized

                                // alfalfa 7 long 6 BBs (al lf fa al lf fa) 3 distinct BBs (al lf fa)
                                // fast 4 0-1-2 fa as st
//Global is next line already:
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed

                                // Elsiane r.2 ]

                                ulHashPattern = *(unsigned long *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes

                                i=0;
           if ( cbPattern>10) { // r.2
                                while (i <= cbTarget-cbPattern-1) { // -1 because Sunday is used
                                        Gulliver = bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]];

                                        if ( Gulliver == cbPattern-2 ) { // CASE #1: means the pair (char order 2) is found
                                                if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                        count = countSTATIC; // Last two chars already matched, to be fixed with -2
                                                        while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                count--;
                                                        if ( count == 0) return(pbTarget+i);
                                                }
                                                //i = i + 1; // r.1, obviuosly this is the worst skip so turning to 'SunHorse': lines below
// r.2 [
if ( bm_bc[(unsigned char)pbTarget[i+cbPattern-1]] < bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]] )
        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
else
        i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
// r.2 ]
                                        } else if ( Gulliver == cbPattern-1 ) // CASE #2: means the pair (char order 2) is not found
                                                i = i + Gulliver; // the pair is not found, skip the whole pattern and fall back (Order-1) chars i.e. one char for Order 2
                                        else
                                                i = i +  cbPattern - Gulliver - 2; // CASE #3: the pair is found and not as suffix i.e. rightmost position

// 32323218 Order 1 Horspool
// fa af fa af fa as st Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// fafafast
//   fafafast +2 Order 1 'a' vs 't'
//   fafafast +2 = (cbPattern-Gulliver-2 = 8-4-2 = 2) Order 2 'fa' vs 'st' i.e. CASE #3

// 76543218 Order 1 Horspool
// lo on ng gp pa ac ce Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// longpace
//   longpace +2 Order 1 'a' vs 'e'
//        longpace +7 = (cbPattern-1 = 8-1 = 7) Order 2 'fa' vs 'ce' i.e. CASE #2
```

```
                                        //GlobalI++;
                                }
            } else { // r.2
                                while (i <= cbTarget-cbPattern-1) {
                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                count = countSTATIC;
                                                while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e.
no need of comparing in-between chars.
                                                        count--;
                                                }
                                                if ( count == 0) return(pbTarget+i);
                                        }
                                        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
                                        //GlobalI++;
                                }
            } // r.2

                                if (i == cbTarget-cbPattern) {
                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                count = countSTATIC;
                                                while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                        count--;
                                                if ( count == 0) return(pbTarget+i);
                                        }
                                        //GlobalI++;
                                }

                                return(NULL);
                        } //if (cbTarget<961)
            } //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Sunday-Horspool algorithm ]

// Revision: 2, 2012-Jan-30, the main disadvantage: the preprocessing overhead.
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
//
char * Railgun_Quadruplet_7Gulliver_3 (char * pbTarget, char * pbPattern, unsigned long cbTarget, unsigned long cbPattern)
{
            char * pbTargetMax = pbTarget + cbTarget;
            register unsigned long ulHashPattern;
            register unsigned long ulHashTarget;
            signed long count;
            signed long countSTATIC;

            unsigned char SINGLET;
            unsigned long Quadruplet2nd;
            unsigned long Quadruplet3rd;
            unsigned long Quadruplet4th;

            unsigned long  AdvanceHopperGrass;

            long i; //BMH needed
            int a, j;
//Global are next 2 lines already:
            //unsigned int bm_bc[256]; //BMH needed
            //unsigned int bm_bc2nd[256]; //BMS needed
//Global is next line already:
            //unsigned char bm_Horspool_Order2[256*256]; //BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
            unsigned long Gulliver; // or unsigned char or unsigned short

            if (cbPattern > cbTarget)
                        return(NULL);

            if ( cbPattern<4) {
                        pbTarget = pbTarget+cbPattern;
                        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));

                        if ( cbPattern==3) {
                                for ( ;; ) {
```

```
                                        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
                                                if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
                                        }
                                        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
                                        pbTarget++;
                                        if (pbTarget > pbTargetMax)
                                                return(NULL);
                                }
                        } else {
                        }
                        for ( ;; ) {
                                if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
                                        return((pbTarget-2));
                                if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
                                pbTarget++;
                                if (pbTarget > pbTargetMax)
                                        return(NULL);
                        }
                } else { //if ( cbPattern<4)
                        if (cbTarget<961) { // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.

                                pbTarget = pbTarget+cbPattern;
                                ulHashPattern = *(unsigned long *)(pbPattern);

                                SINGLET = ulHashPattern & 0xFF;
                                Quadruplet2nd = SINGLET<<8;
                                Quadruplet3rd = SINGLET<<16;
                                Quadruplet4th = SINGLET<<24;

                                for ( ;; ) {
                                        AdvanceHopperGrass = 0;
                                        ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

                                        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
                                                count = cbPattern-1;
                                                while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                                                        if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
                                                        count--;
                                                }
                                                if ( count == 0) return((pbTarget-cbPattern));
                                        } else { // The goal here: to avoid memory accesses by stressing the registers.
                                                if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                                        AdvanceHopperGrass++;
                                                        if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                                AdvanceHopperGrass++;
                                                                if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                                        }
                                                }
                                        }

                                        AdvanceHopperGrass++;

                                        pbTarget = pbTarget + AdvanceHopperGrass;
                                        if (pbTarget > pbTargetMax)
                                                return(NULL);
                                }

                        } else { //if (cbTarget<961)
                                countSTATIC = cbPattern-2-2;

//Global are next 3 lines already:
                                //for (a=0; a < 256; a++) {bm_bc[a]=cbPattern; bm_bc2nd[a]=cbPattern+1;}
                                //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
                                //for (j=0; j < cbPattern; j++) bm_bc2nd[pbPattern[j]]=cbPattern-j;

                                // Elsiane r.2  [
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // 'memset' if not optimized
```

```
                                // alfalfa 7 long 6 BBs (al lf fa al lf fa) 3 distinct BBs (al lf fa)
                                // fast 4 0-1-2 fa as st
//Global is next line already:
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed

                                // Elsiane r.2 ]

                                ulHashPattern = *(unsigned long *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes

                                i=0;
            if ( cbPattern>10) { // r.2
                                while (i <= cbTarget-cbPattern-1) { // -1 because Sunday is used
                                            Gulliver = bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]];

                                            if ( Gulliver == cbPattern-2 ) { // CASE #1: means the pair (char order 2) is found
                                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                                    count = countSTATIC; // Last two chars already matched, to be fixed with -2
                                                                    while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                                count--;
                                                                    if ( count == 0) return(pbTarget+i);
                                                        }
                                                        //i = i + 1; // r.1, obviuosly this is the worst skip so turning to 'SunHorse': lines below
// r.2 [
if ( bm_bc[(unsigned char)pbTarget[i+cbPattern-1]] < bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]] )
        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
else
        i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
// r.2 ]
                                            } else if ( Gulliver == cbPattern-1 ) // CASE #2: means the pair (char order 2) is not found
                                                        i = i + Gulliver; // the pair is not found, skip the whole pattern and fall back (Order-1) chars i.e. one char for Order 2
                                            else
                                                        i = i +  cbPattern - Gulliver - 2; // CASE #3: the pair is found and not as suffix i.e. rightmost position

// 32323218 Order 1 Horspool
// fa af fa af fa as st Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// fafafast
//   fafafast +2 Order 1 'a' vs 't'
//   fafafast +2 = (cbPattern-Gulliver-2 = 8-4-2 = 2) Order 2 'fa' vs 'st' i.e. CASE #3

// 76543218 Order 1 Horspool
// lo on ng gp pa ac ce Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// longpace
//   longpace +2 Order 1 'a' vs 'e'
//        longpace +7 = (cbPattern-1 = 8-1 = 7) Order 2 'fa' vs 'ce' i.e. CASE #2

                                            //GlobalI++;
                                }
            } else { // r.2
                                while (i <= cbTarget-cbPattern-1) {
                                            if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                        count = countSTATIC;
                                                        while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e.
no need of comparing in-between chars.
                                                                    count--;
                                                        }
                                                        if ( count == 0) return(pbTarget+i);
                                            }
                                            i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
                                            //GlobalI++;
                                }
            } // r.2

                                if (i == cbTarget-cbPattern) {
                                            if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
```

```
                                                count = countSTATIC;
                                                while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                        count--;
                                                if ( count == 0) return(pbTarget+i);
                                        }
                                        //GlobalI++;
                                }

                                return(NULL);
                        } //if (cbTarget<961)
            } //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Sunday-Horspool algorithm ]

// Revision: 2, 2012-Jan-30, the main disadvantage: the preprocessing overhead.
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
//
char * Railgun_Quadruplet_7Gulliver_4 (char * pbTarget, char * pbPattern, unsigned long cbTarget, unsigned long cbPattern)
{
            char * pbTargetMax = pbTarget + cbTarget;
            register unsigned long ulHashPattern;
            register unsigned long ulHashTarget;
            signed long count;
            signed long countSTATIC;

            unsigned char SINGLET;
            unsigned long Quadruplet2nd;
            unsigned long Quadruplet3rd;
            unsigned long Quadruplet4th;

            unsigned long  AdvanceHopperGrass;

            long i; //BMH needed
            int a, j;
//Global are next 2 lines already:
            //unsigned int bm_bc[256]; //BMH needed
            //unsigned int bm_bc2nd[256]; //BMS needed
//Global is next line already:
            //unsigned char bm_Horspool_Order2[256*256]; //BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
            unsigned long Gulliver; // or unsigned char or unsigned short

            if (cbPattern > cbTarget)
                        return(NULL);

            if ( cbPattern<4) {
                        pbTarget = pbTarget+cbPattern;
                        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));

                        if ( cbPattern==3) {
                                    for ( ;; ) {
                                                if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
                                                            if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
                                                }
                                                if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
                                                pbTarget++;
                                                if (pbTarget > pbTargetMax)
                                                            return(NULL);
                                    }
                        } else {
                        }
                        for ( ;; ) {
                                    if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
                                                return((pbTarget-2));
                                    if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
                                    pbTarget++;
                                    if (pbTarget > pbTargetMax)
                                                return(NULL);
                        }
            } else { //if ( cbPattern<4)
```

```
                    if (cbTarget<961) { // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.

                              pbTarget = pbTarget+cbPattern;
                              ulHashPattern = *(unsigned long *)(pbPattern);

                              SINGLET = ulHashPattern & 0xFF;
                              Quadruplet2nd = SINGLET<<8;
                              Quadruplet3rd = SINGLET<<16;
                              Quadruplet4th = SINGLET<<24;

                              for ( ;; ) {
                                        AdvanceHopperGrass = 0;
                                        ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

                                      if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
                                                  count = cbPattern-1;
                                                  while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                                                            if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
                                                            count--;
                                                  }
                                                  if ( count == 0) return((pbTarget-cbPattern));
                                      } else { // The goal here: to avoid memory accesses by stressing the registers.
                                                  if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                                            AdvanceHopperGrass++;
                                                            if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                                      AdvanceHopperGrass++;
                                                                      if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                                            }
                                                  }
                                        }

                                        AdvanceHopperGrass++;

                                        pbTarget = pbTarget + AdvanceHopperGrass;
                                        if (pbTarget > pbTargetMax)
                                                  return(NULL);
                              }

                    } else { //if (cbTarget<961)
                              countSTATIC = cbPattern-2-2;

//Global are next 3 lines already:
                              //for (a=0; a < 256; a++) {bm_bc[a]=cbPattern; bm_bc2nd[a]=cbPattern+1;}
                              //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
                              //for (j=0; j < cbPattern; j++) bm_bc2nd[pbPattern[j]]=cbPattern-j;

                              // Elsiane r.2  [
//Global is next line already:
                              //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // 'memset' if not optimized

                              // alfalfa 7 long 6 BBs (al lf fa al lf fa) 3 distinct BBs (al lf fa)
                              // fast 4 0-1-2 fa as st
//Global is next line already:
                              //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed

                              // Elsiane r.2 ]

                              ulHashPattern = *(unsigned long *)(pbPattern); // First four bytes
                              //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes

                              i=0;
          if ( cbPattern>10) { // r.2
                              while (i <= cbTarget-cbPattern-1) { // -1 because Sunday is used
                                        Gulliver = bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]];

                                        if ( Gulliver == cbPattern-2 ) { // CASE #1: means the pair (char order 2) is found
                                                  if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                            count = countSTATIC; // Last two chars already matched, to be fixed with -2
                                                            while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
```

```
                                                                        count--;
                                                                if ( count == 0) return(pbTarget+i);
                                                        }
                                                        //i = i + 1; // r.1, obviuosly this is the worst skip so turning to 'SunHorse': lines below
// r.2 [
if ( bm_bc[(unsigned char)pbTarget[i+cbPattern-1]] < bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]] )
        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
else
        i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
// r.2 ]
                                                } else if ( Gulliver == cbPattern-1 ) // CASE #2: means the pair (char order 2) is not found
                                                        i = i + Gulliver; // the pair is not found, skip the whole pattern and fall back (Order-1) chars i.e. one char for Order 2
                                                else
                                                        i = i +  cbPattern - Gulliver - 2; // CASE #3: the pair is found and not as suffix i.e. rightmost position

// 32323218 Order 1 Horspool
// fa af fa af fa as st Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// fafafast
//   fafafast +2 Order 1 'a' vs 't'
//   fafafast +2 = (cbPattern-Gulliver-2 = 8-4-2 = 2) Order 2 'fa' vs 'st' i.e. CASE #3

// 76543218 Order 1 Horspool
// lo on ng gp pa ac ce Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// longpace
//   longpace +2 Order 1 'a' vs 'e'
//        longpace +7 = (cbPattern-1 = 8-1 = 7) Order 2 'fa' vs 'ce' i.e. CASE #2

                                                //GlobalI++;
                                        }
                                } else { // r.2
                                        while (i <= cbTarget-cbPattern-1) {
                                                if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                        count = countSTATIC;
                                                        while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e.
no need of comparing in-between chars.
                                                                count--;
                                                        }
                                                        if ( count == 0) return(pbTarget+i);
                                                }
                                                i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
                                                //GlobalI++;
                                        }
                                } // r.2

                                        if (i == cbTarget-cbPattern) {
                                                if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                        count = countSTATIC;
                                                        while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                count--;
                                                        if ( count == 0) return(pbTarget+i);
                                                }
                                                //GlobalI++;
                                        }

                                        return(NULL);
                                } //if (cbTarget<961)
                        } //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Sunday-Horspool algorithm ]

// Revision: 2, 2012-Jan-30, the main disadvantage: the preprocessing overhead.
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
//
char * Railgun_Quadruplet_7Gulliver_5 (char * pbTarget, char * pbPattern, unsigned long cbTarget, unsigned long cbPattern)
{
```

```
	char * pbTargetMax = pbTarget + cbTarget;
	register unsigned long ulHashPattern;
	register unsigned long ulHashTarget;
	signed long count;
	signed long countSTATIC;

	unsigned char SINGLET;
	unsigned long Quadruplet2nd;
	unsigned long Quadruplet3rd;
	unsigned long Quadruplet4th;

	unsigned long  AdvanceHopperGrass;

	long i; //BMH needed
	int a, j;
//Global are next 2 lines already:
	//unsigned int bm_bc[256]; //BMH needed
	//unsigned int bm_bc2nd[256]; //BMS needed
//Global is next line already:
	//unsigned char bm_Horspool_Order2[256*256]; //BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
	unsigned long Gulliver; // or unsigned char or unsigned short

	if (cbPattern > cbTarget)
		return(NULL);

	if ( cbPattern<4) {
		pbTarget = pbTarget+cbPattern;
		ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));

		if ( cbPattern==3) {
			for ( ;; ) {
				if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
					if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
				}
				if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
				pbTarget++;
				if (pbTarget > pbTargetMax)
					return(NULL);
			}
		} else {
		}
		for ( ;; ) {
			if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
				return((pbTarget-2));
			if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
			pbTarget++;
			if (pbTarget > pbTargetMax)
				return(NULL);
		}
	} else { //if ( cbPattern<4)
		if (cbTarget<961) { // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.

			pbTarget = pbTarget+cbPattern;
			ulHashPattern = *(unsigned long *)(pbPattern);

			SINGLET = ulHashPattern & 0xFF;
			Quadruplet2nd = SINGLET<<8;
			Quadruplet3rd = SINGLET<<16;
			Quadruplet4th = SINGLET<<24;

			for ( ;; ) {
				AdvanceHopperGrass = 0;
				ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

				if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
					count = cbPattern-1;
					while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
						if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
						count--;
```

```
                                        }
                                        if ( count == 0) return((pbTarget-cbPattern));
                                } else { // The goal here: to avoid memory accesses by stressing the registers.
                                        if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                                AdvanceHopperGrass++;
                                                if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                        AdvanceHopperGrass++;
                                                        if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                                }
                                        }
                                }

                                AdvanceHopperGrass++;

                                pbTarget = pbTarget + AdvanceHopperGrass;
                                if (pbTarget > pbTargetMax)
                                        return(NULL);
                        }

                } else { //if (cbTarget<961)
                        countSTATIC = cbPattern-2-2;

//Global are next 3 lines already:
                        //for (a=0; a < 256; a++) {bm_bc[a]=cbPattern; bm_bc2nd[a]=cbPattern+1;}
                        //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
                        //for (j=0; j < cbPattern; j++) bm_bc2nd[pbPattern[j]]=cbPattern-j;

                        // Elsiane r.2  [
//Global is next line already:
                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // 'memset' if not optimized

                        // alfalfa 7 long 6 BBs (al lf fa al lf fa) 3 distinct BBs (al lf fa)
                        // fast 4 0-1-2 fa as st
//Global is next line already:
                        //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed

                        // Elsiane r.2 ]

                        ulHashPattern = *(unsigned long *)(pbPattern); // First four bytes
                        //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes

                        i=0;
          if ( cbPattern>10) { // r.2
                        while (i <= cbTarget-cbPattern-1) { // -1 because Sunday is used
                                Gulliver = bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]];

                                if ( Gulliver == cbPattern-2 ) { // CASE #1: means the pair (char order 2) is found
                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                count = countSTATIC; // Last two chars already matched, to be fixed with -2
                                                while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                        count--;
                                                if ( count == 0) return(pbTarget+i);
                                        }
                                        //i = i + 1; // r.1, obviuosly this is the worst skip so turning to 'SunHorse': lines below
// r.2 [
if ( bm_bc[(unsigned char)pbTarget[i+cbPattern-1]] < bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]] )
        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
else
        i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
// r.2 ]
                                } else if ( Gulliver == cbPattern-1 ) // CASE #2: means the pair (char order 2) is not found
                                        i = i + Gulliver; // the pair is not found, skip the whole pattern and fall back (Order-1) chars i.e. one char for Order 2
                                else
                                        i = i +  cbPattern - Gulliver - 2; // CASE #3: the pair is found and not as suffix i.e. rightmost position

// 32323218 Order 1 Horspool
// fa af fa af fa as st Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
```

```
// fafafast
//   fafafast +2 Order 1 'a' vs 't'
//   fafafast +2 = (cbPattern-Gulliver-2 = 8-4-2 = 2) Order 2 'fa' vs 'st' i.e. CASE #3

// 76543218 Order 1 Horspool
// lo on ng gp pa ac ce Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// longpace
//   longpace +2 Order 1 'a' vs 'e'
//        longpace +7 = (cbPattern-1 = 8-1 = 7) Order 2 'fa' vs 'ce' i.e. CASE #2

                                        //GlobalI++;
                                }
            } else { // r.2
                                while (i <= cbTarget-cbPattern-1) {
                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                count = countSTATIC;
                                                while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e.
no need of comparing in-between chars.
                                                        count--;
                                                }
                                                if ( count == 0) return(pbTarget+i);
                                        }
                                        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
                                        //GlobalI++;
                                }
            } // r.2

                                if (i == cbTarget-cbPattern) {
                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                count = countSTATIC;
                                                while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                        count--;
                                                if ( count == 0) return(pbTarget+i);
                                        }
                                        //GlobalI++;
                                }

                                return(NULL);
                        } //if (cbTarget<961)
            } //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Sunday-Horspool algorithm ]

// Revision: 2, 2012-Jan-30, the main disadvantage: the preprocessing overhead.
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
//
char * Railgun_Quadruplet_7Gulliver_6 (char * pbTarget, char * pbPattern, unsigned long cbTarget, unsigned long cbPattern)
{
            char * pbTargetMax = pbTarget + cbTarget;
            register unsigned long ulHashPattern;
            register unsigned long ulHashTarget;
            signed long count;
            signed long countSTATIC;

            unsigned char SINGLET;
            unsigned long Quadruplet2nd;
            unsigned long Quadruplet3rd;
            unsigned long Quadruplet4th;

            unsigned long  AdvanceHopperGrass;

            long i; //BMH needed
            int a, j;
//Global are next 2 lines already:
            //unsigned int bm_bc[256]; //BMH needed
            //unsigned int bm_bc2nd[256]; //BMS needed
//Global is next line already:
```

```
//unsigned char bm_Horspool_Order2[256*256]; //BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
unsigned long Gulliver; // or unsigned char or unsigned short

if (cbPattern > cbTarget)
          return(NULL);

if ( cbPattern<4) {
          pbTarget = pbTarget+cbPattern;
          ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));

          if ( cbPattern==3) {
                    for ( ;; ) {
                              if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
                                        if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
                              }
                              if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
                              pbTarget++;
                              if (pbTarget > pbTargetMax)
                                        return(NULL);
                    }
          } else {
          }
          for ( ;; ) {
                    if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
                              return((pbTarget-2));
                    if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
                    pbTarget++;
                    if (pbTarget > pbTargetMax)
                              return(NULL);
          }
} else { //if ( cbPattern<4)
          if (cbTarget<961) { // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.

                    pbTarget = pbTarget+cbPattern;
                    ulHashPattern = *(unsigned long *)(pbPattern);

                    SINGLET = ulHashPattern & 0xFF;
                    Quadruplet2nd = SINGLET<<8;
                    Quadruplet3rd = SINGLET<<16;
                    Quadruplet4th = SINGLET<<24;

                    for ( ;; ) {
                              AdvanceHopperGrass = 0;
                              ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

                          if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
                                        count = cbPattern-1;
                                        while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                                                  if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
                                                  count--;
                                        }
                                        if ( count == 0) return((pbTarget-cbPattern));
                          } else { // The goal here: to avoid memory accesses by stressing the registers.
                                        if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                                  AdvanceHopperGrass++;
                                                  if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                            AdvanceHopperGrass++;
                                                            if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                                  }
                                        }
                              }

                              AdvanceHopperGrass++;

                              pbTarget = pbTarget + AdvanceHopperGrass;
                              if (pbTarget > pbTargetMax)
                                        return(NULL);
                    }
```

```
                    } else { //if (cbTarget<961)
                                        countSTATIC = cbPattern-2-2;

//Global are next 3 lines already:
                                        //for (a=0; a < 256; a++) {bm_bc[a]=cbPattern; bm_bc2nd[a]=cbPattern+1;}
                                        //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
                                        //for (j=0; j < cbPattern; j++) bm_bc2nd[pbPattern[j]]=cbPattern-j;

                                        // Elsiane r.2  [
//Global is next line already:
                                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // 'memset' if not optimized

                                        // alfalfa 7 long 6 BBs (al lf fa al lf fa) 3 distinct BBs (al lf fa)
                                        // fast 4 0-1-2 fa as st
//Global is next line already:
                                        //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed

                                        // Elsiane r.2 ]

                                        ulHashPattern = *(unsigned long *)(pbPattern); // First four bytes
                                        //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes

                                        i=0;
            if ( cbPattern>10) { // r.2
                                        while (i <= cbTarget-cbPattern-1) { // -1 because Sunday is used
                                                    Gulliver = bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]];

                                                    if ( Gulliver == cbPattern-2 ) { // CASE #1: means the pair (char order 2) is found
                                                                if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                                            count = countSTATIC; // Last two chars already matched, to be fixed with -2
                                                                            while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                                        count--;
                                                                            if ( count == 0) return(pbTarget+i);
                                                                }
                                                                //i = i + 1; // r.1, obviuosly this is the worst skip so turning to 'SunHorse': lines below
// r.2 [
if ( bm_bc[(unsigned char)pbTarget[i+cbPattern-1]] < bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]] )
        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
else
        i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
// r.2 ]
                                                    } else if ( Gulliver == cbPattern-1 ) // CASE #2: means the pair (char order 2) is not found
                                                                i = i + Gulliver; // the pair is not found, skip the whole pattern and fall back (Order-1) chars i.e. one char for Order 2
                                                    else
                                                                i = i +  cbPattern - Gulliver - 2; // CASE #3: the pair is found and not as suffix i.e. rightmost position

// 32323218 Order 1 Horspool
// fa af fa af fa as st Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// fafafast
//   fafafast +2 Order 1 'a' vs 't'
//   fafafast +2 = (cbPattern-Gulliver-2 = 8-4-2 = 2) Order 2 'fa' vs 'st' i.e. CASE #3

// 76543218 Order 1 Horspool
// lo on ng gp pa ac ce Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// longpace
//   longpace +2 Order 1 'a' vs 'e'
//        longpace +7 = (cbPattern-1 = 8-1 = 7) Order 2 'fa' vs 'ce' i.e. CASE #2

                                                    //GlobalI++;
                                        }
            } else { // r.2
                                        while (i <= cbTarget-cbPattern-1) {
                                                    if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                                count = countSTATIC;
                                                                while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e.
```

```
no need of comparing in-between chars.
                                                                        count--;
                                                                }
                                                                if ( count == 0) return(pbTarget+i);
                                                        }
                                                        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
                                                        //GlobalI++;
                                                }
                                } // r.2

                                                if (i == cbTarget-cbPattern) {
                                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                                count = countSTATIC;
                                                                while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                        count--;
                                                                if ( count == 0) return(pbTarget+i);
                                                        }
                                                        //GlobalI++;
                                                }

                                                return(NULL);
                                        } //if (cbTarget<961)
                                } //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Sunday-Horspool algorithm ]

// Revision: 2, 2012-Jan-30, the main disadvantage: the preprocessing overhead.
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
//
char * Railgun_Quadruplet_7Gulliver_7 (char * pbTarget, char * pbPattern, unsigned long cbTarget, unsigned long cbPattern)
{
                                char * pbTargetMax = pbTarget + cbTarget;
                                register unsigned long ulHashPattern;
                                register unsigned long ulHashTarget;
                                signed long count;
                                signed long countSTATIC;

                                unsigned char SINGLET;
                                unsigned long Quadruplet2nd;
                                unsigned long Quadruplet3rd;
                                unsigned long Quadruplet4th;

                                unsigned long  AdvanceHopperGrass;

                                long i; //BMH needed
                                int a, j;
//Global are next 2 lines already:
                                //unsigned int bm_bc[256]; //BMH needed
                                //unsigned int bm_bc2nd[256]; //BMS needed
//Global is next line already:
                                //unsigned char bm_Horspool_Order2[256*256]; //BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
                                unsigned long Gulliver; // or unsigned char or unsigned short

                                if (cbPattern > cbTarget)
                                        return(NULL);

                                if ( cbPattern<4) {
                                        pbTarget = pbTarget+cbPattern;
                                        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));

                                        if ( cbPattern==3) {
                                                for ( ;; ) {
                                                        if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
                                                                if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
                                                        }
                                                        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
                                                        pbTarget++;
                                                        if (pbTarget > pbTargetMax)
                                                                return(NULL);
```

```
                                }
                        } else {
                        }
                        for ( ;; ) {
                                if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
                                        return((pbTarget-2));
                                if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
                                pbTarget++;
                                if (pbTarget > pbTargetMax)
                                        return(NULL);
                        }
                } else { //if ( cbPattern<4)
                        if (cbTarget<961) { // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.

                                pbTarget = pbTarget+cbPattern;
                                ulHashPattern = *(unsigned long *)(pbPattern);

                                SINGLET = ulHashPattern & 0xFF;
                                Quadruplet2nd = SINGLET<<8;
                                Quadruplet3rd = SINGLET<<16;
                                Quadruplet4th = SINGLET<<24;

                                for ( ;; ) {
                                        AdvanceHopperGrass = 0;
                                        ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

                                        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
                                                count = cbPattern-1;
                                                while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                                                        if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
                                                        count--;
                                                }
                                                if ( count == 0) return((pbTarget-cbPattern));
                                        } else { // The goal here: to avoid memory accesses by stressing the registers.
                                                if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                                        AdvanceHopperGrass++;
                                                        if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                                AdvanceHopperGrass++;
                                                                if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                                        }
                                                }
                                        }

                                        AdvanceHopperGrass++;

                                        pbTarget = pbTarget + AdvanceHopperGrass;
                                        if (pbTarget > pbTargetMax)
                                                return(NULL);
                                }

                        } else { //if (cbTarget<961)
                                countSTATIC = cbPattern-2-2;

//Global are next 3 lines already:
                                //for (a=0; a < 256; a++) {bm_bc[a]=cbPattern; bm_bc2nd[a]=cbPattern+1;}
                                //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
                                //for (j=0; j < cbPattern; j++) bm_bc2nd[pbPattern[j]]=cbPattern-j;

                                // Elsiane r.2  [
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // 'memset' if not optimized

                                // alfalfa 7 long 6 BBs (al lf fa al lf fa) 3 distinct BBs (al lf fa)
                                // fast 4 0-1-2 fa as st
//Global is next line already:
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed

                                // Elsiane r.2 ]
```

```
                                ulHashPattern = *(unsigned long *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes

                                i=0;
            if ( cbPattern>10) { // r.2
                                while (i <= cbTarget-cbPattern-1) { // -1 because Sunday is used
                                            Gulliver = bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]];

                                            if ( Gulliver == cbPattern-2 ) { // CASE #1: means the pair (char order 2) is found
                                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                                    count = countSTATIC; // Last two chars already matched, to be fixed with -2
                                                                    while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                                count--;
                                                                    if ( count == 0) return(pbTarget+i);
                                                        }
                                                        //i = i + 1; // r.1, obviuosly this is the worst skip so turning to 'SunHorse': lines below
// r.2 [
if ( bm_bc[(unsigned char)pbTarget[i+cbPattern-1]] < bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]] )
         i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
else
         i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
// r.2 ]
                                            } else if ( Gulliver == cbPattern-1 ) // CASE #2: means the pair (char order 2) is not found
                                                        i = i + Gulliver; // the pair is not found, skip the whole pattern and fall back (Order-1) chars i.e. one char for Order 2
                                            else
                                                        i = i +  cbPattern - Gulliver - 2; // CASE #3: the pair is found and not as suffix i.e. rightmost position

// 32323218 Order 1 Horspool
// fa af fa af fa as st Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// fafafast
//   fafafast +2 Order 1 'a' vs 't'
//   fafafast +2 = (cbPattern-Gulliver-2 = 8-4-2 = 2) Order 2 'fa' vs 'st' i.e. CASE #3

// 76543218 Order 1 Horspool
// lo on ng gp pa ac ce Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// longpace
//   longpace +2 Order 1 'a' vs 'e'
//        longpace +7 = (cbPattern-1 = 8-1 = 7) Order 2 'fa' vs 'ce' i.e. CASE #2

                                            //GlobalI++;
                                }
            } else { // r.2
                                while (i <= cbTarget-cbPattern-1) {
                                            if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                        count = countSTATIC;
                                                        while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e.
no need of comparing in-between chars.
                                                                    count--;
                                                        }
                                                        if ( count == 0) return(pbTarget+i);
                                            }
                                            i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
                                            //GlobalI++;
                                }
            } // r.2

                                if (i == cbTarget-cbPattern) {
                                            if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                        count = countSTATIC;
                                                        while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                    count--;
                                                        if ( count == 0) return(pbTarget+i);
                                            }
                                            //GlobalI++;
                                }
```

```
                                return(NULL);
                        } //if (cbTarget<961)
            } //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Sunday-Horspool algorithm ]

// Revision: 2, 2012-Jan-30, the main disadvantage: the preprocessing overhead.
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
//
char * Railgun_Quadruplet_7Gulliver_8 (char * pbTarget, char * pbPattern, unsigned long cbTarget, unsigned long cbPattern)
{
            char * pbTargetMax = pbTarget + cbTarget;
            register unsigned long ulHashPattern;
            register unsigned long ulHashTarget;
            signed long count;
            signed long countSTATIC;

            unsigned char SINGLET;
            unsigned long Quadruplet2nd;
            unsigned long Quadruplet3rd;
            unsigned long Quadruplet4th;

            unsigned long  AdvanceHopperGrass;

            long i; //BMH needed
            int a, j;
//Global are next 2 lines already:
            //unsigned int bm_bc[256]; //BMH needed
            //unsigned int bm_bc2nd[256]; //BMS needed
//Global is next line already:
            //unsigned char bm_Horspool_Order2[256*256]; //BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
            unsigned long Gulliver; // or unsigned char or unsigned short

            if (cbPattern > cbTarget)
                        return(NULL);

            if ( cbPattern<4) {
                        pbTarget = pbTarget+cbPattern;
                        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));

                        if ( cbPattern==3) {
                                    for ( ;; ) {
                                                if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
                                                            if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
                                                }
                                                if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
                                                pbTarget++;
                                                if (pbTarget > pbTargetMax)
                                                            return(NULL);
                                    }
                        } else {
                        }
                        for ( ;; ) {
                                    if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
                                                return((pbTarget-2));
                                    if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
                                    pbTarget++;
                                    if (pbTarget > pbTargetMax)
                                                return(NULL);
                        }
            } else { //if ( cbPattern<4)
                        if (cbTarget<961) { // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.

                                    pbTarget = pbTarget+cbPattern;
                                    ulHashPattern = *(unsigned long *)(pbPattern);

                                    SINGLET = ulHashPattern & 0xFF;
                                    Quadruplet2nd = SINGLET<<8;
```

```
                                Quadruplet3rd = SINGLET<<16;
                                Quadruplet4th = SINGLET<<24;

                                for ( ;; ) {
                                            AdvanceHopperGrass = 0;
                                            ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

                                        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
                                                    count = cbPattern-1;
                                                    while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                                                                if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
                                                                count--;
                                                    }
                                                    if ( count == 0) return((pbTarget-cbPattern));
                                        } else { // The goal here: to avoid memory accesses by stressing the registers.
                                                    if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                                                AdvanceHopperGrass++;
                                                                if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                                            AdvanceHopperGrass++;
                                                                            if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                                                }
                                                    }
                                            }

                                            AdvanceHopperGrass++;

                                            pbTarget = pbTarget + AdvanceHopperGrass;
                                            if (pbTarget > pbTargetMax)
                                                        return(NULL);
                                }

                        } else { //if (cbTarget<961)
                                countSTATIC = cbPattern-2-2;

//Global are next 3 lines already:
                                //for (a=0; a < 256; a++) {bm_bc[a]=cbPattern; bm_bc2nd[a]=cbPattern+1;}
                                //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
                                //for (j=0; j < cbPattern; j++) bm_bc2nd[pbPattern[j]]=cbPattern-j;

                                // Elsiane r.2  [
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // 'memset' if not optimized

                                // alfalfa 7 long 6 BBs (al lf fa al lf fa) 3 distinct BBs (al lf fa)
                                // fast 4 0-1-2 fa as st
//Global is next line already:
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed

                                // Elsiane r.2 ]

                                ulHashPattern = *(unsigned long *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes

                                i=0;
            if ( cbPattern>10) { // r.2
                                while (i <= cbTarget-cbPattern-1) { // -1 because Sunday is used
                                            Gulliver = bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]];

                                            if ( Gulliver == cbPattern-2 ) { // CASE #1: means the pair (char order 2) is found
                                                    if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                                count = countSTATIC; // Last two chars already matched, to be fixed with -2
                                                                while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                            count--;
                                                                if ( count == 0) return(pbTarget+i);
                                                    }
                                                    //i = i + 1; // r.1, obviuosly this is the worst skip so turning to 'SunHorse': lines below
// r.2 [
if ( bm_bc[(unsigned char)pbTarget[i+cbPattern-1]] < bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]] )
          i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
```

```
else
        i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
// r.2 ]
                                                } else if ( Gulliver == cbPattern-1 ) // CASE #2: means the pair (char order 2) is not found
                                                          i = i + Gulliver; // the pair is not found, skip the whole pattern and fall back (Order-1) chars i.e. one char for Order 2
                                                else
                                                          i = i +  cbPattern - Gulliver - 2; // CASE #3: the pair is found and not as suffix i.e. rightmost position

// 32323218 Order 1 Horspool
// fa af fa af fa as st Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// fafafast
//   fafafast +2 Order 1 'a' vs 't'
//   fafafast +2 = (cbPattern-Gulliver-2 = 8-4-2 = 2) Order 2 'fa' vs 'st' i.e. CASE #3

// 76543218 Order 1 Horspool
// lo on ng gp pa ac ce Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// longpace
//   longpace +2 Order 1 'a' vs 'e'
//        longpace +7 = (cbPattern-1 = 8-1 = 7) Order 2 'fa' vs 'ce' i.e. CASE #2

                                                //GlobalI++;
                                        }
           } else { // r.2
                                        while (i <= cbTarget-cbPattern-1) {
                                                if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                          count = countSTATIC;
                                                          while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e.
no need of comparing in-between chars.
                                                                    count--;
                                                          }
                                                          if ( count == 0) return(pbTarget+i);
                                                }
                                                i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
                                                //GlobalI++;
                                        }
           } // r.2

                                        if (i == cbTarget-cbPattern) {
                                                if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                          count = countSTATIC;
                                                          while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                    count--;
                                                          if ( count == 0) return(pbTarget+i);
                                                }
                                                //GlobalI++;
                                        }

                                        return(NULL);
                    } //if (cbTarget<961)
           } //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Sunday-Horspool algorithm ]

// Revision: 2, 2012-Jan-30, the main disadvantage: the preprocessing overhead.
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
//
char * Railgun_Quadruplet_7Gulliver_9 (char * pbTarget, char * pbPattern, unsigned long cbTarget, unsigned long cbPattern)
{
           char * pbTargetMax = pbTarget + cbTarget;
           register unsigned long ulHashPattern;
           register unsigned long ulHashTarget;
           signed long count;
           signed long countSTATIC;

           unsigned char SINGLET;
```

```
	unsigned long Quadruplet2nd;
	unsigned long Quadruplet3rd;
	unsigned long Quadruplet4th;

	unsigned long  AdvanceHopperGrass;

	long i; //BMH needed
	int a, j;
//Global are next 2 lines already:
	//unsigned int bm_bc[256]; //BMH needed
	//unsigned int bm_bc2nd[256]; //BMS needed
//Global is next line already:
	//unsigned char bm_Horspool_Order2[256*256]; //BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
	unsigned long Gulliver; // or unsigned char or unsigned short

	if (cbPattern > cbTarget)
		return(NULL);

	if ( cbPattern<4) {
		pbTarget = pbTarget+cbPattern;
		ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));

		if ( cbPattern==3) {
			for ( ;; ) {
				if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
					if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
				}
				if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
				pbTarget++;
				if (pbTarget > pbTargetMax)
					return(NULL);
			}
		} else {
		}
		for ( ;; ) {
			if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
				return((pbTarget-2));
			if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
			pbTarget++;
			if (pbTarget > pbTargetMax)
				return(NULL);
		}
	} else { //if ( cbPattern<4)
		if (cbTarget<961) { // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.

			pbTarget = pbTarget+cbPattern;
			ulHashPattern = *(unsigned long *)(pbPattern);

			SINGLET = ulHashPattern & 0xFF;
			Quadruplet2nd = SINGLET<<8;
			Quadruplet3rd = SINGLET<<16;
			Quadruplet4th = SINGLET<<24;

			for ( ;; ) {
				AdvanceHopperGrass = 0;
				ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

				if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
					count = cbPattern-1;
					while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
						if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
						count--;
					}
					if ( count == 0) return((pbTarget-cbPattern));
				} else { // The goal here: to avoid memory accesses by stressing the registers.
					if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
						AdvanceHopperGrass++;
						if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
							AdvanceHopperGrass++;
```

```
                                                                        if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                                                }
                                                        }
                                                }

                                                AdvanceHopperGrass++;

                                                pbTarget = pbTarget + AdvanceHopperGrass;
                                                if (pbTarget > pbTargetMax)
                                                        return(NULL);
                                        }

                                } else { //if (cbTarget<961)
                                        countSTATIC = cbPattern-2-2;

//Global are next 3 lines already:
                                        //for (a=0; a < 256; a++) {bm_bc[a]=cbPattern; bm_bc2nd[a]=cbPattern+1;}
                                        //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
                                        //for (j=0; j < cbPattern; j++) bm_bc2nd[pbPattern[j]]=cbPattern-j;

                                        // Elsiane r.2  [
//Global is next line already:
                                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // 'memset' if not optimized

                                        // alfalfa 7 long 6 BBs (al lf fa al lf fa) 3 distinct BBs (al lf fa)
                                        // fast 4 0-1-2 fa as st
//Global is next line already:
                                        //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed

                                        // Elsiane r.2 ]

                                        ulHashPattern = *(unsigned long *)(pbPattern); // First four bytes
                                        //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes

                                        i=0;
                                if ( cbPattern>10) { // r.2
                                        while (i <= cbTarget-cbPattern-1) { // -1 because Sunday is used
                                                Gulliver = bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]];

                                                if ( Gulliver == cbPattern-2 ) { // CASE #1: means the pair (char order 2) is found
                                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                                count = countSTATIC; // Last two chars already matched, to be fixed with -2
                                                                while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                        count--;
                                                                if ( count == 0) return(pbTarget+i);
                                                        }
                                                        //i = i + 1; // r.1, obviuosly this is the worst skip so turning to 'SunHorse': lines below
// r.2 [
if ( bm_bc[(unsigned char)pbTarget[i+cbPattern-1]] < bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]] )
        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
else
        i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
// r.2 ]
                                                } else if ( Gulliver == cbPattern-1 ) // CASE #2: means the pair (char order 2) is not found
                                                        i = i + Gulliver; // the pair is not found, skip the whole pattern and fall back (Order-1) chars i.e. one char for Order 2
                                                else
                                                        i = i +  cbPattern - Gulliver - 2; // CASE #3: the pair is found and not as suffix i.e. rightmost position

// 32323218 Order 1 Horspool
// fa af fa af fa as st Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// fafafast
//   fafafast +2 Order 1 'a' vs 't'
//   fafafast +2 = (cbPattern-Gulliver-2 = 8-4-2 = 2) Order 2 'fa' vs 'st' i.e. CASE #3

// 76543218 Order 1 Horspool
// lo on ng gp pa ac ce Order 2 Horspool
//  0  1  2  3  4  5  6
```

```
// HIKARIfast
// longpace
//   longpace +2 Order 1 'a' vs 'e'
//        longpace +7 = (cbPattern-1 = 8-1 = 7) Order 2 'fa' vs 'ce' i.e. CASE #2

                                        //GlobalI++;
                              }
          } else { // r.2
                              while (i <= cbTarget-cbPattern-1) {
                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                  count = countSTATIC;
                                                  while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e.
no need of comparing in-between chars.
                                                            count--;
                                                  }
                                                  if ( count == 0) return(pbTarget+i);
                                        }
                                        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
                                        //GlobalI++;
                              }
          } // r.2

                              if (i == cbTarget-cbPattern) {
                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                  count = countSTATIC;
                                                  while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                            count--;
                                                  if ( count == 0) return(pbTarget+i);
                                        }
                                        //GlobalI++;
                              }

                              return(NULL);
                    } //if (cbTarget<961)
          } //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Sunday-Horspool algorithm ]

// Revision: 2, 2012-Jan-30, the main disadvantage: the preprocessing overhead.
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
//
char * Railgun_Quadruplet_7Gulliver_0 (char * pbTarget, char * pbPattern, unsigned long cbTarget, unsigned long cbPattern)
{
          char * pbTargetMax = pbTarget + cbTarget;
          register unsigned long ulHashPattern;
          register unsigned long ulHashTarget;
          signed long count;
          signed long countSTATIC;

          unsigned char SINGLET;
          unsigned long Quadruplet2nd;
          unsigned long Quadruplet3rd;
          unsigned long Quadruplet4th;

          unsigned long  AdvanceHopperGrass;

          long i; //BMH needed
          int a, j;
//Global are next 2 lines already:
          //unsigned int bm_bc[256]; //BMH needed
          //unsigned int bm_bc2nd[256]; //BMS needed
//Global is next line already:
          //unsigned char bm_Horspool_Order2[256*256]; //BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
          unsigned long Gulliver; // or unsigned char or unsigned short

          if (cbPattern > cbTarget)
                    return(NULL);

          if ( cbPattern<4) {
```

```
		pbTarget = pbTarget+cbPattern;
		ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));

		if ( cbPattern==3) {
			for ( ;; ) {
				if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
					if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
				}
				if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
				pbTarget++;
				if (pbTarget > pbTargetMax)
					return(NULL);
			}
		} else {
		}
		for ( ;; ) {
			if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
				return((pbTarget-2));
			if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
			pbTarget++;
			if (pbTarget > pbTargetMax)
				return(NULL);
		}
	} else { //if ( cbPattern<4)
		if (cbTarget<961) { // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.

			pbTarget = pbTarget+cbPattern;
			ulHashPattern = *(unsigned long *)(pbPattern);

			SINGLET = ulHashPattern & 0xFF;
			Quadruplet2nd = SINGLET<<8;
			Quadruplet3rd = SINGLET<<16;
			Quadruplet4th = SINGLET<<24;

			for ( ;; ) {
				AdvanceHopperGrass = 0;
				ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

				if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
					count = cbPattern-1;
					while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
						if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
						count--;
					}
					if ( count == 0) return((pbTarget-cbPattern));
				} else { // The goal here: to avoid memory accesses by stressing the registers.
					if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
						AdvanceHopperGrass++;
						if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
							AdvanceHopperGrass++;
							if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
						}
					}
				}

				AdvanceHopperGrass++;

				pbTarget = pbTarget + AdvanceHopperGrass;
				if (pbTarget > pbTargetMax)
					return(NULL);
			}

		} else { //if (cbTarget<961)
			countSTATIC = cbPattern-2-2;

//Global are next 3 lines already:
			//for (a=0; a < 256; a++) {bm_bc[a]=cbPattern; bm_bc2nd[a]=cbPattern+1;}
			//for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
			//for (j=0; j < cbPattern; j++) bm_bc2nd[pbPattern[j]]=cbPattern-j;
```

```
                                // Elsiane r.2  [
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // 'memset' if not optimized

                                // alfalfa 7 long 6 BBs (al lf fa al lf fa) 3 distinct BBs (al lf fa)
                                // fast 4 0-1-2 fa as st
//Global is next line already:
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed

                                // Elsiane r.2 ]

                                ulHashPattern = *(unsigned long *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes

                                i=0;
            if ( cbPattern>10) { // r.2
                                while (i <= cbTarget-cbPattern-1) { // -1 because Sunday is used
                                            Gulliver = bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]];

                                            if ( Gulliver == cbPattern-2 ) { // CASE #1: means the pair (char order 2) is found
                                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                                    count = countSTATIC; // Last two chars already matched, to be fixed with -2
                                                                    while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                              count--;
                                                                    if ( count == 0) return(pbTarget+i);
                                                        }
                                                        //i = i + 1; // r.1, obviuosly this is the worst skip so turning to 'SunHorse': lines below
// r.2 [
if ( bm_bc[(unsigned char)pbTarget[i+cbPattern-1]] < bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]] )
         i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
else
         i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
// r.2 ]
                                            } else if ( Gulliver == cbPattern-1 ) // CASE #2: means the pair (char order 2) is not found
                                                        i = i + Gulliver; // the pair is not found, skip the whole pattern and fall back (Order-1) chars i.e. one char for Order 2
                                            else
                                                        i = i +  cbPattern - Gulliver - 2; // CASE #3: the pair is found and not as suffix i.e. rightmost position

// 32323218 Order 1 Horspool
// fa af fa af fa as st Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// fafafast
//   fafafast +2 Order 1 'a' vs 't'
//   fafafast +2 = (cbPattern-Gulliver-2 = 8-4-2 = 2) Order 2 'fa' vs 'st' i.e. CASE #3

// 76543218 Order 1 Horspool
// lo on ng gp pa ac ce Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// longpace
//   longpace +2 Order 1 'a' vs 'e'
//        longpace +7 = (cbPattern-1 = 8-1 = 7) Order 2 'fa' vs 'ce' i.e. CASE #2

                                            //GlobalI++;
                                }
            } else { // r.2
                                while (i <= cbTarget-cbPattern-1) {
                                            if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                        count = countSTATIC;
                                                        while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e.
no need of comparing in-between chars.
                                                                    count--;
                                                        }
                                                        if ( count == 0) return(pbTarget+i);
                                            }
                                            i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
                                            //GlobalI++;
```

```
                                }
            } // r.2

                                if (i == cbTarget-cbPattern) {
                                            if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                        count = countSTATIC;
                                                        while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                    count--;
                                                        if ( count == 0) return(pbTarget+i);
                                            }
                                            //GlobalI++;
                                }

                                return(NULL);
                        } //if (cbTarget<961)
            } //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Sunday-Horspool algorithm ]

// Revision: 2, 2012-Jan-30, the main disadvantage: the preprocessing overhead.
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
//
char * Railgun_Quadruplet_7Gulliver_a (char * pbTarget, char * pbPattern, unsigned long cbTarget, unsigned long cbPattern)
{
            char * pbTargetMax = pbTarget + cbTarget;
            register unsigned long ulHashPattern;
            register unsigned long ulHashTarget;
            signed long count;
            signed long countSTATIC;

            unsigned char SINGLET;
            unsigned long Quadruplet2nd;
            unsigned long Quadruplet3rd;
            unsigned long Quadruplet4th;

            unsigned long  AdvanceHopperGrass;

            long i; //BMH needed
            int a, j;
//Global are next 2 lines already:
            //unsigned int bm_bc[256]; //BMH needed
            //unsigned int bm_bc2nd[256]; //BMS needed
//Global is next line already:
            //unsigned char bm_Horspool_Order2[256*256]; //BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
            unsigned long Gulliver; // or unsigned char or unsigned short

            if (cbPattern > cbTarget)
                        return(NULL);

            if ( cbPattern<4) {
                        pbTarget = pbTarget+cbPattern;
                        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));

                        if ( cbPattern==3) {
                                    for ( ;; ) {
                                                if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
                                                            if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
                                                }
                                                if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
                                                pbTarget++;
                                                if (pbTarget > pbTargetMax)
                                                            return(NULL);
                                    }
                        } else {
                        }
                        for ( ;; ) {
                                    if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
                                                return((pbTarget-2));
                                    if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
```

```
                                pbTarget++;
                                if (pbTarget > pbTargetMax)
                                        return(NULL);
                        }
            } else { //if ( cbPattern<4)
                        if (cbTarget<961) { // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.

                                pbTarget = pbTarget+cbPattern;
                                ulHashPattern = *(unsigned long *)(pbPattern);

                                SINGLET = ulHashPattern & 0xFF;
                                Quadruplet2nd = SINGLET<<8;
                                Quadruplet3rd = SINGLET<<16;
                                Quadruplet4th = SINGLET<<24;

                                for ( ;; ) {
                                        AdvanceHopperGrass = 0;
                                        ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

                                        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
                                                count = cbPattern-1;
                                                while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                                                        if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
                                                        count--;
                                                }
                                                if ( count == 0) return((pbTarget-cbPattern));
                                        } else { // The goal here: to avoid memory accesses by stressing the registers.
                                                if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                                        AdvanceHopperGrass++;
                                                        if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                                AdvanceHopperGrass++;
                                                                if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                                        }
                                                }
                                        }

                                        AdvanceHopperGrass++;

                                        pbTarget = pbTarget + AdvanceHopperGrass;
                                        if (pbTarget > pbTargetMax)
                                                return(NULL);
                                }

                        } else { //if (cbTarget<961)
                                countSTATIC = cbPattern-2-2;

//Global are next 3 lines already:
                                //for (a=0; a < 256; a++) {bm_bc[a]=cbPattern; bm_bc2nd[a]=cbPattern+1;}
                                //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
                                //for (j=0; j < cbPattern; j++) bm_bc2nd[pbPattern[j]]=cbPattern-j;

                                // Elsiane r.2  [
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // 'memset' if not optimized

                                // alfalfa 7 long 6 BBs (al lf fa al lf fa) 3 distinct BBs (al lf fa)
                                // fast 4 0-1-2 fa as st
//Global is next line already:
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed

                                // Elsiane r.2 ]

                                ulHashPattern = *(unsigned long *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes

                                i=0;
            if ( cbPattern>10) { // r.2
                                while (i <= cbTarget-cbPattern-1) { // -1 because Sunday is used
                                        Gulliver = bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]];
```

```
                                        if ( Gulliver == cbPattern-2 ) { // CASE #1: means the pair (char order 2) is found
                                                if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                        count = countSTATIC; // Last two chars already matched, to be fixed with -2
                                                        while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                count--;
                                                        if ( count == 0) return(pbTarget+i);
                                                }
                                                //i = i + 1; // r.1, obviuosly this is the worst skip so turning to 'SunHorse': lines below
// r.2 [
if ( bm_bc[(unsigned char)pbTarget[i+cbPattern-1]] < bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]] )
        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
else
        i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
// r.2 ]
                                        } else if ( Gulliver == cbPattern-1 ) // CASE #2: means the pair (char order 2) is not found
                                                i = i + Gulliver; // the pair is not found, skip the whole pattern and fall back (Order-1) chars i.e. one char for Order 2
                                        else
                                                i = i +  cbPattern - Gulliver - 2; // CASE #3: the pair is found and not as suffix i.e. rightmost position

// 32323218 Order 1 Horspool
// fa af fa af fa as st Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// fafafast
//   fafafast +2 Order 1 'a' vs 't'
//   fafafast +2 = (cbPattern-Gulliver-2 = 8-4-2 = 2) Order 2 'fa' vs 'st' i.e. CASE #3

// 76543218 Order 1 Horspool
// lo on ng gp pa ac ce Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// longpace
//   longpace +2 Order 1 'a' vs 'e'
//        longpace +7 = (cbPattern-1 = 8-1 = 7) Order 2 'fa' vs 'ce' i.e. CASE #2

                                        //GlobalI++;
                                }
                        } else { // r.2
                                while (i <= cbTarget-cbPattern-1) {
                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                count = countSTATIC;
                                                while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e.
no need of comparing in-between chars.
                                                        count--;
                                                }
                                                if ( count == 0) return(pbTarget+i);
                                        }
                                        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
                                        //GlobalI++;
                                }
                        } // r.2

                                if (i == cbTarget-cbPattern) {
                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                count = countSTATIC;
                                                while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                        count--;
                                                if ( count == 0) return(pbTarget+i);
                                        }
                                        //GlobalI++;
                                }

                                return(NULL);
                        } //if (cbTarget<961)
                } //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Sunday-Horspool algorithm ]
```

```
// Revision: 2, 2012-Jan-30, the main disadvantage: the preprocessing overhead.
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
//
char * Railgun_Quadruplet_7Gulliver_b (char * pbTarget, char * pbPattern, unsigned long cbTarget, unsigned long cbPattern)
{
	char * pbTargetMax = pbTarget + cbTarget;
	register unsigned long ulHashPattern;
	register unsigned long ulHashTarget;
	signed long count;
	signed long countSTATIC;

	unsigned char SINGLET;
	unsigned long Quadruplet2nd;
	unsigned long Quadruplet3rd;
	unsigned long Quadruplet4th;

	unsigned long  AdvanceHopperGrass;

	long i; //BMH needed
	int a, j;
//Global are next 2 lines already:
	//unsigned int bm_bc[256]; //BMH needed
	//unsigned int bm_bc2nd[256]; //BMS needed
//Global is next line already:
	//unsigned char bm_Horspool_Order2[256*256]; //BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
	unsigned long Gulliver; // or unsigned char or unsigned short

	if (cbPattern > cbTarget)
		return(NULL);

	if ( cbPattern<4) {
		pbTarget = pbTarget+cbPattern;
		ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));

		if ( cbPattern==3) {
			for ( ;; ) {
				if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
					if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
				}
				if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
				pbTarget++;
				if (pbTarget > pbTargetMax)
					return(NULL);
			}
		} else {
		}
		for ( ;; ) {
			if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
				return((pbTarget-2));
			if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
			pbTarget++;
			if (pbTarget > pbTargetMax)
				return(NULL);
		}
	} else { //if ( cbPattern<4)
		if (cbTarget<961) { // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.

			pbTarget = pbTarget+cbPattern;
			ulHashPattern = *(unsigned long *)(pbPattern);

			SINGLET = ulHashPattern & 0xFF;
			Quadruplet2nd = SINGLET<<8;
			Quadruplet3rd = SINGLET<<16;
			Quadruplet4th = SINGLET<<24;

			for ( ;; ) {
				AdvanceHopperGrass = 0;
				ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);
```

```
                                        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
                                                count = cbPattern-1;
                                                while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                                                        if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
                                                        count--;
                                                }
                                                if ( count == 0) return((pbTarget-cbPattern));
                                        } else { // The goal here: to avoid memory accesses by stressing the registers.
                                                if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                                        AdvanceHopperGrass++;
                                                        if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                                AdvanceHopperGrass++;
                                                                if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                                        }
                                                }
                                           }

                                           AdvanceHopperGrass++;

                                           pbTarget = pbTarget + AdvanceHopperGrass;
                                           if (pbTarget > pbTargetMax)
                                                   return(NULL);
                                   }

                        } else { //if (cbTarget<961)
                                   countSTATIC = cbPattern-2-2;

//Global are next 3 lines already:
                                   //for (a=0; a < 256; a++) {bm_bc[a]=cbPattern; bm_bc2nd[a]=cbPattern+1;}
                                   //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
                                   //for (j=0; j < cbPattern; j++) bm_bc2nd[pbPattern[j]]=cbPattern-j;

                                   // Elsiane r.2  [
//Global is next line already:
                                   //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // 'memset' if not optimized

                                   // alfalfa 7 long 6 BBs (al lf fa al lf fa) 3 distinct BBs (al lf fa)
                                   // fast 4 0-1-2 fa as st
//Global is next line already:
                                   //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed

                                   // Elsiane r.2 ]

                                   ulHashPattern = *(unsigned long *)(pbPattern); // First four bytes
                                   //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes

                                   i=0;
           if ( cbPattern>10) { // r.2
                                   while (i <= cbTarget-cbPattern-1) { // -1 because Sunday is used
                                           Gulliver = bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]];

                                           if ( Gulliver == cbPattern-2 ) { // CASE #1: means the pair (char order 2) is found
                                                   if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                           count = countSTATIC; // Last two chars already matched, to be fixed with -2
                                                           while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                   count--;
                                                           if ( count == 0) return(pbTarget+i);
                                                   }
                                                   //i = i + 1; // r.1, obviuosly this is the worst skip so turning to 'SunHorse': lines below
// r.2 [
if ( bm_bc[(unsigned char)pbTarget[i+cbPattern-1]] < bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]] )
        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
else
        i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
// r.2 ]
                                           } else if ( Gulliver == cbPattern-1 ) // CASE #2: means the pair (char order 2) is not found
                                                   i = i + Gulliver; // the pair is not found, skip the whole pattern and fall back (Order-1) chars i.e. one char for Order 2
                                           else
                                                   i = i +  cbPattern - Gulliver - 2; // CASE #3: the pair is found and not as suffix i.e. rightmost position
```

```
// 32323218 Order 1 Horspool
// fa af fa af fa as st Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// fafafast
//   fafafast +2 Order 1 'a' vs 't'
//   fafafast +2 = (cbPattern-Gulliver-2 = 8-4-2 = 2) Order 2 'fa' vs 'st' i.e. CASE #3

// 76543218 Order 1 Horspool
// lo on ng gp pa ac ce Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// longpace
//   longpace +2 Order 1 'a' vs 'e'
//        longpace +7 = (cbPattern-1 = 8-1 = 7) Order 2 'fa' vs 'ce' i.e. CASE #2

                                        //GlobalI++;
                              }
          } else { // r.2
                              while (i <= cbTarget-cbPattern-1) {
                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                  count = countSTATIC;
                                                  while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e.
no need of comparing in-between chars.
                                                            count--;
                                                  }
                                                  if ( count == 0) return(pbTarget+i);
                                        }
                                        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
                                        //GlobalI++;
                              }
          } // r.2

                              if (i == cbTarget-cbPattern) {
                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                  count = countSTATIC;
                                                  while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                            count--;
                                                  if ( count == 0) return(pbTarget+i);
                                        }
                                        //GlobalI++;
                              }

                              return(NULL);
                    } //if (cbTarget<961)
          } //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Sunday-Horspool algorithm ]

// Revision: 2, 2012-Jan-30, the main disadvantage: the preprocessing overhead.
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
//
char * Railgun_Quadruplet_7Gulliver_c (char * pbTarget, char * pbPattern, unsigned long cbTarget, unsigned long cbPattern)
{
          char * pbTargetMax = pbTarget + cbTarget;
          register unsigned long ulHashPattern;
          register unsigned long ulHashTarget;
          signed long count;
          signed long countSTATIC;

          unsigned char SINGLET;
          unsigned long Quadruplet2nd;
          unsigned long Quadruplet3rd;
          unsigned long Quadruplet4th;

          unsigned long  AdvanceHopperGrass;

          long i; //BMH needed
```

```
	int a, j;
//Global are next 2 lines already:
	//unsigned int bm_bc[256]; //BMH needed
	//unsigned int bm_bc2nd[256]; //BMS needed
//Global is next line already:
	//unsigned char bm_Horspool_Order2[256*256]; //BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
	unsigned long Gulliver; // or unsigned char or unsigned short

	if (cbPattern > cbTarget)
		return(NULL);

	if ( cbPattern<4) {
		pbTarget = pbTarget+cbPattern;
		ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));

		if ( cbPattern==3) {
			for ( ;; ) {
				if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
					if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
				}
				if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
				pbTarget++;
				if (pbTarget > pbTargetMax)
					return(NULL);
			}
		} else {
		}
		for ( ;; ) {
			if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
				return((pbTarget-2));
			if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
			pbTarget++;
			if (pbTarget > pbTargetMax)
				return(NULL);
		}
	} else { //if ( cbPattern<4)
		if (cbTarget<961) { // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.

			pbTarget = pbTarget+cbPattern;
			ulHashPattern = *(unsigned long *)(pbPattern);

			SINGLET = ulHashPattern & 0xFF;
			Quadruplet2nd = SINGLET<<8;
			Quadruplet3rd = SINGLET<<16;
			Quadruplet4th = SINGLET<<24;

			for ( ;; ) {
				AdvanceHopperGrass = 0;
				ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

				if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
					count = cbPattern-1;
					while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
						if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
						count--;
					}
					if ( count == 0) return((pbTarget-cbPattern));
				} else { // The goal here: to avoid memory accesses by stressing the registers.
					if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
						AdvanceHopperGrass++;
						if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
							AdvanceHopperGrass++;
							if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
						}
					}
				}

				AdvanceHopperGrass++;
```

```
                                        pbTarget = pbTarget + AdvanceHopperGrass;
                                        if (pbTarget > pbTargetMax)
                                                return(NULL);
                                }

                        } else { //if (cbTarget<961)
                                countSTATIC = cbPattern-2-2;

//Global are next 3 lines already:
                                //for (a=0; a < 256; a++) {bm_bc[a]=cbPattern; bm_bc2nd[a]=cbPattern+1;}
                                //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
                                //for (j=0; j < cbPattern; j++) bm_bc2nd[pbPattern[j]]=cbPattern-j;

                                // Elsiane r.2  [
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // 'memset' if not optimized

                                // alfalfa 7 long 6 BBs (al lf fa al lf fa) 3 distinct BBs (al lf fa)
                                // fast 4 0-1-2 fa as st
//Global is next line already:
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed

                                // Elsiane r.2 ]

                                ulHashPattern = *(unsigned long *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes

                                i=0;
            if ( cbPattern>10) { // r.2
                                while (i <= cbTarget-cbPattern-1) { // -1 because Sunday is used
                                        Gulliver = bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]];

                                        if ( Gulliver == cbPattern-2 ) { // CASE #1: means the pair (char order 2) is found
                                                if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                        count = countSTATIC; // Last two chars already matched, to be fixed with -2
                                                        while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                count--;
                                                        if ( count == 0) return(pbTarget+i);
                                                }
                                                //i = i + 1; // r.1, obviuosly this is the worst skip so turning to 'SunHorse': lines below
// r.2 [
if ( bm_bc[(unsigned char)pbTarget[i+cbPattern-1]] < bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]] )
        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
else
        i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
// r.2 ]
                                        } else if ( Gulliver == cbPattern-1 ) // CASE #2: means the pair (char order 2) is not found
                                                i = i + Gulliver; // the pair is not found, skip the whole pattern and fall back (Order-1) chars i.e. one char for Order 2
                                        else
                                                i = i +  cbPattern - Gulliver - 2; // CASE #3: the pair is found and not as suffix i.e. rightmost position

// 32323218 Order 1 Horspool
// fa af fa af fa as st Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// fafafast
//   fafafast +2 Order 1 'a' vs 't'
//   fafafast +2 = (cbPattern-Gulliver-2 = 8-4-2 = 2) Order 2 'fa' vs 'st' i.e. CASE #3

// 76543218 Order 1 Horspool
// lo on ng gp pa ac ce Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// longpace
//   longpace +2 Order 1 'a' vs 'e'
//        longpace +7 = (cbPattern-1 = 8-1 = 7) Order 2 'fa' vs 'ce' i.e. CASE #2

                                        //GlobalI++;
                                }
```

```
		} else { // r.2
				while (i <= cbTarget-cbPattern-1) {
					if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
						count = countSTATIC;
						while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e.
no need of comparing in-between chars.
							count--;
						}
						if ( count == 0) return(pbTarget+i);
					}
					i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
					//GlobalI++;
				}
		} // r.2

				if (i == cbTarget-cbPattern) {
					if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
						count = countSTATIC;
						while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
							count--;
						if ( count == 0) return(pbTarget+i);
					}
					//GlobalI++;
				}

				return(NULL);
			} //if (cbTarget<961)
		} //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Sunday-Horspool algorithm ]

// Revision: 2, 2012-Jan-30, the main disadvantage: the preprocessing overhead.
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
//
char * Railgun_Quadruplet_7Gulliver_d (char * pbTarget, char * pbPattern, unsigned long cbTarget, unsigned long cbPattern)
{
		char * pbTargetMax = pbTarget + cbTarget;
		register unsigned long ulHashPattern;
		register unsigned long ulHashTarget;
		signed long count;
		signed long countSTATIC;

		unsigned char SINGLET;
		unsigned long Quadruplet2nd;
		unsigned long Quadruplet3rd;
		unsigned long Quadruplet4th;

		unsigned long  AdvanceHopperGrass;

		long i; //BMH needed
		int a, j;
//Global are next 2 lines already:
		//unsigned int bm_bc[256]; //BMH needed
		//unsigned int bm_bc2nd[256]; //BMS needed
//Global is next line already:
		//unsigned char bm_Horspool_Order2[256*256]; //BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
		unsigned long Gulliver; // or unsigned char or unsigned short

		if (cbPattern > cbTarget)
			return(NULL);

		if ( cbPattern<4) {
			pbTarget = pbTarget+cbPattern;
			ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));

			if ( cbPattern==3) {
				for ( ;; ) {
					if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
						if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
```

```
                                        }
                                        if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
                                        pbTarget++;
                                        if (pbTarget > pbTargetMax)
                                                return(NULL);
                                }
                        } else {
                        }
                        for ( ;; ) {
                                if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
                                        return((pbTarget-2));
                                if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
                                pbTarget++;
                                if (pbTarget > pbTargetMax)
                                        return(NULL);
                        }
                } else { //if ( cbPattern<4)
                        if (cbTarget<961) { // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.

                                pbTarget = pbTarget+cbPattern;
                                ulHashPattern = *(unsigned long *)(pbPattern);

                                SINGLET = ulHashPattern & 0xFF;
                                Quadruplet2nd = SINGLET<<8;
                                Quadruplet3rd = SINGLET<<16;
                                Quadruplet4th = SINGLET<<24;

                                for ( ;; ) {
                                        AdvanceHopperGrass = 0;
                                        ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

                                        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
                                                count = cbPattern-1;
                                                while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                                                        if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
                                                        count--;
                                                }
                                                if ( count == 0) return((pbTarget-cbPattern));
                                        } else { // The goal here: to avoid memory accesses by stressing the registers.
                                                if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                                        AdvanceHopperGrass++;
                                                        if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                                AdvanceHopperGrass++;
                                                                if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                                        }
                                                }
                                        }

                                        AdvanceHopperGrass++;

                                        pbTarget = pbTarget + AdvanceHopperGrass;
                                        if (pbTarget > pbTargetMax)
                                                return(NULL);
                                }

                        } else { //if (cbTarget<961)
                                countSTATIC = cbPattern-2-2;

//Global are next 3 lines already:
                                //for (a=0; a < 256; a++) {bm_bc[a]=cbPattern; bm_bc2nd[a]=cbPattern+1;}
                                //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
                                //for (j=0; j < cbPattern; j++) bm_bc2nd[pbPattern[j]]=cbPattern-j;

                                // Elsiane r.2  [
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // 'memset' if not optimized

                                // alfalfa 7 long 6 BBs (al lf fa al lf fa) 3 distinct BBs (al lf fa)
                                // fast 4 0-1-2 fa as st
```

```
//Global is next line already:
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed

                                // Elsiane r.2 ]

                                ulHashPattern = *(unsigned long *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes

                                i=0;
            if ( cbPattern>10) { // r.2
                                while (i <= cbTarget-cbPattern-1) { // -1 because Sunday is used
                                            Gulliver = bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]];

                                            if ( Gulliver == cbPattern-2 ) { // CASE #1: means the pair (char order 2) is found
                                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                                    count = countSTATIC; // Last two chars already matched, to be fixed with -2
                                                                    while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                                count--;
                                                                    if ( count == 0) return(pbTarget+i);
                                                        }
                                                        //i = i + 1; // r.1, obviuosly this is the worst skip so turning to 'SunHorse': lines below
// r.2 [
if ( bm_bc[(unsigned char)pbTarget[i+cbPattern-1]] < bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]] )
        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
else
        i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
// r.2 ]
                                            } else if ( Gulliver == cbPattern-1 ) // CASE #2: means the pair (char order 2) is not found
                                                        i = i + Gulliver; // the pair is not found, skip the whole pattern and fall back (Order-1) chars i.e. one char for Order 2
                                            else
                                                        i = i +  cbPattern - Gulliver - 2; // CASE #3: the pair is found and not as suffix i.e. rightmost position

// 32323218 Order 1 Horspool
// fa af fa af fa as st Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// fafafast
//   fafafast +2 Order 1 'a' vs 't'
//   fafafast +2 = (cbPattern-Gulliver-2 = 8-4-2 = 2) Order 2 'fa' vs 'st' i.e. CASE #3

// 76543218 Order 1 Horspool
// lo on ng gp pa ac ce Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// longpace
//   longpace +2 Order 1 'a' vs 'e'
//        longpace +7 = (cbPattern-1 = 8-1 = 7) Order 2 'fa' vs 'ce' i.e. CASE #2

                                            //GlobalI++;
                                }
            } else { // r.2
                                while (i <= cbTarget-cbPattern-1) {
                                            if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                        count = countSTATIC;
                                                        while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e.
no need of comparing in-between chars.
                                                                    count--;
                                                        }
                                                        if ( count == 0) return(pbTarget+i);
                                            }
                                            i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
                                            //GlobalI++;
                                }
            } // r.2

                                if (i == cbTarget-cbPattern) {
                                            if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                        count = countSTATIC;
                                                        while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
```

```
                                                        count--;
                                                if ( count == 0) return(pbTarget+i);
                                        }
                                        //GlobalI++;
                                }

                                return(NULL);
                        } //if (cbTarget<961)
            } //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Sunday-Horspool algorithm ]

// Revision: 2, 2012-Jan-30, the main disadvantage: the preprocessing overhead.
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
//
char * Railgun_Quadruplet_7Gulliver_e (char * pbTarget, char * pbPattern, unsigned long cbTarget, unsigned long cbPattern)
{
            char * pbTargetMax = pbTarget + cbTarget;
            register unsigned long ulHashPattern;
            register unsigned long ulHashTarget;
            signed long count;
            signed long countSTATIC;

            unsigned char SINGLET;
            unsigned long Quadruplet2nd;
            unsigned long Quadruplet3rd;
            unsigned long Quadruplet4th;

            unsigned long  AdvanceHopperGrass;

            long i; //BMH needed
            int a, j;
//Global are next 2 lines already:
            //unsigned int bm_bc[256]; //BMH needed
            //unsigned int bm_bc2nd[256]; //BMS needed
//Global is next line already:
            //unsigned char bm_Horspool_Order2[256*256]; //BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
            unsigned long Gulliver; // or unsigned char or unsigned short

            if (cbPattern > cbTarget)
                        return(NULL);

            if ( cbPattern<4) {
                        pbTarget = pbTarget+cbPattern;
                        ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));

                        if ( cbPattern==3) {
                                    for ( ;; ) {
                                                if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
                                                            if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
                                                }
                                                if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
                                                pbTarget++;
                                                if (pbTarget > pbTargetMax)
                                                            return(NULL);
                                    }
                        } else {
                        }
                        for ( ;; ) {
                                    if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
                                                return((pbTarget-2));
                                    if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
                                    pbTarget++;
                                    if (pbTarget > pbTargetMax)
                                                return(NULL);
                        }
            } else { //if ( cbPattern<4)
                        if (cbTarget<961) { // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.
```

```
                                pbTarget = pbTarget+cbPattern;
                                ulHashPattern = *(unsigned long *)(pbPattern);

                                SINGLET = ulHashPattern & 0xFF;
                                Quadruplet2nd = SINGLET<<8;
                                Quadruplet3rd = SINGLET<<16;
                                Quadruplet4th = SINGLET<<24;

                                for ( ;; ) {
                                           AdvanceHopperGrass = 0;
                                           ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

                                        if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
                                                   count = cbPattern-1;
                                                   while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
                                                              if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
                                                              count--;
                                                   }
                                                   if ( count == 0) return((pbTarget-cbPattern));
                                        } else { // The goal here: to avoid memory accesses by stressing the registers.
                                                   if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                                              AdvanceHopperGrass++;
                                                              if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                                         AdvanceHopperGrass++;
                                                                         if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                                              }
                                                   }
                                           }

                                           AdvanceHopperGrass++;

                                           pbTarget = pbTarget + AdvanceHopperGrass;
                                           if (pbTarget > pbTargetMax)
                                                   return(NULL);
                                }

                        } else { //if (cbTarget<961)
                                countSTATIC = cbPattern-2-2;

//Global are next 3 lines already:
                                //for (a=0; a < 256; a++) {bm_bc[a]=cbPattern; bm_bc2nd[a]=cbPattern+1;}
                                //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
                                //for (j=0; j < cbPattern; j++) bm_bc2nd[pbPattern[j]]=cbPattern-j;

                                // Elsiane r.2  [
//Global is next line already:
                                //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // 'memset' if not optimized

                                // alfalfa 7 long 6 BBs (al lf fa al lf fa) 3 distinct BBs (al lf fa)
                                // fast 4 0-1-2 fa as st
//Global is next line already:
                                //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed

                                // Elsiane r.2 ]

                                ulHashPattern = *(unsigned long *)(pbPattern); // First four bytes
                                //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes

                                i=0;
            if ( cbPattern>10) { // r.2
                                while (i <= cbTarget-cbPattern-1) { // -1 because Sunday is used
                                           Gulliver = bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]];

                                           if ( Gulliver == cbPattern-2 ) { // CASE #1: means the pair (char order 2) is found
                                                   if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                              count = countSTATIC; // Last two chars already matched, to be fixed with -2
                                                              while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                         count--;
                                                              if ( count == 0) return(pbTarget+i);
```

```
                                                        }
                                                        //i = i + 1; // r.1, obviuosly this is the worst skip so turning to 'SunHorse': lines below
// r.2 [
if ( bm_bc[(unsigned char)pbTarget[i+cbPattern-1]] < bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]] )
        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
else
        i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
// r.2 ]
                                                } else if ( Gulliver == cbPattern-1 ) // CASE #2: means the pair (char order 2) is not found
                                                        i = i + Gulliver; // the pair is not found, skip the whole pattern and fall back (Order-1) chars i.e. one char for Order 2
                                                else
                                                        i = i +  cbPattern - Gulliver - 2; // CASE #3: the pair is found and not as suffix i.e. rightmost position

// 32323218 Order 1 Horspool
// fa af fa af fa as st Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// fafafast
//   fafafast +2 Order 1 'a' vs 't'
//   fafafast +2 = (cbPattern-Gulliver-2 = 8-4-2 = 2) Order 2 'fa' vs 'st' i.e. CASE #3

// 76543218 Order 1 Horspool
// lo on ng gp pa ac ce Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// longpace
//   longpace +2 Order 1 'a' vs 'e'
//        longpace +7 = (cbPattern-1 = 8-1 = 7) Order 2 'fa' vs 'ce' i.e. CASE #2

                                                //GlobalI++;
                                        }
                                } else { // r.2
                                        while (i <= cbTarget-cbPattern-1) {
                                                if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                        count = countSTATIC;
                                                        while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e.
no need of comparing in-between chars.
                                                                count--;
                                                        }
                                                        if ( count == 0) return(pbTarget+i);
                                                }
                                                i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
                                                //GlobalI++;
                                        }
                                } // r.2

                                        if (i == cbTarget-cbPattern) {
                                                if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                        count = countSTATIC;
                                                        while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                count--;
                                                        if ( count == 0) return(pbTarget+i);
                                                }
                                                //GlobalI++;
                                        }

                                        return(NULL);
                                } //if (cbTarget<961)
                        } //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Sunday-Horspool algorithm ]

// Revision: 2, 2012-Jan-30, the main disadvantage: the preprocessing overhead.
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
//
char * Railgun_Quadruplet_7Gulliver_f (char * pbTarget, char * pbPattern, unsigned long cbTarget, unsigned long cbPattern)
{
                        char * pbTargetMax = pbTarget + cbTarget;
                        register unsigned long ulHashPattern;
```

```
	register unsigned long ulHashTarget;
	signed long count;
	signed long countSTATIC;

	unsigned char SINGLET;
	unsigned long Quadruplet2nd;
	unsigned long Quadruplet3rd;
	unsigned long Quadruplet4th;

	unsigned long  AdvanceHopperGrass;

	long i; //BMH needed
	int a, j;
//Global are next 2 lines already:
	//unsigned int bm_bc[256]; //BMH needed
	//unsigned int bm_bc2nd[256]; //BMS needed
//Global is next line already:
	//unsigned char bm_Horspool_Order2[256*256]; //BMHSS(Elsiane) needed, 'char' limits patterns to 255, if 'long' then table becomes 256KB, grrr.
	unsigned long Gulliver; // or unsigned char or unsigned short

	if (cbPattern > cbTarget)
		return(NULL);

	if ( cbPattern<4) {
		pbTarget = pbTarget+cbPattern;
		ulHashPattern = ( (*(char *)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));

		if ( cbPattern==3) {
			for ( ;; ) {
				if ( ulHashPattern == ( (*(char *)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
					if ( *(char *)(pbPattern+1) == *(char *)(pbTarget-2) ) return((pbTarget-3));
				}
				if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
				pbTarget++;
				if (pbTarget > pbTargetMax)
					return(NULL);
			}
		} else {
		}
		for ( ;; ) {
			if ( ulHashPattern == ( (*(char *)(pbTarget-2))<<8 ) + *(pbTarget-1) )
				return((pbTarget-2));
			if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
			pbTarget++;
			if (pbTarget > pbTargetMax)
				return(NULL);
		}
	} else { //if ( cbPattern<4)
		if (cbTarget<961) { // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than 'Boyer_Moore_Horspool'.

			pbTarget = pbTarget+cbPattern;
			ulHashPattern = *(unsigned long *)(pbPattern);

			SINGLET = ulHashPattern & 0xFF;
			Quadruplet2nd = SINGLET<<8;
			Quadruplet3rd = SINGLET<<16;
			Quadruplet4th = SINGLET<<24;

			for ( ;; ) {
				AdvanceHopperGrass = 0;
				ulHashTarget = *(unsigned long *)(pbTarget-cbPattern);

				if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a higher priority.
					count = cbPattern-1;
					while ( count && *(char *)(pbPattern+(cbPattern-count)) == *(char *)(pbTarget-count) ) {
						if ( cbPattern-1==AdvanceHopperGrass+count && SINGLET != *(char *)(pbTarget-count) ) AdvanceHopperGrass++;
						count--;
					}
					if ( count == 0) return((pbTarget-cbPattern));
```

```
                                        } else { // The goal here: to avoid memory accesses by stressing the registers.
                                                    if ( Quadruplet2nd != (ulHashTarget & 0x0000FF00) ) {
                                                                AdvanceHopperGrass++;
                                                                if ( Quadruplet3rd != (ulHashTarget & 0x00FF0000) ) {
                                                                            AdvanceHopperGrass++;
                                                                            if ( Quadruplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
                                                                }
                                                    }
                                                }

                                                AdvanceHopperGrass++;

                                                pbTarget = pbTarget + AdvanceHopperGrass;
                                                if (pbTarget > pbTargetMax)
                                                            return(NULL);
                                        }

                                } else { //if (cbTarget<961)
                                        countSTATIC = cbPattern-2-2;

//Global are next 3 lines already:
                                        //for (a=0; a < 256; a++) {bm_bc[a]=cbPattern; bm_bc2nd[a]=cbPattern+1;}
                                        //for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
                                        //for (j=0; j < cbPattern; j++) bm_bc2nd[pbPattern[j]]=cbPattern-j;

                                        // Elsiane r.2  [
//Global is next line already:
                                        //for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= cbPattern-1;} // 'memset' if not optimized

                                        // alfalfa 7 long 6 BBs (al lf fa al lf fa) 3 distinct BBs (al lf fa)
                                        // fast 4 0-1-2 fa as st
//Global is next line already:
                                        //for (j=0; j < cbPattern-1; j++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+j)]=j; // Rightmost appearance/position is needed

                                        // Elsiane r.2 ]

                                        ulHashPattern = *(unsigned long *)(pbPattern); // First four bytes
                                        //ulHashTarget = *(unsigned short *)(pbPattern+cbPattern-1-1); // Last two bytes

                                        i=0;
                    if ( cbPattern>10) { // r.2
                                        while (i <= cbTarget-cbPattern-1) { // -1 because Sunday is used
                                                    Gulliver = bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]];

                                                    if ( Gulliver == cbPattern-2 ) { // CASE #1: means the pair (char order 2) is found
                                                                if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                                            count = countSTATIC; // Last two chars already matched, to be fixed with -2
                                                                            while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                                                        count--;
                                                                            if ( count == 0) return(pbTarget+i);
                                                                }
                                                                //i = i + 1; // r.1, obviuosly this is the worst skip so turning to 'SunHorse': lines below
// r.2 [
if ( bm_bc[(unsigned char)pbTarget[i+cbPattern-1]] < bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]] )
        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
else
        i= i + bm_bc[(unsigned char)pbTarget[i+cbPattern-1]];
// r.2 ]
                                                    } else if ( Gulliver == cbPattern-1 ) // CASE #2: means the pair (char order 2) is not found
                                                                i = i + Gulliver; // the pair is not found, skip the whole pattern and fall back (Order-1) chars i.e. one char for Order 2
                                                    else
                                                                i = i +  cbPattern - Gulliver - 2; // CASE #3: the pair is found and not as suffix i.e. rightmost position

// 32323218 Order 1 Horspool
// fa af fa af fa as st Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// fafafast
//   fafafast +2 Order 1 'a' vs 't'
```

```
//   fafafast +2 = (cbPattern-Gulliver-2 = 8-4-2 = 2) Order 2 'fa' vs 'st' i.e. CASE #3

// 76543218 Order 1 Horspool
// lo on ng gp pa ac ce Order 2 Horspool
//  0  1  2  3  4  5  6
// HIKARIfast
// longpace
//   longpace +2 Order 1 'a' vs 'e'
//        longpace +7 = (cbPattern-1 = 8-1 = 7) Order 2 'fa' vs 'ce' i.e. CASE #2

                                        //GlobalI++;
                                }
            } else { // r.2
                                while (i <= cbTarget-cbPattern-1) {
                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                count = countSTATIC;
                                                while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) ) { // if pattern length is 4 or 5 we have count=-1 and count=0 respectively i.e.
no need of comparing in-between chars.
                                                        count--;
                                                }
                                                if ( count == 0) return(pbTarget+i);
                                        }
                                        i= i + bm_bc2nd[(unsigned char)pbTarget[i+(cbPattern)]];
                                        //GlobalI++;
                                }
            } // r.2

                                if (i == cbTarget-cbPattern) {
                                        if ( *(unsigned long *)&pbTarget[i] == ulHashPattern) {
                                                count = countSTATIC;
                                                while ( count !=0 && *(char *)(pbPattern+(countSTATIC-count)+4) == *(char *)(&pbTarget[i]+(countSTATIC-count)+4) )
                                                        count--;
                                                if ( count == 0) return(pbTarget+i);
                                        }
                                        //GlobalI++;
                                }

                                return(NULL);
                        } //if (cbTarget<961)
            } //if ( cbPattern<4)
}
// ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Sunday-Horspool algorithm ]


void * memchrKAZE (const void * buf, const void * chr, unsigned long cnt)
{
        while ( cnt && (*(unsigned char *)buf != *(unsigned char*)chr) ) {
                buf = (unsigned char *)buf + 1;
                cnt--;
        }

        return(cnt ? (void *)buf : NULL);
}
//Exit:
//       returns pointer to first occurence of chr in buf
//       returns NULL if chr not found in the first cnt bytes


long KAZE_strlen (const char * str)
{
        const char *eos = str;

        while( *eos++ ) ;

        return( (int)(eos - str - 1) );
}
//_KAZE_strlen PROC NEAR
//; Line 225: const char *eos = str;
//        mov     ecx, DWORD PTR _str$[esp-4]
```

```
//        mov     eax, ecx
//$L725:
//; Line 227: while( *eos++ ) ;
//        mov     dl, BYTE PTR [eax]
//        inc     eax
//        test    dl, dl
//        jne     SHORT $L725
//; Line 229: return( (int)(eos - str - 1) );
//        sub     eax, ecx
//        dec     eax
//; Line 230
//        ret     0
//_KAZE_strlen ENDP

long KAZE_strlenLF (const char * str)
{
        const char *eos = str;
        char LFa[1];
        LFa[0] = 10; //BUG UNcrushed yet: for Windows 13 for POSIX 10
        while( *eos++ != LFa[0] ) ;

        return( (int)(eos - str - 1) );
}

//       wildcard '*' any character(s) or empty,
//       wildcard '.' any ALPHA character(s) or empty,
//       wildcard '`' any NON-ALPHA character(s) or empty,
//       wildcard '@'/'#' any character {or empty}/{and not empty},
//       wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
//       wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty},
//       TO-DO: wildcard '+'/'`' any WORD {or empty}/{and not empty}.

// Note: Due to different line endings(CRLF in Windows; LF in UNIX) you must add a '|' wildcard in place of CR: for example in case of searching for '*.pdf' write '*.pdf|'.
// Pattern example: *%%take@%%$$|


// Igor Pavlov's variant modified by Kaze
// '&' stands for "standard" '*', '+' stands for "standard" '?'
static boolean Wildcard_IP(const char *mask, int maskPos, const char *name, int namePos)
{
char maskChar;
int maskLen = maskGLOBALlen - maskPos;
int nameLen = nameGLOBALlen1 - namePos;
if (maskLen == 0)
           if (nameLen == 0)
                      return true;
           else
                      return false;
maskChar = mask[maskPos];
if (maskChar == '+') { // and not empty
           /*
           if (Wildcard_IP(mask, maskPos + 1, name, namePos))
                      return true;
           */
           if (nameLen == 0)
                      return false;
           return Wildcard_IP(mask, maskPos + 1, name, namePos + 1);
} else if (maskChar == '&') {
           if (Wildcard_IP(mask, maskPos + 1, name, namePos))
                      return true;
           if (nameLen == 0)
                      return false;
           return Wildcard_IP(mask, maskPos, name, namePos + 1);
} else {
           if (maskChar != name[namePos])
                      return false;
           return Wildcard_IP(mask, maskPos + 1, name, namePos + 1);
}
}
```

```
/*
; Compiler: Intel C++ Compiler XE for applications running on IA-32, Version 12.1
; Options: "-O3 -Qunroll";
_Wildcard_IP          PROC NEAR PRIVATE
; parameter 1: eax
; parameter 2: edx
; parameter 3: ecx
; parameter 4: 72 + esp
.B6.1:

;;; {

  00000 56              push esi
  00001 57              push edi
  00002 53              push ebx
  00003 55              push ebp
  00004 83 ec 28        sub esp, 40
  00007 8b fa           mov edi, edx
  00009 89 44 24 10     mov DWORD PTR [16+esp], eax
  0000d f7 df           neg edi
  0000f 89 4c 24 14     mov DWORD PTR [20+esp], ecx
  00013 8b 44 24 48     mov eax, DWORD PTR [72+esp]

;;; char maskChar;
;;; int maskLen = maskGLOBALlen - maskPos;

  00017 8b e8           mov ebp, eax
  00019 89 7c 24 18     mov DWORD PTR [24+esp], edi
  0001d 8b f8           mov edi, eax
  0001f 8b 35 00 00 00
        00              mov esi, DWORD PTR [_maskGLOBALlen]
  00025 f7 df           neg edi

;;; int nameLen = nameGLOBALlen1 - namePos;

  00027 8b 0d 00 00 00
        00              mov ecx, DWORD PTR [_nameGLOBALlen1]
  0002d 89 74 24 1c     mov DWORD PTR [28+esp], esi
  00031 2b f2           sub esi, edx
  00033 89 4c 24 24     mov DWORD PTR [36+esp], ecx
  00037 89 54 24 20     mov DWORD PTR [32+esp], edx

.B6.2:
  0003b 8b 54 24 24     mov edx, DWORD PTR [36+esp]

;;; if (maskLen == 0)

  0003f 85 f6           test esi, esi
  00041 8d 1c 3a        lea ebx, DWORD PTR [edx+edi]
  00044 75 18           jne .B6.4 ; Prob 50%

.B6.3:
  00046 8b c3           mov eax, ebx
  00048 33 c9           xor ecx, ecx
  0004a ba 01 00 00 00  mov edx, 1
  0004f 85 c0           test eax, eax
  00051 0f 44 ca        cmove ecx, edx
  00054 8b c1           mov eax, ecx
  00056 83 c4 28        add esp, 40
  00059 5d              pop ebp
  0005a 5b              pop ebx
  0005b 5f              pop edi
  0005c 5e              pop esi
  0005d c3              ret

.B6.4:

;;;        if (nameLen == 0)
;;;                    return true;
```

```
;;;        else
;;;                    return false;
;;; maskChar = mask[maskPos];

  0005e 8b 4c 24 10      mov ecx, DWORD PTR [16+esp]
  00062 8b 54 24 20      mov edx, DWORD PTR [32+esp]
  00066 0f be 14 0a      movsx edx, BYTE PTR [edx+ecx]

;;; if (maskChar == '+') { // and not empty

  0006a 83 fa 2b         cmp edx, 43
  0006d 75 1c            jne .B6.7 ; Prob 67%

.B6.5:

;;;        / *
;;;        if (Wildcard_IP(mask, maskPos + 1, name, namePos))
;;;                    return true;
;;;        * /
;;;        if (nameLen == 0)

  0006f 85 db            test ebx, ebx
  00071 74 74            je .B6.14 ; Prob 5%

.B6.6:

;;;                    return false;
;;;        return Wildcard_IP(mask, maskPos + 1, name, namePos + 1);

  00073 8b 4c 24 18      mov ecx, DWORD PTR [24+esp]
  00077 4f               dec edi
  00078 49               dec ecx
  00079 45               inc ebp
  0007a 8b 54 24 1c      mov edx, DWORD PTR [28+esp]
  0007e ff 44 24 20      inc DWORD PTR [32+esp]
  00082 89 4c 24 18      mov DWORD PTR [24+esp], ecx
  00086 8d 34 0a         lea esi, DWORD PTR [edx+ecx]
  00089 eb b0            jmp .B6.2 ; Prob 100%

.B6.7:
  0008b 83 fa 26         cmp edx, 38
  0008e 75 32            jne .B6.12 ; Prob 50%

.B6.8:

;;; } else if (maskChar == '&') {
;;;        if (Wildcard_IP(mask, maskPos + 1, name, namePos))

  00090 8b 54 24 20      mov edx, DWORD PTR [32+esp]
  00094 89 6c 24 0c      mov DWORD PTR [12+esp], ebp
  00098 8b c1            mov eax, ecx
  0009a 8b 4c 24 14      mov ecx, DWORD PTR [20+esp]
  0009e 8d 52 01         lea edx, DWORD PTR [1+edx]
  000a1 e8 fc ff ff ff   call _Wildcard_IP

.B6.19:
  000a6 0f b6 c0         movzx eax, al
  000a9 85 c0            test eax, eax
  000ab 75 08            jne .B6.11 ; Prob 28%

.B6.9:

;;;                    return true;
;;;        if (nameLen == 0)

  000ad 85 db            test ebx, ebx
  000af 74 36            je .B6.14 ; Prob 5%

.B6.10:
```

```
;;;                     return false;
;;;         return Wildcard_IP(mask, maskPos, name, namePos + 1);

  000b1 4f              dec edi
  000b2 45              inc ebp
  000b3 eb 86           jmp .B6.2 ; Prob 100%

.B6.11:
  000b5 b8 01 00 00 00  mov eax, 1
  000ba 83 c4 28        add esp, 40
  000bd 5d              pop ebp
  000be 5b              pop ebx
  000bf 5f              pop edi
  000c0 5e              pop esi
  000c1 c3              ret

.B6.12:

;;; } else {
;;;         if (maskChar != name[namePos])

  000c2 8b 4c 24 14     mov ecx, DWORD PTR [20+esp]
  000c6 3a 54 0d 00     cmp dl, BYTE PTR [ebp+ecx]
  000ca 75 1b           jne .B6.14 ; Prob 5%

.B6.13:

;;;                     return false;
;;;         return Wildcard_IP(mask, maskPos + 1, name, namePos + 1);

  000cc 8b 4c 24 18     mov ecx, DWORD PTR [24+esp]
  000d0 4f              dec edi
  000d1 49              dec ecx
  000d2 45              inc ebp
  000d3 8b 54 24 1c     mov edx, DWORD PTR [28+esp]
  000d7 ff 44 24 20     inc DWORD PTR [32+esp]
  000db 89 4c 24 18     mov DWORD PTR [24+esp], ecx
  000df 8d 34 0a        lea esi, DWORD PTR [edx+ecx]
  000e2 e9 54 ff ff ff  jmp .B6.2 ; Prob 100%

.B6.14:
  000e7 33 c0           xor eax, eax
  000e9 83 c4 28        add esp, 40
  000ec 5d              pop ebp
  000ed 5b              pop ebx
  000ee 5f              pop edi
  000ef 5e              pop esi
  000f0 c3              ret
  000f1 90 8d b4 26 00
        00 00 00 8d bc
        27 00 00 00 00  ALIGN     16

_Wildcard_IP ENDP
*/


// Results on my laptop Core 2 T7500 2200MHz:
/*
D:\_KAZE\_KAZE_GOLD\Kazahana_source_EXEs_Benchmark\WildBench>_compile_Intel.bat

D:\_KAZE\_KAZE_GOLD\Kazahana_source_EXEs_Benchmark\WildBench>icl /O3 wildbench.c /FAcs /Fewildbench_Intel12
Intel(R) C++ Compiler XE for applications running on IA-32, Version 12.1.1.258 Build 20111011
Copyright (C) 1985-2011 Intel Corporation.  All rights reserved.

wildbench.c
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.
```

```
-out:wildbench_Intel12.exe
wildbench.obj

D:\_KAZE\_KAZE_GOLD\Kazahana_source_EXEs_Benchmark\WildBench>wildbench_Intel12.exe
Running three times, for charm ...

50000000 runs of WildcardMatch_Recursive_DezhiZhao  = 44.272s, r = 350000000
50000000 runs of WildcardMatch_Iterative_JackHandy  = 15.959s, r = 350000000
50000000 runs of WildcardMatch_Iterative_Kaze       = 18.237s, r = 350000000

50000000 runs of WildcardMatch_Recursive_DezhiZhao  = 46.035s, r = 350000000
50000000 runs of WildcardMatch_Iterative_JackHandy  = 14.711s, r = 350000000
50000000 runs of WildcardMatch_Iterative_Kaze       = 21.403s, r = 350000000

50000000 runs of WildcardMatch_Recursive_DezhiZhao  = 44.164s, r = 350000000
50000000 runs of WildcardMatch_Iterative_JackHandy  = 16.302s, r = 350000000
50000000 runs of WildcardMatch_Iterative_Kaze       = 18.595s, r = 350000000

D:\_KAZE\_KAZE_GOLD\Kazahana_source_EXEs_Benchmark\WildBench>_compile_VS2010.bat

D:\_KAZE\_KAZE_GOLD\Kazahana_source_EXEs_Benchmark\WildBench>cl /Ox wildbench.c /FAcs /Fewildbench_VS2010
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

wildbench.c
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:wildbench_VS2010.exe
wildbench.obj

D:\_KAZE\_KAZE_GOLD\Kazahana_source_EXEs_Benchmark\WildBench>wildbench_VS2010.exe
Running three times, for charm ...

50000000 runs of WildcardMatch_Recursive_DezhiZhao  = 63.835s, r = 350000000
50000000 runs of WildcardMatch_Iterative_JackHandy  = 26.130s, r = 350000000
50000000 runs of WildcardMatch_Iterative_Kaze       = 26.567s, r = 350000000

50000000 runs of WildcardMatch_Recursive_DezhiZhao  = 62.384s, r = 350000000
50000000 runs of WildcardMatch_Iterative_JackHandy  = 26.005s, r = 350000000
50000000 runs of WildcardMatch_Iterative_Kaze       = 26.068s, r = 350000000

50000000 runs of WildcardMatch_Recursive_DezhiZhao  = 63.040s, r = 350000000
50000000 runs of WildcardMatch_Iterative_JackHandy  = 26.036s, r = 350000000
50000000 runs of WildcardMatch_Iterative_Kaze       = 26.333s, r = 350000000

D:\_KAZE\_KAZE_GOLD\Kazahana_source_EXEs_Benchmark\WildBench>
*/

// Igor Pavlov's recursive variant modified (and converted to iterative) by Kaze, 2013-Nov-28:
//static boolean Wildcard_IP(const char *mask, int maskPos, const char *name, int namePos)
//{
//int maskLen = maskGLOBALlen - maskPos;
//int nameLen = nameGLOBALlen - namePos;
//if (maskLen == 0)
//         if (nameLen == 0)
//                   return true;
//         else
//                   return false;
//if (mask[maskPos] == '*') {
//         if (Wildcard_IP(mask, maskPos + 1, name, namePos))
//                   return true;
//         if (nameLen == 0)
//                   return false;
//         return Wildcard_IP(mask, maskPos, name, namePos + 1);
//} else if (mask[maskPos] == '?') {
//         if (nameLen == 0)
//                   return false;
//         return Wildcard_IP(mask, maskPos + 1, name, namePos + 1);
```

```
//} else {
//        if (mask[maskPos] != name[namePos])
//                return false;
//        return Wildcard_IP(mask, maskPos + 1, name, namePos + 1);
//}
//}
int WildcardMatch_Iterative_Kaze(const char* mask, const char* name) {
const char* maskSTACK;
const char* nameSTACK;
int BacktrackFlag = 0;
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
        if (*maskSTACK == '*') {
                mask = maskSTACK+1;
                if (!*mask) return 1;
                name = nameSTACK;
                BacktrackFlag = -1;
                goto Backtrack;
        }
        //else if (*maskSTACK == '?') {
        //} else {
        else if (*maskSTACK != '?') {
                //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                if (*nameSTACK != *maskSTACK) {
                        if (!BacktrackFlag) return 0;
                        name++;
                        goto Backtrack;
                }
        }
}
while (*maskSTACK == '*') ++maskSTACK;
return (!*maskSTACK);
}
/*
; mark_description "Intel(R) C++ Compiler XE for applications running on IA-32, Version 12.1.1.258 Build 20111011";
; mark_description "-O3 -FAcs -Fewildbench";

_WildcardMatch_Iterative_Kaze     PROC NEAR
; parameter 1: 16 + esp
; parameter 2: 20 + esp
.B5.1:

;;; int WildcardMatch_Iterative_Kaze(const char* mask, const char* name) {

  00000 56              push esi
  00001 57              push edi
  00002 56              push esi
  00003 8b 4c 24 14     mov ecx, DWORD PTR [20+esp]
  00007 8b 44 24 10     mov eax, DWORD PTR [16+esp]

;;; const char* maskSTACK;
;;; const char* nameSTACK;
;;; int BacktrackFlag = 0;
;;; Backtrack:
;;; for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {

  0000b 8b f1           mov esi, ecx
  0000d 8b d0           mov edx, eax
  0000f 0f be 39        movsx edi, BYTE PTR [ecx]
  00012 85 ff           test edi, edi
  00014 74 2b           je .B5.9 ; Prob 10%

.B5.2:
  00016 89 1c 24        mov DWORD PTR [esp], ebx
  00019 33 ff           xor edi, edi
  0001b 8b d8           mov ebx, eax

.B5.3:
```

```
;;;        if (*maskSTACK == '*') {

  0001d 0f be 02         movsx eax, BYTE PTR [edx]
  00020 83 f8 2a         cmp eax, 42
  00023 74 4b            je .B5.16 ; Prob 16%

.B5.4:

;;;                    mask = maskSTACK+1;
;;;                    if (!*mask) return 1;
;;;                    name = nameSTACK;
;;;                    BacktrackFlag = -1;
;;;                    goto Backtrack;
;;;        }
;;;        //else if (*maskSTACK == '?') {
;;;        //} else {
;;;        else if (*maskSTACK != '?') {

  00025 83 f8 3f         cmp eax, 63
  00028 74 3b            je .B5.14 ; Prob 20%

.B5.5:

;;;                    //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
;;;                    if (*nameSTACK != *maskSTACK) {

  0002a 3a 06            cmp al, BYTE PTR [esi]
  0002c 74 37            je .B5.14 ; Prob 20%

.B5.6:

;;;                             if (!BacktrackFlag) return 0;

  0002e 85 ff            test edi, edi
  00030 74 5b            je .B5.20 ; Prob 4%

.B5.7:

;;;                             name++;

  00032 41               inc ecx
  00033 8b d3            mov edx, ebx
  00035 8b f1            mov esi, ecx
  00037 0f be 01         movsx eax, BYTE PTR [ecx]
  0003a 85 c0            test eax, eax
  0003c 75 df            jne .B5.3 ; Prob 82%

.B5.8:
  0003e 8b 1c 24         mov ebx, DWORD PTR [esp]

.B5.9:

;;;                             goto Backtrack;
;;;                    }
;;;        }
;;; }
;;; while (*maskSTACK == '*') ++maskSTACK;

  00041 0f be 02         movsx eax, BYTE PTR [edx]
  00044 83 f8 2a         cmp eax, 42
  00047 75 09            jne .B5.13 ; Prob 37%

.B5.11:
  00049 42               inc edx
  0004a 0f be 02         movsx eax, BYTE PTR [edx]
  0004d 83 f8 2a         cmp eax, 42
  00050 74 f7            je .B5.11 ; Prob 82%

.B5.13:
```

```
;;; return (!*maskSTACK);

  00052 ba 01 00 00 00   mov edx, 1
  00057 85 c0            test eax, eax
  00059 b8 00 00 00 00   mov eax, 0
  0005e 0f 44 c2         cmove eax, edx
  00061 59               pop ecx
  00062 5f               pop edi
  00063 5e               pop esi
  00064 c3               ret

.B5.14:
  00065 46               inc esi
  00066 42               inc edx
  00067 0f be 06         movsx eax, BYTE PTR [esi]
  0006a 85 c0            test eax, eax
  0006c 75 af            jne .B5.3 ; Prob 82%
  0006e eb ce            jmp .B5.8 ; Prob 100%

.B5.16:
  00070 8d 5a 01         lea ebx, DWORD PTR [1+edx]
  00073 0f be 52 01      movsx edx, BYTE PTR [1+edx]
  00077 85 d2            test edx, edx
  00079 74 1b            je .B5.21 ; Prob 4%

.B5.17:
  0007b 0f be 3e         movsx edi, BYTE PTR [esi]
  0007e 8b ce            mov ecx, esi
  00080 8b d3            mov edx, ebx
  00082 85 ff            test edi, edi
  00084 74 b8            je .B5.8 ; Prob 18%

.B5.18:
  00086 bf ff ff ff ff   mov edi, -1
  0008b eb 90            jmp .B5.3 ; Prob 100%

.B5.20:
  0008d 8b 1c 24         mov ebx, DWORD PTR [esp]
  00090 33 c0            xor eax, eax
  00092 59               pop ecx
  00093 5f               pop edi
  00094 5e               pop esi
  00095 c3               ret

.B5.21:
  00096 8b 1c 24         mov ebx, DWORD PTR [esp]
  00099 b8 01 00 00 00   mov eax, 1
  0009e 59               pop ecx
  0009f 5f               pop edi
  000a0 5e               pop esi
  000a1 c3               ret
  000a2 8d b4 26 00 00
        00 00 8d bc 27
        00 00 00 00      ALIGN     16

_WildcardMatch_Iterative_Kaze ENDP
*/


// Igor Pavlov's recursive variant modified (and converted to iterative) by Kaze, 2013-Nov-28:
//static boolean Wildcard_IP(const char *mask, int maskPos, const char *name, int namePos)
//{
//int maskLen = maskGLOBALlen - maskPos;
//int nameLen = nameGLOBALlen - namePos;
//if (maskLen == 0)
//         if (nameLen == 0)
//                     return true;
//         else
```

```
//                      return false;
//if (mask[maskPos] == '*') {
//         if (Wildcard_IP(mask, maskPos + 1, name, namePos))
//                      return true;
//         if (nameLen == 0)
//                      return false;
//         return Wildcard_IP(mask, maskPos, name, namePos + 1);
//} else if (mask[maskPos] == '?') {
//         if (nameLen == 0)
//                      return false;
//         return Wildcard_IP(mask, maskPos + 1, name, namePos + 1);
//} else {
//         if (mask[maskPos] != name[namePos])
//                      return false;
//         return Wildcard_IP(mask, maskPos + 1, name, namePos + 1);
//}
//}
int WildcardMatch_Iterative_Kaze1(const char* mask, const char* name) {
// Revision 1:
/*
const char* maskSTACK;
const char* nameSTACK;
int BacktrackFlag = 0;
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
           if (*maskSTACK == '&') {
                      mask = maskSTACK+1;
                      if (!*mask) return 1;
                      name = nameSTACK;
                      BacktrackFlag = -1;
                      goto Backtrack;
           }
           //else if (*maskSTACK == '+') {
           //} else {
           else if (*maskSTACK != '+') {
                      //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                      if (*nameSTACK != *maskSTACK) {
                                 if (!BacktrackFlag) return 0;
                                 name++;
                                 goto Backtrack;
                      }
           }
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
*/
// Revision 2, 2013-Nov-30:
const char* maskSTACK;
const char* nameSTACK;
//int BacktrackFlag = 0; // No need of it in rev.2
/*
    // Simplest heuristic with SUPEROVERHEAD enforced: trying to skip the whole wildcard section by comparing the two arrays order 1 of mask&name.
    int i;
    unsigned char maskOrder1[256];
    unsigned char nameOrder1[256];
    for (i='a'; i <= 'z'; i++) { maskOrder1[i]=0; nameOrder1[i]=0;}
    for (i=0; i < strlen(mask); i++) maskOrder1[mask[i]]=1;
    for (i=0; i < strlen(name); i++) nameOrder1[name[i]]=1;
    // Assuming the incoming strings are already lowercased (as it should for speed) and if we don't have matching alphabet parts (from mask side) means we don't need to compare any further i.e. the match fails.
    for (i='a'; i <= 'z'; i++) if ( maskOrder1[i] == 1 && nameOrder1[i] == 0 ) return 0;
*/
/*
    int i;
    for (i=0; i < strlen(mask); i++) {
        if ( mask[i] != '&' && mask[i] != '+' )
            if ( !memchr(name,mask[i],strlen(name)) ) return 0;
    }
*/
/*
```

```
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
	if (*maskSTACK == '&') {
		mask = maskSTACK+1;
		if (!*mask) return 1;
		name = nameSTACK;
		BacktrackFlag = -1;
		goto Backtrack;
	}
	//else if (*maskSTACK == '+') {
	//} else {
	else if (*maskSTACK != '+') {
		//if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
		if (*nameSTACK != *maskSTACK) {
			if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
			name++;
			goto Backtrack;
		}
	}
}
*/
// Here, outside the main/second 'for', in order to avoid branching we need to set the old/obsolete BacktrackFlag i.e. we need first occurrence of '&':
for (name, mask; *name; ++name, ++mask) {
	if (*mask == '&') {
		goto Backtrack;
	}
	//else if (*mask == '+') {
	//} else {
	else if (*mask != '+') {
		if (*name != *mask) {
			return 0;
		}
	}
}
// We are entering the main/second 'for' with mask pointing to '&' as if BacktrackFlag is already set in the very first iteration at first condition:
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
	if (*maskSTACK == '&') {
		mask = maskSTACK+1;
		if (!*mask) return 1;
		name = nameSTACK;
		//BacktrackFlag = -1;
		goto Backtrack;
	}
	//else if (*maskSTACK == '+') {
	//} else {
	else if (*maskSTACK != '+') {
		//if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
		if (*nameSTACK != *maskSTACK) {
			//if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
			name++;
			goto Backtrack;
		}
	}
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
}
int WildcardMatch_Iterative_Kaze2(const char* mask, const char* name) {
// Revision 1:
/*
const char* maskSTACK;
const char* nameSTACK;
int BacktrackFlag = 0;
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
	if (*maskSTACK == '&') {
		mask = maskSTACK+1;
		if (!*mask) return 1;
```

```
                        name = nameSTACK;
                        BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    if (!BacktrackFlag) return 0;
                                    name++;
                                    goto Backtrack;
                        }
            }
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
*/
// Revision 2, 2013-Nov-30:
const char* maskSTACK;
const char* nameSTACK;
//int BacktrackFlag = 0; // No need of it in rev.2
/*
    // Simplest heuristic with SUPEROVERHEAD enforced: trying to skip the whole wildcard section by comparing the two arrays order 1 of mask&name.
    int i;
    unsigned char maskOrder1[256];
    unsigned char nameOrder1[256];
    for (i='a'; i <= 'z'; i++) { maskOrder1[i]=0; nameOrder1[i]=0;}
    for (i=0; i < strlen(mask); i++) maskOrder1[mask[i]]=1;
    for (i=0; i < strlen(name); i++) nameOrder1[name[i]]=1;
    // Assuming the incoming strings are already lowercased (as it should for speed) and if we don't have matching alphabet parts (from mask side) means we don't need to compare any further i.e. the match fails.
    for (i='a'; i <= 'z'; i++) if ( maskOrder1[i] == 1 && nameOrder1[i] == 0 ) return 0;
*/
/*
    int i;
    for (i=0; i < strlen(mask); i++) {
        if ( mask[i] != '&' && mask[i] != '+' )
            if ( !memchr(name,mask[i],strlen(name)) ) return 0;
    }
*/
/*
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                    name++;
                                    goto Backtrack;
                        }
            }
}
*/
// Here, outside the main/second 'for', in order to avoid branching we need to set the old/obsolete BacktrackFlag i.e. we need first occurrence of '&':
for (name, mask; *name; ++name, ++mask) {
            if (*mask == '&') {
                        goto Backtrack;
            }
            //else if (*mask == '+') {
            //} else {
```

```
		else if (*mask != '+') {
			if (*name != *mask) {
				return 0;
			}
		}
}
// We are entering the main/second 'for' with mask pointing to '&' as if BacktrackFlag is already set in the very first iteration at first condition:
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
		if (*maskSTACK == '&') {
			mask = maskSTACK+1;
			if (!*mask) return 1;
			name = nameSTACK;
			//BacktrackFlag = -1;
			goto Backtrack;
		}
		//else if (*maskSTACK == '+') {
		//} else {
		else if (*maskSTACK != '+') {
			//if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
			if (*nameSTACK != *maskSTACK) {
				//if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
				name++;
				goto Backtrack;
			}
		}
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
}
int WildcardMatch_Iterative_Kaze3(const char* mask, const char* name) {
// Revision 1:
/*
const char* maskSTACK;
const char* nameSTACK;
int BacktrackFlag = 0;
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
		if (*maskSTACK == '&') {
			mask = maskSTACK+1;
			if (!*mask) return 1;
			name = nameSTACK;
			BacktrackFlag = -1;
			goto Backtrack;
		}
		//else if (*maskSTACK == '+') {
		//} else {
		else if (*maskSTACK != '+') {
			//if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
			if (*nameSTACK != *maskSTACK) {
				if (!BacktrackFlag) return 0;
				name++;
				goto Backtrack;
			}
		}
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
*/
// Revision 2, 2013-Nov-30:
const char* maskSTACK;
const char* nameSTACK;
//int BacktrackFlag = 0; // No need of it in rev.2
/*
    // Simplest heuristic with SUPEROVERHEAD enforced: trying to skip the whole wildcard section by comparing the two arrays order 1 of mask&name.
    int i;
    unsigned char maskOrder1[256];
    unsigned char nameOrder1[256];
    for (i='a'; i <= 'z'; i++) { maskOrder1[i]=0; nameOrder1[i]=0;}
```

```
    for (i=0; i < strlen(mask); i++) maskOrder1[mask[i]]=1;
    for (i=0; i < strlen(name); i++) nameOrder1[name[i]]=1;
    // Assuming the incoming strings are already lowercased (as it should for speed) and if we don't have matching alphabet parts (from mask side) means we don't need to compare any further i.e. the match fails.
    for (i='a'; i <= 'z'; i++) if ( maskOrder1[i] == 1 && nameOrder1[i] == 0 ) return 0;
*/
/*
    int i;
    for (i=0; i < strlen(mask); i++) {
        if ( mask[i] != '&' && mask[i] != '+' )
            if ( !memchr(name,mask[i],strlen(name)) ) return 0;
    }
*/
/*
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
           if (*maskSTACK == '&') {
                      mask = maskSTACK+1;
                      if (!*mask) return 1;
                      name = nameSTACK;
                      BacktrackFlag = -1;
                      goto Backtrack;
           }
           //else if (*maskSTACK == '+') {
           //} else {
           else if (*maskSTACK != '+') {
                      //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                      if (*nameSTACK != *maskSTACK) {
                                 if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                 name++;
                                 goto Backtrack;
                      }
           }
}
*/
// Here, outside the main/second 'for', in order to avoid branching we need to set the old/obsolete BacktrackFlag i.e. we need first occurrence of '&':
for (name, mask; *name; ++name, ++mask) {
           if (*mask == '&') {
                      goto Backtrack;
           }
           //else if (*mask == '+') {
           //} else {
           else if (*mask != '+') {
                      if (*name != *mask) {
                                 return 0;
                      }
           }
}
// We are entering the main/second 'for' with mask pointing to '&' as if BacktrackFlag is already set in the very first iteration at first condition:
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
           if (*maskSTACK == '&') {
                      mask = maskSTACK+1;
                      if (!*mask) return 1;
                      name = nameSTACK;
                      //BacktrackFlag = -1;
                      goto Backtrack;
           }
           //else if (*maskSTACK == '+') {
           //} else {
           else if (*maskSTACK != '+') {
                      //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                      if (*nameSTACK != *maskSTACK) {
                                 //if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                 name++;
                                 goto Backtrack;
                      }
           }
}
while (*maskSTACK == '&') ++maskSTACK;
```

```
return (!*maskSTACK);
}
int WildcardMatch_Iterative_Kaze4(const char* mask, const char* name) {
// Revision 1:
/*
const char* maskSTACK;
const char* nameSTACK;
int BacktrackFlag = 0;
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
	if (*maskSTACK == '&') {
		mask = maskSTACK+1;
		if (!*mask) return 1;
		name = nameSTACK;
		BacktrackFlag = -1;
		goto Backtrack;
	}
	//else if (*maskSTACK == '+') {
	//} else {
	else if (*maskSTACK != '+') {
		//if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
		if (*nameSTACK != *maskSTACK) {
			if (!BacktrackFlag) return 0;
			name++;
			goto Backtrack;
		}
	}
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
*/
// Revision 2, 2013-Nov-30:
const char* maskSTACK;
const char* nameSTACK;
//int BacktrackFlag = 0; // No need of it in rev.2
/*
    // Simplest heuristic with SUPEROVERHEAD enforced: trying to skip the whole wildcard section by comparing the two arrays order 1 of mask&name.
    int i;
    unsigned char maskOrder1[256];
    unsigned char nameOrder1[256];
    for (i='a'; i <= 'z'; i++) { maskOrder1[i]=0; nameOrder1[i]=0;}
    for (i=0; i < strlen(mask); i++) maskOrder1[mask[i]]=1;
    for (i=0; i < strlen(name); i++) nameOrder1[name[i]]=1;
    // Assuming the incoming strings are already lowercased (as it should for speed) and if we don't have matching alphabet parts (from mask side) means we don't need to compare any further i.e. the match fails.
    for (i='a'; i <= 'z'; i++) if ( maskOrder1[i] == 1 && nameOrder1[i] == 0 ) return 0;
*/
/*
    int i;
    for (i=0; i < strlen(mask); i++) {
        if ( mask[i] != '&' && mask[i] != '+' )
            if ( !memchr(name,mask[i],strlen(name)) ) return 0;
    }
*/
/*
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
	if (*maskSTACK == '&') {
		mask = maskSTACK+1;
		if (!*mask) return 1;
		name = nameSTACK;
		BacktrackFlag = -1;
		goto Backtrack;
	}
	//else if (*maskSTACK == '+') {
	//} else {
	else if (*maskSTACK != '+') {
		//if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
		if (*nameSTACK != *maskSTACK) {
			if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
```

```
                                name++;
                                goto Backtrack;
                        }
            }
}
*/
// Here, outside the main/second 'for', in order to avoid branching we need to set the old/obsolete BacktrackFlag i.e. we need first occurrence of '&':
for (name, mask; *name; ++name, ++mask) {
            if (*mask == '&') {
                        goto Backtrack;
            }
            //else if (*mask == '+') {
            //} else {
            else if (*mask != '+') {
                        if (*name != *mask) {
                                return 0;
                        }
            }
}
// We are entering the main/second 'for' with mask pointing to '&' as if BacktrackFlag is already set in the very first iteration at first condition:
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        //BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                //if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                name++;
                                goto Backtrack;
                        }
            }
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
}
int WildcardMatch_Iterative_Kaze5(const char* mask, const char* name) {
// Revision 1:
/*
const char* maskSTACK;
const char* nameSTACK;
int BacktrackFlag = 0;
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                if (!BacktrackFlag) return 0;
                                name++;
                                goto Backtrack;
                        }
            }
}
```

```
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
*/
// Revision 2, 2013-Nov-30:
const char* maskSTACK;
const char* nameSTACK;
//int BacktrackFlag = 0; // No need of it in rev.2
/*
    // Simplest heuristic with SUPEROVERHEAD enforced: trying to skip the whole wildcard section by comparing the two arrays order 1 of mask&name.
    int i;
    unsigned char maskOrder1[256];
    unsigned char nameOrder1[256];
    for (i='a'; i <= 'z'; i++) { maskOrder1[i]=0; nameOrder1[i]=0;}
    for (i=0; i < strlen(mask); i++) maskOrder1[mask[i]]=1;
    for (i=0; i < strlen(name); i++) nameOrder1[name[i]]=1;
    // Assuming the incoming strings are already lowercased (as it should for speed) and if we don't have matching alphabet parts (from mask side) means we don't need to compare any further i.e. the match fails.
    for (i='a'; i <= 'z'; i++) if ( maskOrder1[i] == 1 && nameOrder1[i] == 0 ) return 0;
*/
/*
    int i;
    for (i=0; i < strlen(mask); i++) {
        if ( mask[i] != '&' && mask[i] != '+' )
            if ( !memchr(name,mask[i],strlen(name)) ) return 0;
    }
*/
/*
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                    name++;
                                    goto Backtrack;
                        }
            }
}
*/
// Here, outside the main/second 'for', in order to avoid branching we need to set the old/obsolete BacktrackFlag i.e. we need first occurrence of '&':
for (name, mask; *name; ++name, ++mask) {
            if (*mask == '&') {
                        goto Backtrack;
            }
            //else if (*mask == '+') {
            //} else {
            else if (*mask != '+') {
                        if (*name != *mask) {
                                    return 0;
                        }
            }
}
// We are entering the main/second 'for' with mask pointing to '&' as if BacktrackFlag is already set in the very first iteration at first condition:
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        //BacktrackFlag = -1;
                        goto Backtrack;
```

```
		}
		//else if (*maskSTACK == '+') {
		//} else {
		else if (*maskSTACK != '+') {
			//if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
			if (*nameSTACK != *maskSTACK) {
				//if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
				name++;
				goto Backtrack;
			}
		}
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
}
int WildcardMatch_Iterative_Kaze6(const char* mask, const char* name) {
// Revision 1:
/*
const char* maskSTACK;
const char* nameSTACK;
int BacktrackFlag = 0;
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
		if (*maskSTACK == '&') {
			mask = maskSTACK+1;
			if (!*mask) return 1;
			name = nameSTACK;
			BacktrackFlag = -1;
			goto Backtrack;
		}
		//else if (*maskSTACK == '+') {
		//} else {
		else if (*maskSTACK != '+') {
			//if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
			if (*nameSTACK != *maskSTACK) {
				if (!BacktrackFlag) return 0;
				name++;
				goto Backtrack;
			}
		}
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
*/
// Revision 2, 2013-Nov-30:
const char* maskSTACK;
const char* nameSTACK;
//int BacktrackFlag = 0; // No need of it in rev.2
/*
    // Simplest heuristic with SUPEROVERHEAD enforced: trying to skip the whole wildcard section by comparing the two arrays order 1 of mask&name.
    int i;
    unsigned char maskOrder1[256];
    unsigned char nameOrder1[256];
    for (i='a'; i <= 'z'; i++) { maskOrder1[i]=0; nameOrder1[i]=0;}
    for (i=0; i < strlen(mask); i++) maskOrder1[mask[i]]=1;
    for (i=0; i < strlen(name); i++) nameOrder1[name[i]]=1;
    // Assuming the incoming strings are already lowercased (as it should for speed) and if we don't have matching alphabet parts (from mask side) means we don't need to compare any further i.e. the match fails.
    for (i='a'; i <= 'z'; i++) if ( maskOrder1[i] == 1 && nameOrder1[i] == 0 ) return 0;
*/
/*
    int i;
    for (i=0; i < strlen(mask); i++) {
        if ( mask[i] != '&' && mask[i] != '+' )
            if ( !memchr(name,mask[i],strlen(name)) ) return 0;
    }
*/
/*
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
```

```
		if (*maskSTACK == '&') {
			mask = maskSTACK+1;
			if (!*mask) return 1;
			name = nameSTACK;
			BacktrackFlag = -1;
			goto Backtrack;
		}
		//else if (*maskSTACK == '+') {
		//} else {
		else if (*maskSTACK != '+') {
			//if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
			if (*nameSTACK != *maskSTACK) {
				if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
				name++;
				goto Backtrack;
			}
		}
}
*/
// Here, outside the main/second 'for', in order to avoid branching we need to set the old/obsolete BacktrackFlag i.e. we need first occurrence of '&':
for (name, mask; *name; ++name, ++mask) {
		if (*mask == '&') {
			goto Backtrack;
		}
		//else if (*mask == '+') {
		//} else {
		else if (*mask != '+') {
			if (*name != *mask) {
				return 0;
			}
		}
}
// We are entering the main/second 'for' with mask pointing to '&' as if BacktrackFlag is already set in the very first iteration at first condition:
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
		if (*maskSTACK == '&') {
			mask = maskSTACK+1;
			if (!*mask) return 1;
			name = nameSTACK;
			//BacktrackFlag = -1;
			goto Backtrack;
		}
		//else if (*maskSTACK == '+') {
		//} else {
		else if (*maskSTACK != '+') {
			//if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
			if (*nameSTACK != *maskSTACK) {
				//if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
				name++;
				goto Backtrack;
			}
		}
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
}
int WildcardMatch_Iterative_Kaze7(const char* mask, const char* name) {
// Revision 1:
/*
const char* maskSTACK;
const char* nameSTACK;
int BacktrackFlag = 0;
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
		if (*maskSTACK == '&') {
			mask = maskSTACK+1;
			if (!*mask) return 1;
			name = nameSTACK;
			BacktrackFlag = -1;
```

```
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    if (!BacktrackFlag) return 0;
                                    name++;
                                    goto Backtrack;
                        }
            }
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
*/
// Revision 2, 2013-Nov-30:
const char* maskSTACK;
const char* nameSTACK;
//int BacktrackFlag = 0; // No need of it in rev.2
/*
    // Simplest heuristic with SUPEROVERHEAD enforced: trying to skip the whole wildcard section by comparing the two arrays order 1 of mask&name.
    int i;
    unsigned char maskOrder1[256];
    unsigned char nameOrder1[256];
    for (i='a'; i <= 'z'; i++) { maskOrder1[i]=0; nameOrder1[i]=0;}
    for (i=0; i < strlen(mask); i++) maskOrder1[mask[i]]=1;
    for (i=0; i < strlen(name); i++) nameOrder1[name[i]]=1;
    // Assuming the incoming strings are already lowercased (as it should for speed) and if we don't have matching alphabet parts (from mask side) means we don't need to compare any further i.e. the match fails.
    for (i='a'; i <= 'z'; i++) if ( maskOrder1[i] == 1 && nameOrder1[i] == 0 ) return 0;
*/
/*
    int i;
    for (i=0; i < strlen(mask); i++) {
        if ( mask[i] != '&' && mask[i] != '+' )
            if ( !memchr(name,mask[i],strlen(name)) ) return 0;
    }
*/
/*
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                    name++;
                                    goto Backtrack;
                        }
            }
}
*/
// Here, outside the main/second 'for', in order to avoid branching we need to set the old/obsolete BacktrackFlag i.e. we need first occurrence of '&':
for (name, mask; *name; ++name, ++mask) {
            if (*mask == '&') {
                        goto Backtrack;
            }
            //else if (*mask == '+') {
            //} else {
            else if (*mask != '+') {
                        if (*name != *mask) {
```

```
                                return 0;
                        }
            }
}
// We are entering the main/second 'for' with mask pointing to '&' as if BacktrackFlag is already set in the very first iteration at first condition:
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        //BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    //if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                    name++;
                                    goto Backtrack;
                        }
            }
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
}
int WildcardMatch_Iterative_Kaze8(const char* mask, const char* name) {
// Revision 1:
/*
const char* maskSTACK;
const char* nameSTACK;
int BacktrackFlag = 0;
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    if (!BacktrackFlag) return 0;
                                    name++;
                                    goto Backtrack;
                        }
            }
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
*/
// Revision 2, 2013-Nov-30:
const char* maskSTACK;
const char* nameSTACK;
//int BacktrackFlag = 0; // No need of it in rev.2
/*
    // Simplest heuristic with SUPEROVERHEAD enforced: trying to skip the whole wildcard section by comparing the two arrays order 1 of mask&name.
    int i;
    unsigned char maskOrder1[256];
    unsigned char nameOrder1[256];
    for (i='a'; i <= 'z'; i++) { maskOrder1[i]=0; nameOrder1[i]=0;}
    for (i=0; i < strlen(mask); i++) maskOrder1[mask[i]]=1;
    for (i=0; i < strlen(name); i++) nameOrder1[name[i]]=1;
```

```
    // Assuming the incoming strings are already lowercased (as it should for speed) and if we don't have matching alphabet parts (from mask side) means we don't need to compare any further i.e. the match fails.
    for (i='a'; i <= 'z'; i++) if ( maskOrder1[i] == 1 && nameOrder1[i] == 0 ) return 0;
*/
/*
    int i;
    for (i=0; i < strlen(mask); i++) {
        if ( mask[i] != '&' && mask[i] != '+' )
            if ( !memchr(name,mask[i],strlen(name)) ) return 0;
    }
*/
/*
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
           if (*maskSTACK == '&') {
                      mask = maskSTACK+1;
                      if (!*mask) return 1;
                      name = nameSTACK;
                      BacktrackFlag = -1;
                      goto Backtrack;
           }
           //else if (*maskSTACK == '+') {
           //} else {
           else if (*maskSTACK != '+') {
                      //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                      if (*nameSTACK != *maskSTACK) {
                                 if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                 name++;
                                 goto Backtrack;
                      }
           }
}
*/
// Here, outside the main/second 'for', in order to avoid branching we need to set the old/obsolete BacktrackFlag i.e. we need first occurrence of '&':
for (name, mask; *name; ++name, ++mask) {
           if (*mask == '&') {
                      goto Backtrack;
           }
           //else if (*mask == '+') {
           //} else {
           else if (*mask != '+') {
                      if (*name != *mask) {
                                 return 0;
                      }
           }
}
// We are entering the main/second 'for' with mask pointing to '&' as if BacktrackFlag is already set in the very first iteration at first condition:
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
           if (*maskSTACK == '&') {
                      mask = maskSTACK+1;
                      if (!*mask) return 1;
                      name = nameSTACK;
                      //BacktrackFlag = -1;
                      goto Backtrack;
           }
           //else if (*maskSTACK == '+') {
           //} else {
           else if (*maskSTACK != '+') {
                      //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                      if (*nameSTACK != *maskSTACK) {
                                 //if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                 name++;
                                 goto Backtrack;
                      }
           }
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
}
```

```
int WildcardMatch_Iterative_Kaze9(const char* mask, const char* name) {
// Revision 1:
/*
const char* maskSTACK;
const char* nameSTACK;
int BacktrackFlag = 0;
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
	if (*maskSTACK == '&') {
		mask = maskSTACK+1;
		if (!*mask) return 1;
		name = nameSTACK;
		BacktrackFlag = -1;
		goto Backtrack;
	}
	//else if (*maskSTACK == '+') {
	//} else {
	else if (*maskSTACK != '+') {
		//if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
		if (*nameSTACK != *maskSTACK) {
			if (!BacktrackFlag) return 0;
			name++;
			goto Backtrack;
		}
	}
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
*/
// Revision 2, 2013-Nov-30:
const char* maskSTACK;
const char* nameSTACK;
//int BacktrackFlag = 0; // No need of it in rev.2
/*
    // Simplest heuristic with SUPEROVERHEAD enforced: trying to skip the whole wildcard section by comparing the two arrays order 1 of mask&name.
    int i;
    unsigned char maskOrder1[256];
    unsigned char nameOrder1[256];
    for (i='a'; i <= 'z'; i++) { maskOrder1[i]=0; nameOrder1[i]=0;}
    for (i=0; i < strlen(mask); i++) maskOrder1[mask[i]]=1;
    for (i=0; i < strlen(name); i++) nameOrder1[name[i]]=1;
    // Assuming the incoming strings are already lowercased (as it should for speed) and if we don't have matching alphabet parts (from mask side) means we don't need to compare any further i.e. the match fails.
    for (i='a'; i <= 'z'; i++) if ( maskOrder1[i] == 1 && nameOrder1[i] == 0 ) return 0;
*/
/*
    int i;
    for (i=0; i < strlen(mask); i++) {
        if ( mask[i] != '&' && mask[i] != '+' )
            if ( !memchr(name,mask[i],strlen(name)) ) return 0;
    }
*/
/*
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
	if (*maskSTACK == '&') {
		mask = maskSTACK+1;
		if (!*mask) return 1;
		name = nameSTACK;
		BacktrackFlag = -1;
		goto Backtrack;
	}
	//else if (*maskSTACK == '+') {
	//} else {
	else if (*maskSTACK != '+') {
		//if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
		if (*nameSTACK != *maskSTACK) {
			if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
			name++;
			goto Backtrack;
```

```
                        }
            }
}
*/
// Here, outside the main/second 'for', in order to avoid branching we need to set the old/obsolete BacktrackFlag i.e. we need first occurrence of '&':
for (name, mask; *name; ++name, ++mask) {
            if (*mask == '&') {
                        goto Backtrack;
            }
            //else if (*mask == '+') {
            //} else {
            else if (*mask != '+') {
                        if (*name != *mask) {
                                    return 0;
                        }
            }
}
// We are entering the main/second 'for' with mask pointing to '&' as if BacktrackFlag is already set in the very first iteration at first condition:
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        //BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    //if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                    name++;
                                    goto Backtrack;
                        }
            }
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
}
int WildcardMatch_Iterative_Kaze0(const char* mask, const char* name) {
// Revision 1:
/*
const char* maskSTACK;
const char* nameSTACK;
int BacktrackFlag = 0;
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    if (!BacktrackFlag) return 0;
                                    name++;
                                    goto Backtrack;
                        }
            }
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
```

```
*/
// Revision 2, 2013-Nov-30:
const char* maskSTACK;
const char* nameSTACK;
//int BacktrackFlag = 0; // No need of it in rev.2
/*
    // Simplest heuristic with SUPEROVERHEAD enforced: trying to skip the whole wildcard section by comparing the two arrays order 1 of mask&name.
    int i;
    unsigned char maskOrder1[256];
    unsigned char nameOrder1[256];
    for (i='a'; i <= 'z'; i++) { maskOrder1[i]=0; nameOrder1[i]=0;}
    for (i=0; i < strlen(mask); i++) maskOrder1[mask[i]]=1;
    for (i=0; i < strlen(name); i++) nameOrder1[name[i]]=1;
    // Assuming the incoming strings are already lowercased (as it should for speed) and if we don't have matching alphabet parts (from mask side) means we don't need to compare any further i.e. the match fails.
    for (i='a'; i <= 'z'; i++) if ( maskOrder1[i] == 1 && nameOrder1[i] == 0 ) return 0;
*/
/*
    int i;
    for (i=0; i < strlen(mask); i++) {
        if ( mask[i] != '&' && mask[i] != '+' )
            if ( !memchr(name,mask[i],strlen(name)) ) return 0;
    }
*/
/*
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                    name++;
                                    goto Backtrack;
                        }
            }
}
*/
// Here, outside the main/second 'for', in order to avoid branching we need to set the old/obsolete BacktrackFlag i.e. we need first occurrence of '&':
for (name, mask; *name; ++name, ++mask) {
            if (*mask == '&') {
                        goto Backtrack;
            }
            //else if (*mask == '+') {
            //} else {
            else if (*mask != '+') {
                        if (*name != *mask) {
                                    return 0;
                        }
            }
}
// We are entering the main/second 'for' with mask pointing to '&' as if BacktrackFlag is already set in the very first iteration at first condition:
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        //BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
```

```
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    //if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                    name++;
                                    goto Backtrack;
                        }
            }
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
}
int WildcardMatch_Iterative_Kazea(const char* mask, const char* name) {
// Revision 1:
/*
const char* maskSTACK;
const char* nameSTACK;
int BacktrackFlag = 0;
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    if (!BacktrackFlag) return 0;
                                    name++;
                                    goto Backtrack;
                        }
            }
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
*/
// Revision 2, 2013-Nov-30:
const char* maskSTACK;
const char* nameSTACK;
//int BacktrackFlag = 0; // No need of it in rev.2
/*
    // Simplest heuristic with SUPEROVERHEAD enforced: trying to skip the whole wildcard section by comparing the two arrays order 1 of mask&name.
    int i;
    unsigned char maskOrder1[256];
    unsigned char nameOrder1[256];
    for (i='a'; i <= 'z'; i++) { maskOrder1[i]=0; nameOrder1[i]=0;}
    for (i=0; i < strlen(mask); i++) maskOrder1[mask[i]]=1;
    for (i=0; i < strlen(name); i++) nameOrder1[name[i]]=1;
    // Assuming the incoming strings are already lowercased (as it should for speed) and if we don't have matching alphabet parts (from mask side) means we don't need to compare any further i.e. the match fails.
    for (i='a'; i <= 'z'; i++) if ( maskOrder1[i] == 1 && nameOrder1[i] == 0 ) return 0;
*/
/*
    int i;
    for (i=0; i < strlen(mask); i++) {
        if ( mask[i] != '&' && mask[i] != '+' )
            if ( !memchr(name,mask[i],strlen(name)) ) return 0;
    }
*/
/*
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
```

```
                        if (!*mask) return 1;
                        name = nameSTACK;
                        BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                    name++;
                                    goto Backtrack;
                        }
            }
}
*/
// Here, outside the main/second 'for', in order to avoid branching we need to set the old/obsolete BacktrackFlag i.e. we need first occurrence of '&':
for (name, mask; *name; ++name, ++mask) {
            if (*mask == '&') {
                        goto Backtrack;
            }
            //else if (*mask == '+') {
            //} else {
            else if (*mask != '+') {
                        if (*name != *mask) {
                                    return 0;
                        }
            }
}
// We are entering the main/second 'for' with mask pointing to '&' as if BacktrackFlag is already set in the very first iteration at first condition:
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        //BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    //if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                    name++;
                                    goto Backtrack;
                        }
            }
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
}
int WildcardMatch_Iterative_Kazeb(const char* mask, const char* name) {
// Revision 1:
/*
const char* maskSTACK;
const char* nameSTACK;
int BacktrackFlag = 0;
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        BacktrackFlag = -1;
                        goto Backtrack;
            }
```

```
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    if (!BacktrackFlag) return 0;
                                    name++;
                                    goto Backtrack;
                        }
            }
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
*/
// Revision 2, 2013-Nov-30:
const char* maskSTACK;
const char* nameSTACK;
//int BacktrackFlag = 0; // No need of it in rev.2
/*
    // Simplest heuristic with SUPEROVERHEAD enforced: trying to skip the whole wildcard section by comparing the two arrays order 1 of mask&name.
    int i;
    unsigned char maskOrder1[256];
    unsigned char nameOrder1[256];
    for (i='a'; i <= 'z'; i++) { maskOrder1[i]=0; nameOrder1[i]=0;}
    for (i=0; i < strlen(mask); i++) maskOrder1[mask[i]]=1;
    for (i=0; i < strlen(name); i++) nameOrder1[name[i]]=1;
    // Assuming the incoming strings are already lowercased (as it should for speed) and if we don't have matching alphabet parts (from mask side) means we don't need to compare any further i.e. the match fails.
    for (i='a'; i <= 'z'; i++) if ( maskOrder1[i] == 1 && nameOrder1[i] == 0 ) return 0;
*/
/*
    int i;
    for (i=0; i < strlen(mask); i++) {
        if ( mask[i] != '&' && mask[i] != '+' )
            if ( !memchr(name,mask[i],strlen(name)) ) return 0;
    }
*/
/*
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                    name++;
                                    goto Backtrack;
                        }
            }
}
*/
// Here, outside the main/second 'for', in order to avoid branching we need to set the old/obsolete BacktrackFlag i.e. we need first occurrence of '&':
for (name, mask; *name; ++name, ++mask) {
            if (*mask == '&') {
                        goto Backtrack;
            }
            //else if (*mask == '+') {
            //} else {
            else if (*mask != '+') {
                        if (*name != *mask) {
                                    return 0;
                        }
```

```
		}
}
// We are entering the main/second 'for' with mask pointing to '&' as if BacktrackFlag is already set in the very first iteration at first condition:
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
		if (*maskSTACK == '&') {
			mask = maskSTACK+1;
			if (!*mask) return 1;
			name = nameSTACK;
			//BacktrackFlag = -1;
			goto Backtrack;
		}
		//else if (*maskSTACK == '+') {
		//} else {
		else if (*maskSTACK != '+') {
			//if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
			if (*nameSTACK != *maskSTACK) {
				//if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
				name++;
				goto Backtrack;
			}
		}
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
}
int WildcardMatch_Iterative_Kazec(const char* mask, const char* name) {
// Revision 1:
/*
const char* maskSTACK;
const char* nameSTACK;
int BacktrackFlag = 0;
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
		if (*maskSTACK == '&') {
			mask = maskSTACK+1;
			if (!*mask) return 1;
			name = nameSTACK;
			BacktrackFlag = -1;
			goto Backtrack;
		}
		//else if (*maskSTACK == '+') {
		//} else {
		else if (*maskSTACK != '+') {
			//if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
			if (*nameSTACK != *maskSTACK) {
				if (!BacktrackFlag) return 0;
				name++;
				goto Backtrack;
			}
		}
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
*/
// Revision 2, 2013-Nov-30:
const char* maskSTACK;
const char* nameSTACK;
//int BacktrackFlag = 0; // No need of it in rev.2
/*
    // Simplest heuristic with SUPEROVERHEAD enforced: trying to skip the whole wildcard section by comparing the two arrays order 1 of mask&name.
    int i;
    unsigned char maskOrder1[256];
    unsigned char nameOrder1[256];
    for (i='a'; i <= 'z'; i++) { maskOrder1[i]=0; nameOrder1[i]=0;}
    for (i=0; i < strlen(mask); i++) maskOrder1[mask[i]]=1;
    for (i=0; i < strlen(name); i++) nameOrder1[name[i]]=1;
    // Assuming the incoming strings are already lowercased (as it should for speed) and if we don't have matching alphabet parts (from mask side) means we don't need to compare any further i.e. the match fails.
    for (i='a'; i <= 'z'; i++) if ( maskOrder1[i] == 1 && nameOrder1[i] == 0 ) return 0;
```

```
*/
/*
    int i;
    for (i=0; i < strlen(mask); i++) {
        if ( mask[i] != '&' && mask[i] != '+' )
            if ( !memchr(name,mask[i],strlen(name)) ) return 0;
    }
*/
/*
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
           if (*maskSTACK == '&') {
                      mask = maskSTACK+1;
                      if (!*mask) return 1;
                      name = nameSTACK;
                      BacktrackFlag = -1;
                      goto Backtrack;
           }
           //else if (*maskSTACK == '+') {
           //} else {
           else if (*maskSTACK != '+') {
                      //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                      if (*nameSTACK != *maskSTACK) {
                                 if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                 name++;
                                 goto Backtrack;
                      }
           }
}
*/
// Here, outside the main/second 'for', in order to avoid branching we need to set the old/obsolete BacktrackFlag i.e. we need first occurrence of '&':
for (name, mask; *name; ++name, ++mask) {
           if (*mask == '&') {
                      goto Backtrack;
           }
           //else if (*mask == '+') {
           //} else {
           else if (*mask != '+') {
                      if (*name != *mask) {
                                 return 0;
                      }
           }
}
// We are entering the main/second 'for' with mask pointing to '&' as if BacktrackFlag is already set in the very first iteration at first condition:
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
           if (*maskSTACK == '&') {
                      mask = maskSTACK+1;
                      if (!*mask) return 1;
                      name = nameSTACK;
                      //BacktrackFlag = -1;
                      goto Backtrack;
           }
           //else if (*maskSTACK == '+') {
           //} else {
           else if (*maskSTACK != '+') {
                      //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                      if (*nameSTACK != *maskSTACK) {
                                 //if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                 name++;
                                 goto Backtrack;
                      }
           }
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
}
int WildcardMatch_Iterative_Kazed(const char* mask, const char* name) {
// Revision 1:
```

```
/*
const char* maskSTACK;
const char* nameSTACK;
int BacktrackFlag = 0;
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
	if (*maskSTACK == '&') {
		mask = maskSTACK+1;
		if (!*mask) return 1;
		name = nameSTACK;
		BacktrackFlag = -1;
		goto Backtrack;
	}
	//else if (*maskSTACK == '+') {
	//} else {
	else if (*maskSTACK != '+') {
		//if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
		if (*nameSTACK != *maskSTACK) {
			if (!BacktrackFlag) return 0;
			name++;
			goto Backtrack;
		}
	}
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
*/
// Revision 2, 2013-Nov-30:
const char* maskSTACK;
const char* nameSTACK;
//int BacktrackFlag = 0; // No need of it in rev.2
/*
    // Simplest heuristic with SUPEROVERHEAD enforced: trying to skip the whole wildcard section by comparing the two arrays order 1 of mask&name.
    int i;
    unsigned char maskOrder1[256];
    unsigned char nameOrder1[256];
    for (i='a'; i <= 'z'; i++) { maskOrder1[i]=0; nameOrder1[i]=0;}
    for (i=0; i < strlen(mask); i++) maskOrder1[mask[i]]=1;
    for (i=0; i < strlen(name); i++) nameOrder1[name[i]]=1;
    // Assuming the incoming strings are already lowercased (as it should for speed) and if we don't have matching alphabet parts (from mask side) means we don't need to compare any further i.e. the match fails.
    for (i='a'; i <= 'z'; i++) if ( maskOrder1[i] == 1 && nameOrder1[i] == 0 ) return 0;
*/
/*
    int i;
    for (i=0; i < strlen(mask); i++) {
        if ( mask[i] != '&' && mask[i] != '+' )
            if ( !memchr(name,mask[i],strlen(name)) ) return 0;
    }
*/
/*
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
	if (*maskSTACK == '&') {
		mask = maskSTACK+1;
		if (!*mask) return 1;
		name = nameSTACK;
		BacktrackFlag = -1;
		goto Backtrack;
	}
	//else if (*maskSTACK == '+') {
	//} else {
	else if (*maskSTACK != '+') {
		//if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
		if (*nameSTACK != *maskSTACK) {
			if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
			name++;
			goto Backtrack;
		}
	}
```

```
}
*/
// Here, outside the main/second 'for', in order to avoid branching we need to set the old/obsolete BacktrackFlag i.e. we need first occurrence of '&':
for (name, mask; *name; ++name, ++mask) {
	if (*mask == '&') {
		goto Backtrack;
	}
	//else if (*mask == '+') {
	//} else {
	else if (*mask != '+') {
		if (*name != *mask) {
			return 0;
		}
	}
}
// We are entering the main/second 'for' with mask pointing to '&' as if BacktrackFlag is already set in the very first iteration at first condition:
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
	if (*maskSTACK == '&') {
		mask = maskSTACK+1;
		if (!*mask) return 1;
		name = nameSTACK;
		//BacktrackFlag = -1;
		goto Backtrack;
	}
	//else if (*maskSTACK == '+') {
	//} else {
	else if (*maskSTACK != '+') {
		//if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
		if (*nameSTACK != *maskSTACK) {
			//if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
			name++;
			goto Backtrack;
		}
	}
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
}
int WildcardMatch_Iterative_Kazee(const char* mask, const char* name) {
// Revision 1:
/*
const char* maskSTACK;
const char* nameSTACK;
int BacktrackFlag = 0;
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
	if (*maskSTACK == '&') {
		mask = maskSTACK+1;
		if (!*mask) return 1;
		name = nameSTACK;
		BacktrackFlag = -1;
		goto Backtrack;
	}
	//else if (*maskSTACK == '+') {
	//} else {
	else if (*maskSTACK != '+') {
		//if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
		if (*nameSTACK != *maskSTACK) {
			if (!BacktrackFlag) return 0;
			name++;
			goto Backtrack;
		}
	}
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
*/
// Revision 2, 2013-Nov-30:
```

```
const char* maskSTACK;
const char* nameSTACK;
//int BacktrackFlag = 0; // No need of it in rev.2
/*
    // Simplest heuristic with SUPEROVERHEAD enforced: trying to skip the whole wildcard section by comparing the two arrays order 1 of mask&name.
    int i;
    unsigned char maskOrder1[256];
    unsigned char nameOrder1[256];
    for (i='a'; i <= 'z'; i++) { maskOrder1[i]=0; nameOrder1[i]=0;}
    for (i=0; i < strlen(mask); i++) maskOrder1[mask[i]]=1;
    for (i=0; i < strlen(name); i++) nameOrder1[name[i]]=1;
    // Assuming the incoming strings are already lowercased (as it should for speed) and if we don't have matching alphabet parts (from mask side) means we don't need to compare any further i.e. the match fails.
    for (i='a'; i <= 'z'; i++) if ( maskOrder1[i] == 1 && nameOrder1[i] == 0 ) return 0;
*/
/*
    int i;
    for (i=0; i < strlen(mask); i++) {
        if ( mask[i] != '&' && mask[i] != '+' )
            if ( !memchr(name,mask[i],strlen(name)) ) return 0;
    }
*/
/*
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                    name++;
                                    goto Backtrack;
                        }
            }
}
*/
// Here, outside the main/second 'for', in order to avoid branching we need to set the old/obsolete BacktrackFlag i.e. we need first occurrence of '&':
for (name, mask; *name; ++name, ++mask) {
            if (*mask == '&') {
                        goto Backtrack;
            }
            //else if (*mask == '+') {
            //} else {
            else if (*mask != '+') {
                        if (*name != *mask) {
                                    return 0;
                        }
            }
}
// We are entering the main/second 'for' with mask pointing to '&' as if BacktrackFlag is already set in the very first iteration at first condition:
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        //BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
```

```
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    //if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                    name++;
                                    goto Backtrack;
                        }
            }
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
}
int WildcardMatch_Iterative_Kazef(const char* mask, const char* name) {
// Revision 1:
/*
const char* maskSTACK;
const char* nameSTACK;
int BacktrackFlag = 0;
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    if (!BacktrackFlag) return 0;
                                    name++;
                                    goto Backtrack;
                        }
            }
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
*/
// Revision 2, 2013-Nov-30:
const char* maskSTACK;
const char* nameSTACK;
//int BacktrackFlag = 0; // No need of it in rev.2
/*
    // Simplest heuristic with SUPEROVERHEAD enforced: trying to skip the whole wildcard section by comparing the two arrays order 1 of mask&name.
    int i;
    unsigned char maskOrder1[256];
    unsigned char nameOrder1[256];
    for (i='a'; i <= 'z'; i++) { maskOrder1[i]=0; nameOrder1[i]=0;}
    for (i=0; i < strlen(mask); i++) maskOrder1[mask[i]]=1;
    for (i=0; i < strlen(name); i++) nameOrder1[name[i]]=1;
    // Assuming the incoming strings are already lowercased (as it should for speed) and if we don't have matching alphabet parts (from mask side) means we don't need to compare any further i.e. the match fails.
    for (i='a'; i <= 'z'; i++) if ( maskOrder1[i] == 1 && nameOrder1[i] == 0 ) return 0;
*/
/*
    int i;
    for (i=0; i < strlen(mask); i++) {
        if ( mask[i] != '&' && mask[i] != '+' )
            if ( !memchr(name,mask[i],strlen(name)) ) return 0;
    }
*/
/*
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
```

```
                        BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                    name++;
                                    goto Backtrack;
                        }
            }
}
*/
// Here, outside the main/second 'for', in order to avoid branching we need to set the old/obsolete BacktrackFlag i.e. we need first occurrence of '&':
for (name, mask; *name; ++name, ++mask) {
            if (*mask == '&') {
                        goto Backtrack;
            }
            //else if (*mask == '+') {
            //} else {
            else if (*mask != '+') {
                        if (*name != *mask) {
                                    return 0;
                        }
            }
}
// We are entering the main/second 'for' with mask pointing to '&' as if BacktrackFlag is already set in the very first iteration at first condition:
Backtrack:
for (nameSTACK = name, maskSTACK = mask; *nameSTACK; ++nameSTACK, ++maskSTACK) {
            if (*maskSTACK == '&') {
                        mask = maskSTACK+1;
                        if (!*mask) return 1;
                        name = nameSTACK;
                        //BacktrackFlag = -1;
                        goto Backtrack;
            }
            //else if (*maskSTACK == '+') {
            //} else {
            else if (*maskSTACK != '+') {
                        //if (tolower(*nameSTACK) != tolower(*maskSTACK)) { // These 'tolower's are outrageous, they hurt speed BADLY, in real-world usage both should have been lowercased outwith the 'for'.
                        if (*nameSTACK != *maskSTACK) {
                                    //if (!BacktrackFlag) return 0; // Stupid branching, SLOW!
                                    name++;
                                    goto Backtrack;
                        }
            }
}
while (*maskSTACK == '&') ++maskSTACK;
return (!*maskSTACK);
}
```

```
static boolean EnhancedMaskTest_OrEmpty_AndNotEmpty_1(const char *mask, int maskPos, const char *name, int namePos)
{
  char maskChar;
  char c;
  //int maskLen = KAZE_strlen(mask) - maskPos;
  //int nameLen = KAZE_strlenLF(name) - namePos;
  // Above 2 lines are modified with GLOBAL variables for speed as follows:
  int maskLen = maskGLOBALlen - maskPos; // for speed up ((12415s - 755s)/755)*100% = 1544%
  int nameLen = nameGLOBALlen1 - namePos;
  if (maskLen == 0)
```

```
    if (nameLen == 0)
      return true;
    else
      return false;
  maskChar = mask[maskPos];
  if (maskChar == '@') // or empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask, maskPos + 1, name, namePos)) // KAZE: THIS LINE DECIDES whether 'or empty' or 'and not empty'
      return true;                                                  //       uncommented is 'or empty'
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '#') // and not empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty},
  else if(maskChar == '|') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '~') // and not empty AND NOT ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
  else if(maskChar == '^') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
```

```
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '$') // and not empty AND ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask,  maskPos + 1, name, namePos + 1);
  }


  else if(maskChar == '`') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask,  maskPos, name, namePos + 1);
  }
  else if(maskChar == '.') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask,  maskPos, name, namePos + 1);
  }


  else if(maskChar == '*')
  {
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask, maskPos + 1, name, namePos))
      return true;
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask, maskPos, name, namePos + 1);
  }
  else
  {
    c = name[namePos];
    //if (maskChar != c)
                      if (CaseSensitiveWildcardMatching_flag == 0) {
      if (KAZE_toupper(maskChar) != KAZE_toupper(c))
        return false;
                      } else {
      if ((maskChar) != (c))
        return false;
                      }
```

```
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask,  maskPos + 1, name, namePos + 1);
  }
}

static boolean EnhancedMaskTest_OrEmpty_AndNotEmpty_2(const char *mask, int maskPos, const char *name, int namePos)
{
  char maskChar;
  char c;
  //int maskLen = KAZE_strlen(mask) - maskPos;
  //int nameLen = KAZE_strlenLF(name) - namePos;
  // Above 2 lines are modified with GLOBAL variables for speed as follows:
  int maskLen = maskGLOBALlen - maskPos; // for speed up ((12415s - 755s)/755)*100% = 1544%
  int nameLen = nameGLOBALlen2 - namePos;
  if (maskLen == 0)
    if (nameLen == 0)
      return true;
    else
      return false;
  maskChar = mask[maskPos];
  if (maskChar == '@') // or empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask, maskPos + 1, name, namePos)) // KAZE: THIS LINE DECIDES whether 'or empty' or 'and not empty'
      return true;                                                  //       uncommented is 'or empty'
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '#') // and not empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask,  maskPos + 1, name, namePos + 1);
  }
//       wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty},
  else if(maskChar == '|') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '~') // and not empty AND NOT ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
```

```
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
  else if(maskChar == '^') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '$') // and not empty AND ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask,  maskPos + 1, name, namePos + 1);
  }


  else if(maskChar == '`') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask,  maskPos, name, namePos + 1);
  }
  else if(maskChar == '.') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask,  maskPos, name, namePos + 1);
  }


  else if(maskChar == '*')
  {
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask, maskPos + 1, name, namePos))
      return true;
    if (nameLen == 0)
```

```
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask, maskPos, name, namePos + 1);
  }
  else
  {
    c = name[namePos];
    //if (maskChar != c)
                      if (CaseSensitiveWildcardMatching_flag == 0) {
      if (KAZE_toupper(maskChar) != KAZE_toupper(c))
        return false;
                      } else {
      if ((maskChar) != (c))
        return false;
                      }
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask,  maskPos + 1, name, namePos + 1);
  }
}

static boolean EnhancedMaskTest_OrEmpty_AndNotEmpty_3(const char *mask, int maskPos, const char *name, int namePos)
{
  char maskChar;
  char c;
  //int maskLen = KAZE_strlen(mask) - maskPos;
  //int nameLen = KAZE_strlenLF(name) - namePos;
  // Above 2 lines are modified with GLOBAL variables for speed as follows:
  int maskLen = maskGLOBALlen - maskPos; // for speed up ((12415s - 755s)/755)*100% = 1544%
  int nameLen = nameGLOBALlen3 - namePos;
  if (maskLen == 0)
    if (nameLen == 0)
      return true;
    else
      return false;
  maskChar = mask[maskPos];
  if (maskChar == '@') // or empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask, maskPos + 1, name, namePos)) // KAZE: THIS LINE DECIDES whether 'or empty' or 'and not empty'
      return true;                                                  //       uncommented is 'or empty'
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '#') // and not empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask,  maskPos + 1, name, namePos + 1);
  }
//       wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty},
  else if(maskChar == '|') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask,  maskPos + 1, name, namePos + 1);
```

```
  }
  else if(maskChar == '~') // and not empty AND NOT ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
  else if(maskChar == '^') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '$') // and not empty AND ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask,  maskPos + 1, name, namePos + 1);
  }


  else if(maskChar == '`') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask,  maskPos, name, namePos + 1);
  }
  else if(maskChar == '.') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
```

```
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask,  maskPos, name, namePos + 1);
  }


  else if(maskChar == '*')
  {
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask, maskPos + 1, name, namePos))
      return true;
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask, maskPos, name, namePos + 1);
  }
  else
  {
    c = name[namePos];
    //if (maskChar != c)
                      if (CaseSensitiveWildcardMatching_flag == 0) {
      if (KAZE_toupper(maskChar) != KAZE_toupper(c))
        return false;
                      } else {
      if ((maskChar) != (c))
        return false;
                      }
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask,  maskPos + 1, name, namePos + 1);
  }
}

static boolean EnhancedMaskTest_OrEmpty_AndNotEmpty_4(const char *mask, int maskPos, const char *name, int namePos)
{
  char maskChar;
  char c;
  //int maskLen = KAZE_strlen(mask) - maskPos;
  //int nameLen = KAZE_strlenLF(name) - namePos;
  // Above 2 lines are modified with GLOBAL variables for speed as follows:
  int maskLen = maskGLOBALlen - maskPos; // for speed up ((12415s - 755s)/755)*100% = 1544%
  int nameLen = nameGLOBALlen4 - namePos;
  if (maskLen == 0)
    if (nameLen == 0)
      return true;
    else
      return false;
  maskChar = mask[maskPos];
  if (maskChar == '@') // or empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask, maskPos + 1, name, namePos)) // KAZE: THIS LINE DECIDES whether 'or empty' or 'and not empty'
      return true;                                                  //       uncommented is 'or empty'
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '#') // and not empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask,  maskPos + 1, name, namePos + 1);
  }
```

```
//      wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty},
  else if(maskChar == '|') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '~') // and not empty AND NOT ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
  else if(maskChar == '^') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '$') // and not empty AND ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask,  maskPos + 1, name, namePos + 1);
  }


  else if(maskChar == '`') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
```

```
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask,  maskPos, name, namePos + 1);
  }
  else if(maskChar == '.') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask,  maskPos, name, namePos + 1);
  }


  else if(maskChar == '*')
  {
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask, maskPos + 1, name, namePos))
      return true;
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask, maskPos, name, namePos + 1);
  }
  else
  {
    c = name[namePos];
    //if (maskChar != c)
                      if (CaseSensitiveWildcardMatching_flag == 0) {
      if (KAZE_toupper(maskChar) != KAZE_toupper(c))
        return false;
                      } else {
      if ((maskChar) != (c))
        return false;
                      }
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask,  maskPos + 1, name, namePos + 1);
  }
}

static boolean EnhancedMaskTest_OrEmpty_AndNotEmpty_5(const char *mask, int maskPos, const char *name, int namePos)
{
  char maskChar;
  char c;
  //int maskLen = KAZE_strlen(mask) - maskPos;
  //int nameLen = KAZE_strlenLF(name) - namePos;
  // Above 2 lines are modified with GLOBAL variables for speed as follows:
  int maskLen = maskGLOBALlen - maskPos; // for speed up ((12415s - 755s)/755)*100% = 1544%
  int nameLen = nameGLOBALlen5 - namePos;
  if (maskLen == 0)
    if (nameLen == 0)
      return true;
    else
      return false;
  maskChar = mask[maskPos];
  if (maskChar == '@') // or empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask, maskPos + 1, name, namePos)) // KAZE: THIS LINE DECIDES whether 'or empty' or 'and not empty'
      return true;                                                  //       uncommented is 'or empty'
```

```
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '#') // and not empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty},
  else if(maskChar == '|') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '~') // and not empty AND NOT ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
  else if(maskChar == '^') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '$') // and not empty AND ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;
```

```
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask,  maskPos + 1, name, namePos + 1);
  }


  else if(maskChar == '`') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask,  maskPos, name, namePos + 1);
  }
  else if(maskChar == '.') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask,  maskPos, name, namePos + 1);
  }


  else if(maskChar == '*')
  {
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask, maskPos + 1, name, namePos))
      return true;
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask, maskPos, name, namePos + 1);
  }
  else
  {
    c = name[namePos];
    //if (maskChar != c)
                    if (CaseSensitiveWildcardMatching_flag == 0) {
      if (KAZE_toupper(maskChar) != KAZE_toupper(c))
        return false;
                    } else {
      if ((maskChar) != (c))
        return false;
                    }
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask,  maskPos + 1, name, namePos + 1);
  }
}

static boolean EnhancedMaskTest_OrEmpty_AndNotEmpty_6(const char *mask, int maskPos, const char *name, int namePos)
{
  char maskChar;
  char c;
  //int maskLen = KAZE_strlen(mask) - maskPos;
  //int nameLen = KAZE_strlenLF(name) - namePos;
  // Above 2 lines are modified with GLOBAL variables for speed as follows:
  int maskLen = maskGLOBALlen - maskPos; // for speed up ((12415s - 755s)/755)*100% = 1544%
  int nameLen = nameGLOBALlen6 - namePos;
```

```
  if (maskLen == 0)
    if (nameLen == 0)
      return true;
    else
      return false;
  maskChar = mask[maskPos];
  if (maskChar == '@') // or empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask, maskPos + 1, name, namePos)) // KAZE: THIS LINE DECIDES whether 'or empty' or 'and not empty'
      return true;                                                  //       uncommented is 'or empty'
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '#') // and not empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty},
  else if(maskChar == '|') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '~') // and not empty AND NOT ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
  else if(maskChar == '^') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;
```

```
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '$') // and not empty AND ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask,  maskPos + 1, name, namePos + 1);
  }


  else if(maskChar == '`') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask,  maskPos, name, namePos + 1);
  }
  else if(maskChar == '.') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask,  maskPos, name, namePos + 1);
  }


  else if(maskChar == '*')
  {
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask, maskPos + 1, name, namePos))
      return true;
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask, maskPos, name, namePos + 1);
  }
  else
  {
    c = name[namePos];
    //if (maskChar != c)
                      if (CaseSensitiveWildcardMatching_flag == 0) {
      if (KAZE_toupper(maskChar) != KAZE_toupper(c))
        return false;
                      } else {
      if ((maskChar) != (c))
        return false;
```

```
                      }
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask,  maskPos + 1, name, namePos + 1);
  }
}

static boolean EnhancedMaskTest_OrEmpty_AndNotEmpty_7(const char *mask, int maskPos, const char *name, int namePos)
{
  char maskChar;
  char c;
  //int maskLen = KAZE_strlen(mask) - maskPos;
  //int nameLen = KAZE_strlenLF(name) - namePos;
  // Above 2 lines are modified with GLOBAL variables for speed as follows:
  int maskLen = maskGLOBALlen - maskPos; // for speed up ((12415s - 755s)/755)*100% = 1544%
  int nameLen = nameGLOBALlen7 - namePos;
  if (maskLen == 0)
    if (nameLen == 0)
      return true;
    else
      return false;
  maskChar = mask[maskPos];
  if (maskChar == '@') // or empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask, maskPos + 1, name, namePos)) // KAZE: THIS LINE DECIDES whether 'or empty' or 'and not empty'
      return true;                                                  //       uncommented is 'or empty'
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '#') // and not empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty},
  else if(maskChar == '|') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '~') // and not empty AND NOT ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
```

```
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask,  maskPos + 1, name, namePos + 1);
  }
//       wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
  else if(maskChar == '^') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '$') // and not empty AND ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask,  maskPos + 1, name, namePos + 1);
  }


  else if(maskChar == '`') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask,  maskPos, name, namePos + 1);
  }
  else if(maskChar == '.') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask,  maskPos, name, namePos + 1);
  }


  else if(maskChar == '*')
  {
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask, maskPos + 1, name, namePos))
      return true;
```

```
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask, maskPos, name, namePos + 1);
  }
  else
  {
    c = name[namePos];
    //if (maskChar != c)
                      if (CaseSensitiveWildcardMatching_flag == 0) {
      if (KAZE_toupper(maskChar) != KAZE_toupper(c))
        return false;
                      } else {
      if ((maskChar) != (c))
        return false;
                      }
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask,  maskPos + 1, name, namePos + 1);
  }
}

static boolean EnhancedMaskTest_OrEmpty_AndNotEmpty_8(const char *mask, int maskPos, const char *name, int namePos)
{
  char maskChar;
  char c;
  //int maskLen = KAZE_strlen(mask) - maskPos;
  //int nameLen = KAZE_strlenLF(name) - namePos;
  // Above 2 lines are modified with GLOBAL variables for speed as follows:
  int maskLen = maskGLOBALlen - maskPos; // for speed up ((12415s - 755s)/755)*100% = 1544%
  int nameLen = nameGLOBALlen8 - namePos;
  if (maskLen == 0)
    if (nameLen == 0)
      return true;
    else
      return false;
  maskChar = mask[maskPos];
  if (maskChar == '@') // or empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask, maskPos + 1, name, namePos)) // KAZE: THIS LINE DECIDES whether 'or empty' or 'and not empty'
      return true;                                                  //       uncommented is 'or empty'
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '#') // and not empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask,  maskPos + 1, name, namePos + 1);
  }
//       wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty},
  else if(maskChar == '|') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
```

```
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '~') // and not empty AND NOT ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
  else if(maskChar == '^') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '$') // and not empty AND ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask,  maskPos + 1, name, namePos + 1);
  }


  else if(maskChar == '`') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask,  maskPos, name, namePos + 1);
  }
  else if(maskChar == '.') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask, maskPos + 1, name, namePos))
      return true;
```

```
    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask,  maskPos, name, namePos + 1);
  }


  else if(maskChar == '*')
  {
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask, maskPos + 1, name, namePos))
      return true;
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask, maskPos, name, namePos + 1);
  }
  else
  {
    c = name[namePos];
    //if (maskChar != c)
                      if (CaseSensitiveWildcardMatching_flag == 0) {
      if (KAZE_toupper(maskChar) != KAZE_toupper(c))
        return false;
                      } else {
      if ((maskChar) != (c))
        return false;
                      }
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask,  maskPos + 1, name, namePos + 1);
  }
}

static boolean EnhancedMaskTest_OrEmpty_AndNotEmpty_9(const char *mask, int maskPos, const char *name, int namePos)
{
  char maskChar;
  char c;
  //int maskLen = KAZE_strlen(mask) - maskPos;
  //int nameLen = KAZE_strlenLF(name) - namePos;
  // Above 2 lines are modified with GLOBAL variables for speed as follows:
  int maskLen = maskGLOBALlen - maskPos; // for speed up ((12415s - 755s)/755)*100% = 1544%
  int nameLen = nameGLOBALlen9 - namePos;
  if (maskLen == 0)
    if (nameLen == 0)
      return true;
    else
      return false;
  maskChar = mask[maskPos];
  if (maskChar == '@') // or empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask, maskPos + 1, name, namePos)) // KAZE: THIS LINE DECIDES whether 'or empty' or 'and not empty'
      return true;                                                  //       uncommented is 'or empty'
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '#') // and not empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask,  maskPos + 1, name, namePos + 1);
```

```
  }
//      wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty},
  else if(maskChar == '|') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '~') // and not empty AND NOT ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
  else if(maskChar == '^') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '$') // and not empty AND ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask,  maskPos + 1, name, namePos + 1);
  }


  else if(maskChar == '`') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask, maskPos + 1, name, namePos))
      return true;
```

```
    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask,  maskPos, name, namePos + 1);
  }
  else if(maskChar == '.') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask,  maskPos, name, namePos + 1);
  }


  else if(maskChar == '*')
  {
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask, maskPos + 1, name, namePos))
      return true;
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask, maskPos, name, namePos + 1);
  }
  else
  {
    c = name[namePos];
    //if (maskChar != c)
                      if (CaseSensitiveWildcardMatching_flag == 0) {
      if (KAZE_toupper(maskChar) != KAZE_toupper(c))
        return false;
                      } else {
      if ((maskChar) != (c))
        return false;
                      }
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask,  maskPos + 1, name, namePos + 1);
  }
}

static boolean EnhancedMaskTest_OrEmpty_AndNotEmpty_0(const char *mask, int maskPos, const char *name, int namePos)
{
  char maskChar;
  char c;
  //int maskLen = KAZE_strlen(mask) - maskPos;
  //int nameLen = KAZE_strlenLF(name) - namePos;
  // Above 2 lines are modified with GLOBAL variables for speed as follows:
  int maskLen = maskGLOBALlen - maskPos; // for speed up ((12415s - 755s)/755)*100% = 1544%
  int nameLen = nameGLOBALlen0 - namePos;
  if (maskLen == 0)
    if (nameLen == 0)
      return true;
    else
      return false;
  maskChar = mask[maskPos];
  if (maskChar == '@') // or empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask, maskPos + 1, name, namePos)) // KAZE: THIS LINE DECIDES whether 'or empty' or 'and not empty'
```

```
      return true;                                              //       uncommented is 'or empty'
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '#') // and not empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask,  maskPos + 1, name, namePos + 1);
  }
//       wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty},
  else if(maskChar == '|') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '~') // and not empty AND NOT ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask,  maskPos + 1, name, namePos + 1);
  }
//       wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
  else if(maskChar == '^') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '$') // and not empty AND ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
```

```
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask,  maskPos + 1, name, namePos + 1);
  }


  else if(maskChar == '`') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask,  maskPos, name, namePos + 1);
  }
  else if(maskChar == '.') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask,  maskPos, name, namePos + 1);
  }


  else if(maskChar == '*')
  {
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask, maskPos + 1, name, namePos))
      return true;
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask, maskPos, name, namePos + 1);
  }
  else
  {
    c = name[namePos];
    //if (maskChar != c)
                      if (CaseSensitiveWildcardMatching_flag == 0) {
      if (KAZE_toupper(maskChar) != KAZE_toupper(c))
        return false;
                      } else {
      if ((maskChar) != (c))
        return false;
                      }
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask,  maskPos + 1, name, namePos + 1);
  }
}

static boolean EnhancedMaskTest_OrEmpty_AndNotEmpty_a(const char *mask, int maskPos, const char *name, int namePos)
{
  char maskChar;
  char c;
  //int maskLen = KAZE_strlen(mask) - maskPos;
  //int nameLen = KAZE_strlenLF(name) - namePos;
  // Above 2 lines are modified with GLOBAL variables for speed as follows:
  int maskLen = maskGLOBALlen - maskPos; // for speed up ((12415s - 755s)/755)*100% = 1544%
```

```
  int nameLen = nameGLOBALlena - namePos;
  if (maskLen == 0)
    if (nameLen == 0)
      return true;
    else
      return false;
  maskChar = mask[maskPos];
  if (maskChar == '@') // or empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask, maskPos + 1, name, namePos)) // KAZE: THIS LINE DECIDES whether 'or empty' or 'and not empty'
      return true;                                                    //       uncommented is 'or empty'
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '#') // and not empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask,  maskPos + 1, name, namePos + 1);
  }
//       wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty},
  else if(maskChar == '|') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '~') // and not empty AND NOT ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask,  maskPos + 1, name, namePos + 1);
  }
//       wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
  else if(maskChar == '^') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;
```

```
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '$') // and not empty AND ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask,  maskPos + 1, name, namePos + 1);
  }


  else if(maskChar == '`') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask,  maskPos, name, namePos + 1);
  }
  else if(maskChar == '.') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask,  maskPos, name, namePos + 1);
  }


  else if(maskChar == '*')
  {
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask, maskPos + 1, name, namePos))
      return true;
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask, maskPos, name, namePos + 1);
  }
  else
  {
    c = name[namePos];
    //if (maskChar != c)
                      if (CaseSensitiveWildcardMatching_flag == 0) {
      if (KAZE_toupper(maskChar) != KAZE_toupper(c))
        return false;
                      } else {
      if ((maskChar) != (c))
```

```
        return false;
                    }
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask,  maskPos + 1, name, namePos + 1);
  }
}

static boolean EnhancedMaskTest_OrEmpty_AndNotEmpty_b(const char *mask, int maskPos, const char *name, int namePos)
{
  char maskChar;
  char c;
  //int maskLen = KAZE_strlen(mask) - maskPos;
  //int nameLen = KAZE_strlenLF(name) - namePos;
  // Above 2 lines are modified with GLOBAL variables for speed as follows:
  int maskLen = maskGLOBALlen - maskPos; // for speed up ((12415s - 755s)/755)*100% = 1544%
  int nameLen = nameGLOBALlenb - namePos;
  if (maskLen == 0)
    if (nameLen == 0)
      return true;
    else
      return false;
  maskChar = mask[maskPos];
  if (maskChar == '@') // or empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask, maskPos + 1, name, namePos)) // KAZE: THIS LINE DECIDES whether 'or empty' or 'and not empty'
      return true;                                                  //       uncommented is 'or empty'
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '#') // and not empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask,  maskPos + 1, name, namePos + 1);
  }
//       wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty},
  else if(maskChar == '|') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '~') // and not empty AND NOT ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;
```

```
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
  else if(maskChar == '^') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '$') // and not empty AND ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask,  maskPos + 1, name, namePos + 1);
  }


  else if(maskChar == '`') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask,  maskPos, name, namePos + 1);
  }
  else if(maskChar == '.') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask,  maskPos, name, namePos + 1);
  }


  else if(maskChar == '*')
  {
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask, maskPos + 1, name, namePos))
```

```
      return true;
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask, maskPos, name, namePos + 1);
  }
  else
  {
    c = name[namePos];
    //if (maskChar != c)
                      if (CaseSensitiveWildcardMatching_flag == 0) {
      if (KAZE_toupper(maskChar) != KAZE_toupper(c))
        return false;
                      } else {
      if ((maskChar) != (c))
        return false;
                      }
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask,  maskPos + 1, name, namePos + 1);
  }
}

static boolean EnhancedMaskTest_OrEmpty_AndNotEmpty_c(const char *mask, int maskPos, const char *name, int namePos)
{
  char maskChar;
  char c;
  //int maskLen = KAZE_strlen(mask) - maskPos;
  //int nameLen = KAZE_strlenLF(name) - namePos;
  // Above 2 lines are modified with GLOBAL variables for speed as follows:
  int maskLen = maskGLOBALlen - maskPos; // for speed up ((12415s - 755s)/755)*100% = 1544%
  int nameLen = nameGLOBALlenc - namePos;
  if (maskLen == 0)
    if (nameLen == 0)
      return true;
    else
      return false;
  maskChar = mask[maskPos];
  if (maskChar == '@') // or empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask, maskPos + 1, name, namePos)) // KAZE: THIS LINE DECIDES whether 'or empty' or 'and not empty'
      return true;                                                //       uncommented is 'or empty'
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '#') // and not empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask,  maskPos + 1, name, namePos + 1);
  }
//       wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty},
  else if(maskChar == '|') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
```

```
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '~') // and not empty AND NOT ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask,  maskPos + 1, name, namePos + 1);
  }
//       wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
  else if(maskChar == '^') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '$') // and not empty AND ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask,  maskPos + 1, name, namePos + 1);
  }


  else if(maskChar == '`') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask,  maskPos, name, namePos + 1);
  }
  else if(maskChar == '.') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask, maskPos + 1, name, namePos))
      return true;
```

```
    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask,  maskPos, name, namePos + 1);
  }


  else if(maskChar == '*')
  {
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask, maskPos + 1, name, namePos))
      return true;
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask, maskPos, name, namePos + 1);
  }
  else
  {
    c = name[namePos];
    //if (maskChar != c)
                      if (CaseSensitiveWildcardMatching_flag == 0) {
      if (KAZE_toupper(maskChar) != KAZE_toupper(c))
        return false;
                      } else {
      if ((maskChar) != (c))
        return false;
                      }
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask,  maskPos + 1, name, namePos + 1);
  }
}

static boolean EnhancedMaskTest_OrEmpty_AndNotEmpty_d(const char *mask, int maskPos, const char *name, int namePos)
{
  char maskChar;
  char c;
  //int maskLen = KAZE_strlen(mask) - maskPos;
  //int nameLen = KAZE_strlenLF(name) - namePos;
  // Above 2 lines are modified with GLOBAL variables for speed as follows:
  int maskLen = maskGLOBALlen - maskPos; // for speed up ((12415s - 755s)/755)*100% = 1544%
  int nameLen = nameGLOBALlend - namePos;
  if (maskLen == 0)
    if (nameLen == 0)
      return true;
    else
      return false;
  maskChar = mask[maskPos];
  if (maskChar == '@') // or empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask, maskPos + 1, name, namePos)) // KAZE: THIS LINE DECIDES whether 'or empty' or 'and not empty'
      return true;                                                  //       uncommented is 'or empty'
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '#') // and not empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (nameLen == 0)
      return false;
```

```
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty},
  else if(maskChar == '|') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '~') // and not empty AND NOT ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
  else if(maskChar == '^') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '$') // and not empty AND ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask,  maskPos + 1, name, namePos + 1);
  }


  else if(maskChar == '`') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask, maskPos + 1, name, namePos))
      return true;
```

```
    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask,  maskPos, name, namePos + 1);
  }
  else if(maskChar == '.') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask,  maskPos, name, namePos + 1);
  }


  else if(maskChar == '*')
  {
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask, maskPos + 1, name, namePos))
      return true;
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask, maskPos, name, namePos + 1);
  }
  else
  {
    c = name[namePos];
    //if (maskChar != c)
                      if (CaseSensitiveWildcardMatching_flag == 0) {
      if (KAZE_toupper(maskChar) != KAZE_toupper(c))
        return false;
                      } else {
      if ((maskChar) != (c))
        return false;
                      }
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask,  maskPos + 1, name, namePos + 1);
  }
}

static boolean EnhancedMaskTest_OrEmpty_AndNotEmpty_e(const char *mask, int maskPos, const char *name, int namePos)
{
  char maskChar;
  char c;
  //int maskLen = KAZE_strlen(mask) - maskPos;
  //int nameLen = KAZE_strlenLF(name) - namePos;
  // Above 2 lines are modified with GLOBAL variables for speed as follows:
  int maskLen = maskGLOBALlen - maskPos; // for speed up ((12415s - 755s)/755)*100% = 1544%
  int nameLen = nameGLOBALlene - namePos;
  if (maskLen == 0)
    if (nameLen == 0)
      return true;
    else
      return false;
  maskChar = mask[maskPos];
  if (maskChar == '@') // or empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask, maskPos + 1, name, namePos))
      return true;
    */
```

```
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask, maskPos + 1, name, namePos)) // KAZE: THIS LINE DECIDES whether 'or empty' or 'and not empty'
      return true;                                                    //       uncommented is 'or empty'
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '#') // and not empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty},
  else if(maskChar == '|') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '~') // and not empty AND NOT ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
  else if(maskChar == '^') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '$') // and not empty AND ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
```

```
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask,  maskPos + 1, name, namePos + 1);
  }


  else if(maskChar == '`') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask,  maskPos, name, namePos + 1);
  }
  else if(maskChar == '.') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask,  maskPos, name, namePos + 1);
  }


  else if(maskChar == '*')
  {
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask, maskPos + 1, name, namePos))
      return true;
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask, maskPos, name, namePos + 1);
  }
  else
  {
    c = name[namePos];
    //if (maskChar != c)
                      if (CaseSensitiveWildcardMatching_flag == 0) {
      if (KAZE_toupper(maskChar) != KAZE_toupper(c))
        return false;
                      } else {
      if ((maskChar) != (c))
        return false;
                      }
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask,  maskPos + 1, name, namePos + 1);
  }
}

static boolean EnhancedMaskTest_OrEmpty_AndNotEmpty_f(const char *mask, int maskPos, const char *name, int namePos)
{
  char maskChar;
  char c;
  //int maskLen = KAZE_strlen(mask) - maskPos;
  //int nameLen = KAZE_strlenLF(name) - namePos;
  // Above 2 lines are modified with GLOBAL variables for speed as follows:
```

```
  int maskLen = maskGLOBALlen - maskPos; // for speed up ((12415s - 755s)/755)*100% = 1544%
  int nameLen = nameGLOBALlenf - namePos;
  if (maskLen == 0)
    if (nameLen == 0)
      return true;
    else
      return false;
  maskChar = mask[maskPos];
  if (maskChar == '@') // or empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask, maskPos + 1, name, namePos)) // KAZE: THIS LINE DECIDES whether 'or empty' or 'and not empty'
      return true;                                                //       uncommented is 'or empty'
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '#') // and not empty
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask, maskPos + 1, name, namePos))
      return true;
    */
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty},
  else if(maskChar == '|') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '~') // and not empty AND NOT ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask,  maskPos + 1, name, namePos + 1);
  }
//      wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
  else if(maskChar == '^') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
```

```
      return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask,  maskPos + 1, name, namePos + 1);
  }
  else if(maskChar == '$') // and not empty AND ALPHA
  {
    /*
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask, maskPos + 1, name, namePos))
      return true;
    */

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask,  maskPos + 1, name, namePos + 1);
  }


  else if(maskChar == '`') // or empty AND NOT ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) >= 'A') && (KAZE_toupper(c) <= 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask,  maskPos, name, namePos + 1);
  }
  else if(maskChar == '.') // or empty AND ALPHA
  {

    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask, maskPos + 1, name, namePos))
      return true;

    c = name[namePos];
      if ( (KAZE_toupper(c) < 'A') || (KAZE_toupper(c) > 'Z') ) // Stupidly slow: make it faster ...
        return false;

    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask,  maskPos, name, namePos + 1);
  }


  else if(maskChar == '*')
  {
    if (EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask, maskPos + 1, name, namePos))
      return true;
    if (nameLen == 0)
      return false;
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask, maskPos, name, namePos + 1);
  }
  else
  {
    c = name[namePos];
    //if (maskChar != c)
                      if (CaseSensitiveWildcardMatching_flag == 0) {
      if (KAZE_toupper(maskChar) != KAZE_toupper(c))
        return false;
                      } else {
```

```
      if ((maskChar) != (c))
        return false;
                        }
    return EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask,  maskPos + 1, name, namePos + 1);
  }
}


/*
boolean CompareWildCardWithName1(const char *mask, const char *name)
{
  boolean Txpbool;
  //VIVA_IgorPavlov_invocations_global_counter++;

  // The two lines below are nasty, Very slow:
  //maskGLOBALlen = KAZE_strlen(mask);
  //nameGLOBALlen1 = KAZE_strlen(name);

  Txpbool = EnhancedMaskTest_OrEmpty_AndNotEmpty_1(mask, 0, name, 0);
  //if (Txpbool) WildGLOBALhits++;
  return Txpbool;
}
boolean CompareWildCardWithName2(const char *mask, const char *name)
{
  boolean Txpbool;
  //VIVA_IgorPavlov_invocations_global_counter++;

  // The two lines below are nasty, Very slow:
  //maskGLOBALlen = KAZE_strlen(mask);
  //nameGLOBALlen2 = KAZE_strlen(name);

  Txpbool = EnhancedMaskTest_OrEmpty_AndNotEmpty_2(mask, 0, name, 0);
  //if (Txpbool) WildGLOBALhits++;
  return Txpbool;
}
boolean CompareWildCardWithName3(const char *mask, const char *name)
{
  boolean Txpbool;
  //VIVA_IgorPavlov_invocations_global_counter++;

  // The two lines below are nasty, Very slow:
  //maskGLOBALlen = KAZE_strlen(mask);
  //nameGLOBALlen3 = KAZE_strlen(name);

  Txpbool = EnhancedMaskTest_OrEmpty_AndNotEmpty_3(mask, 0, name, 0);
  //if (Txpbool) WildGLOBALhits++;
  return Txpbool;
}
boolean CompareWildCardWithName4(const char *mask, const char *name)
{
  boolean Txpbool;
  //VIVA_IgorPavlov_invocations_global_counter++;

  // The two lines below are nasty, Very slow:
  //maskGLOBALlen = KAZE_strlen(mask);
  //nameGLOBALlen4 = KAZE_strlen(name);

  Txpbool = EnhancedMaskTest_OrEmpty_AndNotEmpty_4(mask, 0, name, 0);
  //if (Txpbool) WildGLOBALhits++;
  return Txpbool;
}
boolean CompareWildCardWithName5(const char *mask, const char *name)
{
  boolean Txpbool;
  //VIVA_IgorPavlov_invocations_global_counter++;

  // The two lines below are nasty, Very slow:
  //maskGLOBALlen = KAZE_strlen(mask);
  //nameGLOBALlen5 = KAZE_strlen(name);
```

```
  Txpbool = EnhancedMaskTest_OrEmpty_AndNotEmpty_5(mask, 0, name, 0);
  //if (Txpbool) WildGLOBALhits++;
  return Txpbool;
}
boolean CompareWildCardWithName6(const char *mask, const char *name)
{
  boolean Txpbool;
  //VIVA_IgorPavlov_invocations_global_counter++;

  // The two lines below are nasty, Very slow:
  //maskGLOBALlen = KAZE_strlen(mask);
  //nameGLOBALlen6 = KAZE_strlen(name);

  Txpbool = EnhancedMaskTest_OrEmpty_AndNotEmpty_6(mask, 0, name, 0);
  //if (Txpbool) WildGLOBALhits++;
  return Txpbool;
}
boolean CompareWildCardWithName7(const char *mask, const char *name)
{
  boolean Txpbool;
  //VIVA_IgorPavlov_invocations_global_counter++;

  // The two lines below are nasty, Very slow:
  //maskGLOBALlen = KAZE_strlen(mask);
  //nameGLOBALlen7 = KAZE_strlen(name);

  Txpbool = EnhancedMaskTest_OrEmpty_AndNotEmpty_7(mask, 0, name, 0);
  //if (Txpbool) WildGLOBALhits++;
  return Txpbool;
}
boolean CompareWildCardWithName8(const char *mask, const char *name)
{
  boolean Txpbool;
  //VIVA_IgorPavlov_invocations_global_counter++;

  // The two lines below are nasty, Very slow:
  //maskGLOBALlen = KAZE_strlen(mask);
  //nameGLOBALlen8 = KAZE_strlen(name);

  Txpbool = EnhancedMaskTest_OrEmpty_AndNotEmpty_8(mask, 0, name, 0);
  //if (Txpbool) WildGLOBALhits++;
  return Txpbool;
}
boolean CompareWildCardWithName9(const char *mask, const char *name)
{
  boolean Txpbool;
  //VIVA_IgorPavlov_invocations_global_counter++;

  // The two lines below are nasty, Very slow:
  //maskGLOBALlen = KAZE_strlen(mask);
  //nameGLOBALlen9 = KAZE_strlen(name);

  Txpbool = EnhancedMaskTest_OrEmpty_AndNotEmpty_9(mask, 0, name, 0);
  //if (Txpbool) WildGLOBALhits++;
  return Txpbool;
}
boolean CompareWildCardWithName0(const char *mask, const char *name)
{
  boolean Txpbool;
  //VIVA_IgorPavlov_invocations_global_counter++;

  // The two lines below are nasty, Very slow:
  //maskGLOBALlen = KAZE_strlen(mask);
  //nameGLOBALlen0 = KAZE_strlen(name);

  Txpbool = EnhancedMaskTest_OrEmpty_AndNotEmpty_0(mask, 0, name, 0);
  //if (Txpbool) WildGLOBALhits++;
  return Txpbool;
```

```
}
boolean CompareWildCardWithNamea(const char *mask, const char *name)
{
  boolean Txpbool;
  //VIVA_IgorPavlov_invocations_global_counter++;

  // The two lines below are nasty, Very slow:
  //maskGLOBALlen = KAZE_strlen(mask);
  //nameGLOBALlena = KAZE_strlen(name);

  Txpbool = EnhancedMaskTest_OrEmpty_AndNotEmpty_a(mask, 0, name, 0);
  //if (Txpbool) WildGLOBALhits++;
  return Txpbool;
}
boolean CompareWildCardWithNameb(const char *mask, const char *name)
{
  boolean Txpbool;
  //VIVA_IgorPavlov_invocations_global_counter++;

  // The two lines below are nasty, Very slow:
  //maskGLOBALlen = KAZE_strlen(mask);
  //nameGLOBALlenb = KAZE_strlen(name);

  Txpbool = EnhancedMaskTest_OrEmpty_AndNotEmpty_b(mask, 0, name, 0);
  //if (Txpbool) WildGLOBALhits++;
  return Txpbool;
}
boolean CompareWildCardWithNamec(const char *mask, const char *name)
{
  boolean Txpbool;
  //VIVA_IgorPavlov_invocations_global_counter++;

  // The two lines below are nasty, Very slow:
  //maskGLOBALlen = KAZE_strlen(mask);
  //nameGLOBALlenc = KAZE_strlen(name);

  Txpbool = EnhancedMaskTest_OrEmpty_AndNotEmpty_c(mask, 0, name, 0);
  //if (Txpbool) WildGLOBALhits++;
  return Txpbool;
}
boolean CompareWildCardWithNamed(const char *mask, const char *name)
{
  boolean Txpbool;
  //VIVA_IgorPavlov_invocations_global_counter++;

  // The two lines below are nasty, Very slow:
  //maskGLOBALlen = KAZE_strlen(mask);
  //nameGLOBALlend = KAZE_strlen(name);

  Txpbool = EnhancedMaskTest_OrEmpty_AndNotEmpty_d(mask, 0, name, 0);
  //if (Txpbool) WildGLOBALhits++;
  return Txpbool;
}
boolean CompareWildCardWithNamee(const char *mask, const char *name)
{
  boolean Txpbool;
  //VIVA_IgorPavlov_invocations_global_counter++;

  // The two lines below are nasty, Very slow:
  //maskGLOBALlen = KAZE_strlen(mask);
  //nameGLOBALlene = KAZE_strlen(name);

  Txpbool = EnhancedMaskTest_OrEmpty_AndNotEmpty_e(mask, 0, name, 0);
  //if (Txpbool) WildGLOBALhits++;
  return Txpbool;
}
boolean CompareWildCardWithNamef(const char *mask, const char *name)
{
  boolean Txpbool;
```

```
  //VIVA_IgorPavlov_invocations_global_counter++;

  // The two lines below are nasty, Very slow:
  //maskGLOBALlen = KAZE_strlen(mask);
  //nameGLOBALlenf = KAZE_strlen(name);

  Txpbool = EnhancedMaskTest_OrEmpty_AndNotEmpty_f(mask, 0, name, 0);
  //if (Txpbool) WildGLOBALhits++;
  return Txpbool;
}
*/

// ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]
// Modified Igor Pavlov's 7zip wildcard code ]

/*
                                  ...

                        if( wrdlen < MaxLineLength +1+1)
                        { wrd[ wrdlen ] = KAZE_tolower( workbyte ); }

                        if (workbyte == 10) {NumberOfLines++;
                            if (wrdlen > LongestLine) LongestLine=wrdlen;
                            if (wrdlen < ShortestLine) ShortestLine=wrdlen;
// Wildcard search [
                        if ( 0 < wrdlen && wrdlen < MaxLineLength +1+1)
                                  {
                            wrd[ wrdlen ] = 0;
                                             if ( CompareWildCardWithName ( LINE10_2, wrd ) ) {
                                                        //printf("%s\n", wrd);
                                                        fprintf( fp_outLOG, "%s\n", wrd); DumpedLines++;
                                             if ((DumpedLines & 0xff) == 0xff)
                                                        //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                        fflush(fp_outLOG); // Not sure: CTRL+C doesn't flush?!
                                             }

                                  }
// Wildcard search ]
                            wrdlen = 0;
                        }
                        else wrdlen++;

                                  ...
*/

// SLOW, recursive:
//       wildcard '*' any character(s) or empty,
//       wildcard '.' any ALPHA character(s) or empty,
//       wildcard '`' any NON-ALPHA character(s) or empty,
//       wildcard '@'/'#' any character {or empty}/{and not empty},
//       wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
//       wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty},
//
// SLOW mode is clearly written, however recursive calls are very "overheadY" also it has scary branching:
/*
  if (maskChar == '@') // or empty
  {
  }
  else if(maskChar == '#') // and not empty
  {
  }
  else if(maskChar == '|') // or empty AND NOT ALPHA
  {
  }
  else if(maskChar == '~') // and not empty AND NOT ALPHA
  {
  }
  else if(maskChar == '^') // or empty AND ALPHA
  {
```

```
  }
  else if(maskChar == '$') // and not empty AND ALPHA
  {
  }
  else if(maskChar == '`') // or empty AND NOT ALPHA
  {
  }
  else if(maskChar == '.') // or empty AND ALPHA
  {
  }
  else if(maskChar == '*')
  {
  }
  else
  {
  }
*/
// The way it is done (in Kazahana) now the SLOW overrides the FAST, meaning that presence of at least one of the 9 wildcards trumps the FAST mode.
// FAST, iterative:
//       wildcard '&' any character(s) or empty,
//       wildcard '+' any character and not empty.
// CAUTION: For speed case-insensitivity is achieved outwith the function i.e. by default it is case-sensitive unless you did take care of the opposite.
static boolean IterativeWildcards1(const char *mask, const char *name)
{
const char *maskSTACK;
const char *nameSTACK;

while ( (*name) && (*mask != '&') ) {
           if ( (*mask != *name) && (*mask != '+') )
                      return false;
           mask++;
           name++;
}
while ( *name ) {
           if ( *mask == '&' ) {
                      if (!*++mask)
                                 return true;
                      maskSTACK = mask;
                      nameSTACK = name + 1;
           } else if ( (*mask == *name) || (*mask == '+') ) {
                      mask++;
                      name++;
           } else {
                      mask = maskSTACK;
                      name = nameSTACK++;
           }
}
while ( *mask == '&' )
           mask++;
return !*mask;
}
static boolean IterativeWildcards2(const char *mask, const char *name)
{
const char *maskSTACK;
const char *nameSTACK;

while ( (*name) && (*mask != '&') ) {
           if ( (*mask != *name) && (*mask != '+') )
                      return false;
           mask++;
           name++;
}
while ( *name ) {
           if ( *mask == '&' ) {
                      if (!*++mask)
                                 return true;
                      maskSTACK = mask;
                      nameSTACK = name + 1;
           } else if ( (*mask == *name) || (*mask == '+') ) {
```

```
                        mask++;
                        name++;
            } else {
                        mask = maskSTACK;
                        name = nameSTACK++;
            }
}
while ( *mask == '&' )
            mask++;
return !*mask;
}
static boolean IterativeWildcards3(const char *mask, const char *name)
{
const char *maskSTACK;
const char *nameSTACK;

while ( (*name) && (*mask != '&') ) {
            if ( (*mask != *name) && (*mask != '+') )
                        return false;
            mask++;
            name++;
}
while ( *name ) {
            if ( *mask == '&' ) {
                        if (!*++mask)
                                    return true;
                        maskSTACK = mask;
                        nameSTACK = name + 1;
            } else if ( (*mask == *name) || (*mask == '+') ) {
                        mask++;
                        name++;
            } else {
                        mask = maskSTACK;
                        name = nameSTACK++;
            }
}
while ( *mask == '&' )
            mask++;
return !*mask;
}
static boolean IterativeWildcards4(const char *mask, const char *name)
{
const char *maskSTACK;
const char *nameSTACK;

while ( (*name) && (*mask != '&') ) {
            if ( (*mask != *name) && (*mask != '+') )
                        return false;
            mask++;
            name++;
}
while ( *name ) {
            if ( *mask == '&' ) {
                        if (!*++mask)
                                    return true;
                        maskSTACK = mask;
                        nameSTACK = name + 1;
            } else if ( (*mask == *name) || (*mask == '+') ) {
                        mask++;
                        name++;
            } else {
                        mask = maskSTACK;
                        name = nameSTACK++;
            }
}
while ( *mask == '&' )
            mask++;
return !*mask;
}
```

```
static boolean IterativeWildcards5(const char *mask, const char *name)
{
const char *maskSTACK;
const char *nameSTACK;

while ( (*name) && (*mask != '&') ) {
           if ( (*mask != *name) && (*mask != '+') )
                      return false;
           mask++;
           name++;
}
while ( *name ) {
           if ( *mask == '&' ) {
                      if (!*++mask)
                                 return true;
                      maskSTACK = mask;
                      nameSTACK = name + 1;
           } else if ( (*mask == *name) || (*mask == '+') ) {
                      mask++;
                      name++;
           } else {
                      mask = maskSTACK;
                      name = nameSTACK++;
           }
}
while ( *mask == '&' )
           mask++;
return !*mask;
}
static boolean IterativeWildcards6(const char *mask, const char *name)
{
const char *maskSTACK;
const char *nameSTACK;

while ( (*name) && (*mask != '&') ) {
           if ( (*mask != *name) && (*mask != '+') )
                      return false;
           mask++;
           name++;
}
while ( *name ) {
           if ( *mask == '&' ) {
                      if (!*++mask)
                                 return true;
                      maskSTACK = mask;
                      nameSTACK = name + 1;
           } else if ( (*mask == *name) || (*mask == '+') ) {
                      mask++;
                      name++;
           } else {
                      mask = maskSTACK;
                      name = nameSTACK++;
           }
}
while ( *mask == '&' )
           mask++;
return !*mask;
}
static boolean IterativeWildcards7(const char *mask, const char *name)
{
const char *maskSTACK;
const char *nameSTACK;

while ( (*name) && (*mask != '&') ) {
           if ( (*mask != *name) && (*mask != '+') )
                      return false;
           mask++;
           name++;
}
```

```
while ( *name ) {
		if ( *mask == '&' ) {
			if (!*++mask)
				return true;
			maskSTACK = mask;
			nameSTACK = name + 1;
		} else if ( (*mask == *name) || (*mask == '+') ) {
			mask++;
			name++;
		} else {
			mask = maskSTACK;
			name = nameSTACK++;
		}
}
while ( *mask == '&' )
		mask++;
return !*mask;
}
static boolean IterativeWildcards8(const char *mask, const char *name)
{
const char *maskSTACK;
const char *nameSTACK;

while ( (*name) && (*mask != '&') ) {
		if ( (*mask != *name) && (*mask != '+') )
			return false;
		mask++;
		name++;
}
while ( *name ) {
		if ( *mask == '&' ) {
			if (!*++mask)
				return true;
			maskSTACK = mask;
			nameSTACK = name + 1;
		} else if ( (*mask == *name) || (*mask == '+') ) {
			mask++;
			name++;
		} else {
			mask = maskSTACK;
			name = nameSTACK++;
		}
}
while ( *mask == '&' )
		mask++;
return !*mask;
}
static boolean IterativeWildcards9(const char *mask, const char *name)
{
const char *maskSTACK;
const char *nameSTACK;

while ( (*name) && (*mask != '&') ) {
		if ( (*mask != *name) && (*mask != '+') )
			return false;
		mask++;
		name++;
}
while ( *name ) {
		if ( *mask == '&' ) {
			if (!*++mask)
				return true;
			maskSTACK = mask;
			nameSTACK = name + 1;
		} else if ( (*mask == *name) || (*mask == '+') ) {
			mask++;
			name++;
		} else {
			mask = maskSTACK;
```

```
                        name = nameSTACK++;
            }
}
while ( *mask == '&' )
            mask++;
return !*mask;
}
static boolean IterativeWildcards0(const char *mask, const char *name)
{
const char *maskSTACK;
const char *nameSTACK;

while ( (*name) && (*mask != '&') ) {
            if ( (*mask != *name) && (*mask != '+') )
                        return false;
            mask++;
            name++;
}
while ( *name ) {
            if ( *mask == '&' ) {
                        if (!*++mask)
                                    return true;
                        maskSTACK = mask;
                        nameSTACK = name + 1;
            } else if ( (*mask == *name) || (*mask == '+') ) {
                        mask++;
                        name++;
            } else {
                        mask = maskSTACK;
                        name = nameSTACK++;
            }
}
while ( *mask == '&' )
            mask++;
return !*mask;
}
static boolean IterativeWildcardsa(const char *mask, const char *name)
{
const char *maskSTACK;
const char *nameSTACK;

while ( (*name) && (*mask != '&') ) {
            if ( (*mask != *name) && (*mask != '+') )
                        return false;
            mask++;
            name++;
}
while ( *name ) {
            if ( *mask == '&' ) {
                        if (!*++mask)
                                    return true;
                        maskSTACK = mask;
                        nameSTACK = name + 1;
            } else if ( (*mask == *name) || (*mask == '+') ) {
                        mask++;
                        name++;
            } else {
                        mask = maskSTACK;
                        name = nameSTACK++;
            }
}
while ( *mask == '&' )
            mask++;
return !*mask;
}
static boolean IterativeWildcardsb(const char *mask, const char *name)
{
const char *maskSTACK;
const char *nameSTACK;
```

```
while ( (*name) && (*mask != '&') ) {
           if ( (*mask != *name) && (*mask != '+') )
                      return false;
           mask++;
           name++;
}
while ( *name ) {
           if ( *mask == '&' ) {
                      if (!*++mask)
                                 return true;
                      maskSTACK = mask;
                      nameSTACK = name + 1;
           } else if ( (*mask == *name) || (*mask == '+') ) {
                      mask++;
                      name++;
           } else {
                      mask = maskSTACK;
                      name = nameSTACK++;
           }
}
while ( *mask == '&' )
           mask++;
return !*mask;
}
static boolean IterativeWildcardsc(const char *mask, const char *name)
{
const char *maskSTACK;
const char *nameSTACK;

while ( (*name) && (*mask != '&') ) {
           if ( (*mask != *name) && (*mask != '+') )
                      return false;
           mask++;
           name++;
}
while ( *name ) {
           if ( *mask == '&' ) {
                      if (!*++mask)
                                 return true;
                      maskSTACK = mask;
                      nameSTACK = name + 1;
           } else if ( (*mask == *name) || (*mask == '+') ) {
                      mask++;
                      name++;
           } else {
                      mask = maskSTACK;
                      name = nameSTACK++;
           }
}
while ( *mask == '&' )
           mask++;
return !*mask;
}
static boolean IterativeWildcardsd(const char *mask, const char *name)
{
const char *maskSTACK;
const char *nameSTACK;

while ( (*name) && (*mask != '&') ) {
           if ( (*mask != *name) && (*mask != '+') )
                      return false;
           mask++;
           name++;
}
while ( *name ) {
           if ( *mask == '&' ) {
                      if (!*++mask)
                                 return true;
```

```
                        maskSTACK = mask;
                        nameSTACK = name + 1;
            } else if ( (*mask == *name) || (*mask == '+') ) {
                        mask++;
                        name++;
            } else {
                        mask = maskSTACK;
                        name = nameSTACK++;
            }
}
while ( *mask == '&' )
            mask++;
return !*mask;
}
static boolean IterativeWildcardse(const char *mask, const char *name)
{
const char *maskSTACK;
const char *nameSTACK;

while ( (*name) && (*mask != '&') ) {
            if ( (*mask != *name) && (*mask != '+') )
                        return false;
            mask++;
            name++;
}
while ( *name ) {
            if ( *mask == '&' ) {
                        if (!*++mask)
                                    return true;
                        maskSTACK = mask;
                        nameSTACK = name + 1;
            } else if ( (*mask == *name) || (*mask == '+') ) {
                        mask++;
                        name++;
            } else {
                        mask = maskSTACK;
                        name = nameSTACK++;
            }
}
while ( *mask == '&' )
            mask++;
return !*mask;
}
static boolean IterativeWildcardsf(const char *mask, const char *name)
{
const char *maskSTACK;
const char *nameSTACK;

while ( (*name) && (*mask != '&') ) {
            if ( (*mask != *name) && (*mask != '+') )
                        return false;
            mask++;
            name++;
}
while ( *name ) {
            if ( *mask == '&' ) {
                        if (!*++mask)
                                    return true;
                        maskSTACK = mask;
                        nameSTACK = name + 1;
            } else if ( (*mask == *name) || (*mask == '+') ) {
                        mask++;
                        name++;
            } else {
                        mask = maskSTACK;
                        name = nameSTACK++;
            }
}
while ( *mask == '&' )
```

```
            mask++;
return !*mask;
}


//#define _WIN32ASM_

#if defined(_WIN32ASM_)
// Optimizing subroutines in assembly language
// An optimization guide for x86 platforms
// By Agner Fog. Copenhagen University College of Engineering.
/*
It is possible to calculate the absolute value of a signed integer without branching:
; Example 9.15, Calculate absolute value of eax
            cdq ; Copy sign bit of eax to all bits of edx
            xor eax, edx ; Invert all bits if negative
            sub eax, edx ; Add 1 if negative
The following example finds the minimum of two unsigned numbers: if (b > a) b = a;
; Example 9.16a, Find minimum of eax and ebx (unsigned):
            sub eax, ebx ; = a-b
            sbb edx, edx ; = (b > a) ? 0xFFFFFFFF : 0
            and edx, eax ; = (b > a) ? a-b : 0
            add ebx, edx ; Result is in ebx
*/
// Sadly MS 64bit compiler accepts not the inline ASM: error C4235: nonstandard extension used : '__asm' keyword not supported on this architecture
// MASM style inline assembly, 32 bit mode
unsigned int abs_AF (int n) {
__asm {
            mov eax, n // Move n to eax
                                // abs(n) is calculated by inverting all bits and adding 1 if n < 0:
            cdq                 // Get sign bit into all bits of edx
            xor eax, edx        // Invert bits if negative
            sub eax, edx        // Add 1 if negative. Now eax = abs(n)
}                               // Return value is in eax
}
unsigned int min_AF (int a, int b, int c) {
__asm {
            mov eax, a // Move a to eax
            mov ebx, b // Move b to ebx

            sub eax, ebx ; = a-b
            sbb edx, edx ; = (b > a) ? 0xFFFFFFFF : 0
            and edx, eax ; = (b > a) ? a-b : 0
            add ebx, edx ; Result is in ebx

            mov eax, c // Move c to eax

            sub eax, ebx ; = a-b
            sbb edx, edx ; = (b > a) ? 0xFFFFFFFF : 0
            and edx, eax ; = (b > a) ? a-b : 0
            add ebx, edx ; Result is in ebx

            mov eax, ebx ; Return value is in eax
}
}
#endif

void x64toaKAZE (      /* stdcall is faster and smaller... Might as well use it for the helper. */
        unsigned long long val,
        char *buf,
        unsigned radix,
        int is_neg
        )
{
        char *p;                /* pointer to traverse string */
        char *firstdig;         /* pointer to first digit */
        char temp;              /* temp char */
        unsigned digval;        /* value of digit */
```

```
        p = buf;

        if ( is_neg )
        {
            *p++ = '-';         /* negative, so output '-' and negate */
            val = (unsigned long long)(-(long long)val);
        }

        firstdig = p;           /* save pointer to first digit */

        do {
            digval = (unsigned) (val % radix);
            val /= radix;       /* get next digit */

            /* convert to ascii and store */
            if (digval > 9)
                *p++ = (char) (digval - 10 + 'a');  /* a letter */
            else
                *p++ = (char) (digval + '0');       /* a digit */
        } while (val > 0);

        /* We now have the digit of the number in the buffer, but in reverse
           order.  Thus we reverse them now. */

        *p-- = '\0';            /* terminate string; p points to last digit */

        do {
            temp = *p;
            *p = *firstdig;
            *firstdig = temp;   /* swap *p and *firstdig */
            --p;
            ++firstdig;         /* advance to next two digits */
        } while (firstdig < p); /* repeat until halfway */
}

/* Actual functions just call conversion helper with neg flag set correctly,
   and return pointer to buffer. */

char * _i64toaKAZE (
        long long val,
        char *buf,
        int radix
        )
{
        x64toaKAZE((unsigned long long)val, buf, radix, (radix == 10 && val < 0));
        return buf;
}

char * _ui64toaKAZE (
        unsigned long long val,
        char *buf,
        int radix
        )
{
        x64toaKAZE(val, buf, radix, 0);
        return buf;
}

char * _ui64toaKAZEzerocomma (
        unsigned long long val,
        char *buf,
        int radix
        )
{
                        char *p;
                        char temp;
                        int txpman;
                        int pxnman;
        x64toaKAZE(val, buf, radix, 0);
```

```
                        p = buf;
                        do {
                        } while (*++p != '\0');
                        p--; // p points to last digit
                             // buf points to first digit
                        buf[26] = 0;
                        txpman = 1;
                        pxnman = 0;
                        do
                        { if (buf <= p)
                          { temp = *p;
                            buf[26-txpman] = temp; pxnman++;
                            p--;
                            if (pxnman % 3 == 0)
                            { txpman++;
                              buf[26-txpman] = (char) (',');
                            }
                          }
                          else
                          { buf[26-txpman] = (char) ('0'); pxnman++;
                            if (pxnman % 3 == 0)
                            { txpman++;
                              buf[26-txpman] = (char) (',');
                            }
                          }
                          txpman++;
                        } while (txpman <= 26);
        return buf;
}

char * _ui64toaKAZEcomma (
        unsigned long long val,
        char *buf,
        int radix
        )
{
                        char *p;
                        char temp;
                        int txpman;
                        int pxnman;
        x64toaKAZE(val, buf, radix, 0);
                        p = buf;
                        do {
                        } while (*++p != '\0');
                        p--; // p points to last digit
                             // buf points to first digit
                        buf[26] = 0;
                        txpman = 1;
                        pxnman = 0;
                        while (buf <= p)
                        { temp = *p;
                          buf[26-txpman] = temp; pxnman++;
                          p--;
                          if (pxnman % 3 == 0 && buf <= p)
                          { txpman++;
                            buf[26-txpman] = (char) (',');
                          }
                          txpman++;
                        }
        return buf+26-(txpman-1);
}

// Wagner-Fischer algorithm
// From Wikipedia, the free encyclopedia
/*
 int LevenshteinDistance(char s[1..m], char t[1..n])
 {
   // for all i and j, d[i,j] will hold the Levenshtein distance between
   // the first i characters of s and the first j characters of t;
```

```
   // note that d has (m+1)x(n+1) values
   declare int d[0..m, 0..n]

   for i from 0 to m
     d[i, 0] := i // the distance of any first string to an empty second string
   for j from 0 to n
     d[0, j] := j // the distance of any second string to an empty first string

   for j from 1 to n
   {
     for i from 1 to m
     {
       if s[i] = t[j] then
         d[i, j] := d[i-1, j-1]       // no operation required
       else
         d[i, j] := minimum
                    (
                      d[i-1, j] + 1,  // a deletion
                      d[i, j-1] + 1,  // an insertion
                      d[i-1, j-1] + 1 // a substitution
                    )
     }
   }

   return d[m,n]
 }
*/

#if defined(_WIN32_ENVIRONMENT_)
#include <io.h> // needed for Windows' 'lseeki64' and 'telli64'
#else
#endif /* defined(_WIN32_ENVIRONMENT_)  */

#ifdef Commence_OpenMP
#include <omp.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

            #define THREADSnumber 16
            //#define CACHEsize MasterBuffer*(1<<20) // /7*4/3
            int CACHEsize;
            #define MAXxgamsInCACHE 16*4096 // 16 threads * 4096 xgrams * 128 bytes = 8,388,608
            //#define MaxLineLength 126+100
// !!! How unexpected !!!
// I assumed the above line means 226 always yet the compilers gave:
// MaxLineLength = 226
// (167*MaxLineLength) = 21142
// MaxLineLength +(167*MaxLineLength) = 21368
// So never again such sums, they mean (167*MaxLineLength) = (167*126+100) instead of (167*(126+100))
            #define MaxLineLength 156

            // Stupid bug fixed, 'unsigned int' was an error during pasting code from Galadriel:
            int LevenshteinT1[MaxLineLength+1][MaxLineLength+1]; // declare int d[0..m, 0..n]
            int LevenshteinT2[MaxLineLength+1][MaxLineLength+1]; // declare int d[0..m, 0..n]
            int LevenshteinT3[MaxLineLength+1][MaxLineLength+1]; // declare int d[0..m, 0..n]
            int LevenshteinT4[MaxLineLength+1][MaxLineLength+1]; // declare int d[0..m, 0..n]
            int LevenshteinT5[MaxLineLength+1][MaxLineLength+1]; // declare int d[0..m, 0..n]
            int LevenshteinT6[MaxLineLength+1][MaxLineLength+1]; // declare int d[0..m, 0..n]
            int LevenshteinT7[MaxLineLength+1][MaxLineLength+1]; // declare int d[0..m, 0..n]
            int LevenshteinT8[MaxLineLength+1][MaxLineLength+1]; // declare int d[0..m, 0..n]
            int LevenshteinT9[MaxLineLength+1][MaxLineLength+1]; // declare int d[0..m, 0..n]
            int LevenshteinT0[MaxLineLength+1][MaxLineLength+1]; // declare int d[0..m, 0..n]
            int LevenshteinTa[MaxLineLength+1][MaxLineLength+1]; // declare int d[0..m, 0..n]
            int LevenshteinTb[MaxLineLength+1][MaxLineLength+1]; // declare int d[0..m, 0..n]
            int LevenshteinTc[MaxLineLength+1][MaxLineLength+1]; // declare int d[0..m, 0..n]
```

```
            int LevenshteinTd[MaxLineLength+1][MaxLineLength+1]; // declare int d[0..m, 0..n]
            int LevenshteinTe[MaxLineLength+1][MaxLineLength+1]; // declare int d[0..m, 0..n]
            int LevenshteinTf[MaxLineLength+1][MaxLineLength+1]; // declare int d[0..m, 0..n]

#define MIN(x,y) ((x) < (y) ? (x) : (y))
#define MAX(x,y) ((x) > (y) ? (x) : (y))
int main(int argc, char **argv) {

FILE *fp_inLINE;
FILE *fp_outLINE;

unsigned char workK[1024*32];
signed int workKoffset = -1;

unsigned long long FilesLEN;
unsigned long long k, k2, k3;
unsigned int LINE10len, wrdlen;

#if defined(_WIN32_ENVIRONMENT_)
      unsigned long long size_inLINESIXFOUR;
#else
      //size_t size_inLINESIXFOUR;
      unsigned long long size_inLINESIXFOUR;
#endif /* defined(_WIN32_ENVIRONMENT_)  */

unsigned long long size_inLINESIXFOURleftforparsing;
unsigned char wrdLOW[168*MaxLineLength+1+1]; // crlf
unsigned char wrdARG[168*MaxLineLength+1+1]; // crlf
unsigned char wrd[168*MaxLineLength+1+1]; // crlf
unsigned char wrdCACHED[MaxLineLength+1+1]; // crlf
unsigned char workbyte;
            int xgamsInCACHE = 0;
            int CACHEremainder;
            unsigned int memory_size;
//          unsigned char xgamsCACHE[MAXxgamsInCACHE*(MaxLineLength+1+1)]; // crlf
            unsigned char * xgamsCACHE;
            unsigned char * FOUNDinPTR;
            unsigned char * xgamsCACHEMEMCHR;
            unsigned char wrdCACHEDT1[MaxLineLength+1+1]; // crlf
            unsigned char wrdCACHEDT2[MaxLineLength+1+1]; // crlf
            unsigned char wrdCACHEDT3[MaxLineLength+1+1]; // crlf
            unsigned char wrdCACHEDT4[MaxLineLength+1+1]; // crlf
            unsigned char wrdCACHEDT5[MaxLineLength+1+1]; // crlf
            unsigned char wrdCACHEDT6[MaxLineLength+1+1]; // crlf
            unsigned char wrdCACHEDT7[MaxLineLength+1+1]; // crlf
            unsigned char wrdCACHEDT8[MaxLineLength+1+1]; // crlf
            unsigned char wrdCACHEDT9[MaxLineLength+1+1]; // crlf
            unsigned char wrdCACHEDT0[MaxLineLength+1+1]; // crlf
            unsigned char wrdCACHEDTa[MaxLineLength+1+1]; // crlf
            unsigned char wrdCACHEDTb[MaxLineLength+1+1]; // crlf
            unsigned char wrdCACHEDTc[MaxLineLength+1+1]; // crlf
            unsigned char wrdCACHEDTd[MaxLineLength+1+1]; // crlf
            unsigned char wrdCACHEDTe[MaxLineLength+1+1]; // crlf
            unsigned char wrdCACHEDTf[MaxLineLength+1+1]; // crlf
time_t t1, t2, t3;
clock_t clocks1, clocks2, clocks3, clocks4;
unsigned long long TotalLines=0;
unsigned long long WordsChecked=0;
unsigned long long DumpedLines=0;
// All these needed because of pseudo-bug (see comments at the bottom) uncrushed:
unsigned long long DumpedLines1=0;
unsigned long long DumpedLines2=0;
unsigned long long DumpedLines3=0;
unsigned long long DumpedLines4=0;
unsigned long long DumpedLines5=0;
unsigned long long DumpedLines6=0;
unsigned long long DumpedLines7=0;
unsigned long long DumpedLines8=0;
unsigned long long DumpedLines9=0;
```

```
unsigned long long DumpedLines0=0;
unsigned long long DumpedLinesa=0;
unsigned long long DumpedLinesb=0;
unsigned long long DumpedLinesc=0;
unsigned long long DumpedLinesd=0;
unsigned long long DumpedLinese=0;
unsigned long long DumpedLinesf=0;

unsigned long long TotalLines1=0;
unsigned long long TotalLines2=0;
unsigned long long TotalLines3=0;
unsigned long long TotalLines4=0;
unsigned long long TotalLines5=0;
unsigned long long TotalLines6=0;
unsigned long long TotalLines7=0;
unsigned long long TotalLines8=0;
unsigned long long TotalLines9=0;
unsigned long long TotalLines0=0;
unsigned long long TotalLinesa=0;
unsigned long long TotalLinesb=0;
unsigned long long TotalLinesc=0;
unsigned long long TotalLinesd=0;
unsigned long long TotalLinese=0;
unsigned long long TotalLinesf=0;

unsigned long long WordsChecked1=0;
unsigned long long WordsChecked2=0;
unsigned long long WordsChecked3=0;
unsigned long long WordsChecked4=0;
unsigned long long WordsChecked5=0;
unsigned long long WordsChecked6=0;
unsigned long long WordsChecked7=0;
unsigned long long WordsChecked8=0;
unsigned long long WordsChecked9=0;
unsigned long long WordsChecked0=0;
unsigned long long WordsCheckeda=0;
unsigned long long WordsCheckedb=0;
unsigned long long WordsCheckedc=0;
unsigned long long WordsCheckedd=0;
unsigned long long WordsCheckede=0;
unsigned long long WordsCheckedf=0;

int AtMostLevenshteinDistance;
unsigned int SkipHeuristic;
unsigned int StartingPosition;
           unsigned int WorkAreaLedgeT1;
           unsigned int WorkAreaRedgeT1;
           unsigned int WorkAreaLedgeT2;
           unsigned int WorkAreaRedgeT2;
           unsigned int WorkAreaLedgeT3;
           unsigned int WorkAreaRedgeT3;
           unsigned int WorkAreaLedgeT4;
           unsigned int WorkAreaRedgeT4;
           unsigned int WorkAreaLedgeT5;
           unsigned int WorkAreaRedgeT5;
           unsigned int WorkAreaLedgeT6;
           unsigned int WorkAreaRedgeT6;
           unsigned int WorkAreaLedgeT7;
           unsigned int WorkAreaRedgeT7;
           unsigned int WorkAreaLedgeT8;
           unsigned int WorkAreaRedgeT8;
           unsigned int WorkAreaLedgeT9;
           unsigned int WorkAreaRedgeT9;
           unsigned int WorkAreaLedgeT0;
           unsigned int WorkAreaRedgeT0;
           unsigned int WorkAreaLedgeTa;
           unsigned int WorkAreaRedgeTa;
           unsigned int WorkAreaLedgeTb;
           unsigned int WorkAreaRedgeTb;
```

```
            unsigned int WorkAreaLedgeTc;
            unsigned int WorkAreaRedgeTc;
            unsigned int WorkAreaLedgeTd;
            unsigned int WorkAreaRedgeTd;
            unsigned int WorkAreaLedgeTe;
            unsigned int WorkAreaRedgeTe;
            unsigned int WorkAreaLedgeTf;
            unsigned int WorkAreaRedgeTf;
char llTOaDigits[27]; // 9,223,372,036,854,775,807: 1(sign or carry)+19(digits)+1('\0')+6(,)
char llTOaDigits2[27]; // 9,223,372,036,854,775,807: 1(sign or carry)+19(digits)+1('\0')+6(,)
char llTOaDigits3[27]; // 9,223,372,036,854,775,807: 1(sign or carry)+19(digits)+1('\0')+6(,)

// IP
unsigned char WILDCARD_FAST_flag;
unsigned char WILDCARD_IP_flag;
unsigned char Exact_flag;
unsigned char Dump_flag;
unsigned char EXHAUSTIVE_flag;
int Melnitchka=0;
char *Auberge[4] = {"|\0","/\0","-\0","\\\0"};
int MAXboth;
char *ASCII010 = "\n\0";
int a;
unsigned long FREADclocks=0;
unsigned long long ticksTOTAL=0, ticksStart;

/*
Upper and lower bounds:
The Levenshtein distance has several simple upper and lower bounds that are useful in applications which compute many of them and compare them. These include:
- It is always at least the difference of the sizes of the two strings.
- It is at most the length of the longer string.
- It is zero if and only if the strings are identical.
- If the strings are the same size, the Hamming distance is an upper bound on the Levenshtein distance.
Possible improvements to this algorithm include:
- We can adapt the algorithm to use less space, O(m) instead of O(mn), since it only requires that the previous row and current row be stored at any one time.
- If we are only interested in the distance if it is smaller than a threshold k, then it suffices to compute a diagonal stripe of width 2k+1 in the matrix. In this way, the algorithm can be run in O(kl) time, where l is the length of the shortest
string.[1]
*/

// A very good resource:
// http://shaunwagner.com/writings_computer_levenshtein.html

            int i,j,m,n,l,BB;
            char s[] = "sitting";
            char t[] = "kitten";
            m = strlen(s);
            n = strlen(t);
//                      k         i         t         t         e         n
//          0           1         2         3         4         5         6
// s        1           1         2         3         4         5         6
// i        2           2         1         2         3         4         5
// t        3           3         2         1         2         3         4
// t        4           4         3         2         1         2         3
// i        5           5         4         3         2         2         3
// n        6           6         5         4         3         3         2
// g        7           7         6         5         4         4         3

/*
            for(i=0;i<=m;i++)
                        Levenshtein[i][0] = i;
            for(j=0;j<=n;j++)
                        Levenshtein[0][j] = j;
            for (j=1;j<=n;j++) {
                        for(i=1;i<=m;i++) {
                                    if(s[i-1] == t[j-1])
                                                Levenshtein[i][j] = Levenshtein[i-1][j-1];
                                    else
                                                Levenshtein[i][j] = MIN(MIN((Levenshtein[i-1][j]+1),(Levenshtein[i][j-1]+1)),(Levenshtein[i-1][j-1]+1));
                        }
```

```
        }
        printf("Levenshtein Distance: %d\n", Levenshtein[m][n]);
        exit (0);
*/

        printf("Kazahana, a superfast exact & wildcards & Levenshtein Distance (Wagner-Fischer) searcher, r. 1-++fix+nowait_critical_nixFIX_Wolfram+fixITER+EX+CS_fix_DEFINE, copyleft Kaze 2014-Dec-04.\n");
        if (argc != 4+1 && argc != 3+1) {
        printf("Usage: Kazahana [AtMostLevenshteinDistance][e] string textualfile MasterBufferSize\n");
        printf("Note0: MasterBufferSize is in KB, consider 1024, 3072, 7168 or bigger. Two additional flags were mapped on this value: all dump\n");
        printf("       lines (except fuzzy's) will have/lack pattern-source info when the number is even/odd respectively, see Examples #5 and #6.\n");
        printf("Note1: There are three regimes: exact, wildcards and fuzzy searches. First two kick in when 3 parameters are given, fuzzy when 4.\n");
        printf("Note2: What decides whether exact or wildcards? Of course presence of at least one wildcard. To see exact search see Example #4.\n");
        printf("Note3: Exact search hits with 'Railgun_Sekireigan_Wolfram'.\n");
        printf("Note4a: Incoming string is automatically lowercased for fuzzy searches i.e. they are case insensitive.\n");
        printf("Note4b: Incoming string is NOT automatically lowercased for wildcards searches when MasterBufferSize ends in 0..4 i.e. they are case sensitive.\n");
        printf("Note4c: Incoming string is automatically lowercased for wildcards searches when MasterBufferSize ends in 5..9 i.e. they are case insensitive.\n");
        printf("Note5: Incoming string could be up to %d/%d chars for Exact&Wildcard&ExhaustiveFuzzy/Fuzzy respectively.\n", MaxLineLength +(167*MaxLineLength), MaxLineLength);
        printf("Note5a: Since 2013-Nov-21 Levenshtein search exits not when the incoming line is bigger than %d chars, now it just skips longer lines.\n", MaxLineLength);
        printf("Note5b: Since 2013-Dec-05 Levenshtein search can be EXHAUSTIVE if LD is postfixed with 'e'.\n");
        printf("Note6: Incoming textualfile could be bigger than 4GB.\n");
        printf("Note7: Each line should end with [CR]LF, that is Windows or/and UNIX style.\n");
        printf("Note8: The dump goes to Kazahana.txt file.\n");
        printf("Note9a: Nine SLOW wildcards are available:\n");
        printf("        wildcard '*' any character(s) or empty,\n");
        printf("        wildcard '.' any ALPHA character(s) or empty,\n");
        printf("        wildcard '`' any NON-ALPHA character(s) or empty,\n");
        printf("        wildcard '@'/'#' any character {or empty}/{and not empty},\n");
        printf("        wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},\n");
        printf("        wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty}.\n");
        printf("Note9b: Two FAST wildcards are available:\n");
        printf("        wildcard '&' any character(s) or empty,\n");
        printf("        wildcard '+' any character and not empty.\n");
        printf("Note9c: Don't mix SLOW and FAST, the SLOW overrides the FAST, i.e. presence of at least one of the 9 wildcards cancels FAST mode.\n");
        printf("Example1: E:\\>Kazahana 0 ramjet MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd 1536\n");
        printf("Example2: E:\\>Kazahana 3 psychedlicize MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd 1536\n");
        printf("Example3: E:\\>Kazahana %cpsyched^^^^^^ize^%c MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd 1536\n", 34, 34);
        printf("Example4: E:\\>Kazahana %cmetal fatigue%c enwiki-20121201-pages-articles.xml 7168\n", 34, 34);
        printf("Example5: E:\\>Kazahana %cout^^^^^^^^^^^^ize*%c MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd 1536\n", 34, 34);
        printf("          E:\\>type Kazahana.txt\n");
        printf("          [out^^^^^^^^^^^^ize*] outhyperbolize /MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd/\n");
        printf("          [out^^^^^^^^^^^^ize*] outsize /MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd/\n");
        printf("          [out^^^^^^^^^^^^ize*] outsized /MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd/\n");
        printf("          [out^^^^^^^^^^^^ize*] outstrategize /MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd/\n");
        printf("          [out^^^^^^^^^^^^ize*] outtyrannize /MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd/\n");
        printf("Example6: E:\\>Kazahana %cout^^^^^^^^^^^^ize*%c MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd 1537\n", 34, 34);
        printf("          E:\\>type Kazahana.txt\n");
        printf("          outhyperbolize\n");
        printf("          outsize\n");
        printf("          outsized\n");
        printf("          outstrategize\n");
        printf("          outtyrannize\n");
        printf("Example7: E:\\>Kazahana 2e edelvais MASAKARI_General-Purpose_Grade_English_Wordlist.wrd 1024\n");
        printf("          E:\\>type Kazahana.txt\n");
        printf("          bordelais\n");
        printf("          bordelaise\n");
        printf("          edelweiss\n");
        printf("          edelweisses\n");
        printf("          foredevised\n");
        printf("          predellas\n");
        printf("          psychedelicism\n");
        (void) time(&t1);
        (void) time(&t3);
        while (t3 == t1) (void) time(&t3);
        t1=t3;
        clocks1 = clock();
#if defined(_icl_mumbo_jumbo_)
ticksStart = GetRDTSC();
#endif
        while (t3 != t1+2) (void) time(&t3);
```

```
#if defined(_icl_mumbo_jumbo_)
ticksTOTAL = ticksTOTAL + GetRDTSC() - ticksStart;
#endif
            printf("Info1: One second seems to have %s clocks.\n", _ui64toaKAZEcomma((clock()-clocks1)/2, llTOaDigits, 10));
#if defined(_icl_mumbo_jumbo_)
            printf("Info2: This CPU seems to be working at %s MHz.\n", _ui64toaKAZEcomma(ticksTOTAL/2/1000000, llTOaDigits, 10));
#endif
            } else {
            if (argc == 3+1) WILDCARD_IP_flag = 1; else WILDCARD_IP_flag = 0;
            (void) time(&t1);
            (void) time(&t3);
            while (t3 == t1) (void) time(&t3);
            t1=t3;
            clocks1 = clock();

if (argc == 4+1) {
            // ASCII 48 is '0'
            // ASCII 52 is '4'
            // ASCII 57 is '9'
            if ( (*(unsigned char *)(argv[4]+strlen(argv[4])-1)-48) > 4) CaseSensitiveWildcardMatching_flag=0; else CaseSensitiveWildcardMatching_flag=1;
}
if (argc == 3+1) {
            // ASCII 48 is '0'
            // ASCII 52 is '4'
            // ASCII 57 is '9'
            if ( (*(unsigned char *)(argv[3]+strlen(argv[3])-1)-48) > 4) CaseSensitiveWildcardMatching_flag=0; else CaseSensitiveWildcardMatching_flag=1;
}

            //n = strlen(argv[2-WILDCARD_IP_flag]);
            n = 0;
            while (argv[2-WILDCARD_IP_flag][n]) {
            if (CaseSensitiveWildcardMatching_flag == 1)
                        wrdARG[ n ] = ( argv[2-WILDCARD_IP_flag][n] );
            else
                        wrdARG[ n ] = KAZE_tolower( argv[2-WILDCARD_IP_flag][n] );
                        n++;
                        if (n>MaxLineLength)
                        { printf( "Kazahana: Incoming xgram exceeding the limit.\n" ); return( 1 ); }
            }
            wrdARG[ n ] = 0; // Needed 'cause wrdARG is not zeroed!

            EXHAUSTIVE_flag = 0;
            if (!WILDCARD_IP_flag) {
                        AtMostLevenshteinDistance = atoi(argv[1]);
                        if ( KAZE_tolower( argv[1][strlen(argv[1])-1] ) == 'e') EXHAUSTIVE_flag = 1;
            }

            Exact_flag = 0;
            WILDCARD_FAST_flag = 2;
            if (WILDCARD_IP_flag) {
            if ( memchrKAZE(argv[2-WILDCARD_IP_flag], &TAGfreeFAST[0], n) == NULL && \
                 memchrKAZE(argv[2-WILDCARD_IP_flag], &TAGfreeFAST[1], n) == NULL ) Exact_flag = 1;
            else WILDCARD_FAST_flag = 1;
            }

            if (EXHAUSTIVE_flag && n<=AtMostLevenshteinDistance)
            { printf( "Kazahana: In EXHAUSTIVE mode 'PatternLength' should be bigger than 'AtMostLevenshteinDistance'.\n" ); return( 1 ); }

            // The way it is done now the SLOW overrides the FAST, meaning that presence of at least one of the 9 wildcards trumps the FAST mode.

            if (WILDCARD_IP_flag) {
            if ( memchrKAZE(argv[2-WILDCARD_IP_flag], &TAGfree[0], n) == NULL && \
                 memchrKAZE(argv[2-WILDCARD_IP_flag], &TAGfree[1], n) == NULL && \
                 memchrKAZE(argv[2-WILDCARD_IP_flag], &TAGfree[2], n) == NULL && \
                 memchrKAZE(argv[2-WILDCARD_IP_flag], &TAGfree[3], n) == NULL && \
                 memchrKAZE(argv[2-WILDCARD_IP_flag], &TAGfree[4], n) == NULL && \
                 memchrKAZE(argv[2-WILDCARD_IP_flag], &TAGfree[5], n) == NULL && \
                 memchrKAZE(argv[2-WILDCARD_IP_flag], &TAGfree[6], n) == NULL && \
                 memchrKAZE(argv[2-WILDCARD_IP_flag], &TAGfree[7], n) == NULL && \
```

```
                memchrKAZE(argv[2-WILDCARD_IP_flag], &TAGfree[8], n) == NULL ) Exact_flag = 1;
          else WILDCARD_FAST_flag = 0;
          }

          if (WILDCARD_FAST_flag == 1) Exact_flag = 0;
          if (WILDCARD_FAST_flag == 0) Exact_flag = 0;

          if (WILDCARD_FAST_flag != 2) {
                    if (CaseSensitiveWildcardMatching_flag == 1) printf("Enforcing Case Sensitive wildcard mode ...\n");
                    if (CaseSensitiveWildcardMatching_flag == 0) printf("Enforcing Case Insensitive wildcard mode ...\n");
          }

          if (WILDCARD_FAST_flag == 1) printf("Enforcing FAST wildcard mode ...\n");
          if (WILDCARD_FAST_flag == 0) printf("Enforcing SLOW wildcard mode ...\n");

          if (Exact_flag && n==1)
          { printf( "Kazahana: Incoming xgram should be longer than 1 char.\n" ); return( 1 ); }
          if (n>255)
          { printf( "Kazahana: Incoming xgram should be shorter than 256 chars.\n" ); return( 1 ); }

          if (Exact_flag) {
          n = 0;
          while (argv[2-WILDCARD_IP_flag][n]) {
                    wrdARG[ n ] = argv[2-WILDCARD_IP_flag][n];
                    n++;
                    if (n>MaxLineLength)
                    { printf( "Kazahana: Incoming xgram exceeding the limit.\n" ); return( 1 ); }
          }
          wrdARG[ n ] = 0; // Needed 'cause wrdARG is not zeroed!
          }

printf("Pattern: %s\n",wrdARG);

          if (Exact_flag) {
// Initializing Gulliver's arrays:
for (a=0; a < 256; a++) {bm_bc[a]=n; bm_bc2nd[a]=n+1;}
for (a=0; a < n-1; a++) bm_bc[argv[2-WILDCARD_IP_flag][a]]=n-a-1;
for (a=0; a < n; a++) bm_bc2nd[argv[2-WILDCARD_IP_flag][a]]=n-a;
for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]= n-1;} // 'memset' if not optimized
for (a=0; a < n-1; a++) bm_Horspool_Order2[*(unsigned short *)(argv[2-WILDCARD_IP_flag]+a)]=a; // Rightmost appearance/position is needed
// Bari arrays:
for (a=0; a < 256*256; a++) {bm_Horspool_Order2[a]=0;}
//for (a=0; a < n-1; a++) bm_Horspool_Order2[*(unsigned short *)(argv[2-WILDCARD_IP_flag]+a)]=1;
          }

#ifdef Commence_OpenMP
                    printf("omp_get_num_procs( ) = %d\n", omp_get_num_procs( ));
                    printf("omp_get_max_threads( ) = %d\n", omp_get_max_threads( ));
                    //printf("Enforcing DUAD i.e. double-threads ...\n");
                    //printf("Enforcing QUAD i.e. quadruple-threads ...\n");
                    //printf("Enforcing OCTAD i.e. octuple-threads ...\n");
                    printf("Enforcing HEXADECAD i.e. hexadecuple-threads ...\n");
#else
                    printf("Enforcing MONAD i.e. single-thread ...\n");
#endif

/*
Sadly the topic is not fully covered, so here my attempt to fill the gap follows:
noun, adjective, verb, adverb

single/singles/singlet/singlets, single/singled, single/singles/singled/singling, singly
double/doubles/doublet/doublets, double/doubled, double/doubles/doubled/doubling, doubly
triple/triples/triplet/triplets, triple/tripled, triple/triples/tripled/tripling, triply
quadruple/quadruples/quadruplet/quadruplets, quadruple/quadrupled, quadruple/quadruples/quadrupled/quadrupling, quadruply
quintuple/quintuples/quintuplet/quintuplets, quintuple/quintupled, quintuple/quintuples/quintupled/quintupling, quintuply
sextuple/sextuples/sextuplet/sextuplets, sextuple/sextupled, sextuple/sextuples/sextupled/sextupling, sextuply
septuple/septuples/septuplet/septuplets, septuple/septupled, septuple/septuples/septupled/septupling, septuply
octuple/octuples/octuplet/octuplets, octuple/octupled, octuple/octuples/octupled/octupling, octuply
nonuple/nonuples/nonuplet/nonuplets, nonuple/nonupled, nonuple/nonuples/nonupled/nonupling, nonuply
```

```
decuple/decuples/decuplet/decuplets, decuple/decupled, decuple/decuples/decupled/decupling, decuply

And one adverbial example given in HERITAGE:
triply
adv.

1. In three ways: As an actor, singer, and juggler, she was triply qualified for the role.
2. To a triple degree: a triply redundant navigational system.
3. Three times: Prices were triply inflated.

It is obvious that other analogs are in use (are preferred):
dual instead of double
quad instead of quadruple
hexa instead of sextuple

This prompts for dumping the *ad analogs (NOUNS!):
monad
duad/dyad
triad
quad/tetrad
pentad
hexad
heptad/hebdomad (for 7)
octad
ennead (for 9)
decad (for 10)
...
duodecad (for 12)
I think my choice hexadecad is even better (it refers to a group) than hexadecuple/sexdecuple couple, don't you think? (for 16)
...
chiliad (for 1000)
myriad (for 10000)
*/

// MAXboth = MaxLineLength +1+1 +(167*WILDCARD_IP_flag*MaxLineLength); // Buggy line, fixed with next one in r. ...CS_fix
           if (WILDCARD_IP_flag) {
                      MAXboth = MaxLineLength +1+1 +(167*WILDCARD_IP_flag*MaxLineLength);
           } else {
                      MAXboth = MaxLineLength +1+1 +(167*EXHAUSTIVE_flag*MaxLineLength);
           }

if (argc == 4+1) {
           CACHEsize = atoi(argv[4])*(1<<10);
           if (atoi(argv[4]) & 1) Dump_flag=0; else Dump_flag=1;
}
if (argc == 3+1) {
           CACHEsize = atoi(argv[3])*(1<<10);
           if (atoi(argv[3]) & 1) Dump_flag=0; else Dump_flag=1;
}

memory_size = CACHEsize+65;
printf( "Allocating Master-Buffer %luKB ... ", (memory_size>>10) );
xgamsCACHE = (char *)malloc( memory_size );
if( xgamsCACHE == NULL )
{ puts( "\nKazahana: Needed memory allocation denied!\n" ); return( 1 ); }
printf( "OK\n");
if (64 - (((size_t)xgamsCACHE) % 64) != 0)
           xgamsCACHE = xgamsCACHE + 64 - (((size_t)xgamsCACHE) % 64);

           for(i=0;i<=MaxLineLength;i++) {
                      LevenshteinT1[i][0] = i;
                      LevenshteinT1[0][i] = i;
                      LevenshteinT2[i][0] = i;
                      LevenshteinT2[0][i] = i;
                      LevenshteinT3[i][0] = i;
                      LevenshteinT3[0][i] = i;
                      LevenshteinT4[i][0] = i;
                      LevenshteinT4[0][i] = i;
                      LevenshteinT5[i][0] = i;
```

```
                        LevenshteinT5[0][i] = i;
                        LevenshteinT6[i][0] = i;
                        LevenshteinT6[0][i] = i;
                        LevenshteinT7[i][0] = i;
                        LevenshteinT7[0][i] = i;
                        LevenshteinT8[i][0] = i;
                        LevenshteinT8[0][i] = i;
                        LevenshteinT9[i][0] = i;
                        LevenshteinT9[0][i] = i;
                        LevenshteinT0[i][0] = i;
                        LevenshteinT0[0][i] = i;
                        LevenshteinTa[i][0] = i;
                        LevenshteinTa[0][i] = i;
                        LevenshteinTb[i][0] = i;
                        LevenshteinTb[0][i] = i;
                        LevenshteinTc[i][0] = i;
                        LevenshteinTc[0][i] = i;
                        LevenshteinTd[i][0] = i;
                        LevenshteinTd[0][i] = i;
                        LevenshteinTe[i][0] = i;
                        LevenshteinTe[0][i] = i;
                        LevenshteinTf[i][0] = i;
                        LevenshteinTf[0][i] = i;
            }

// Out of courtesy...
memset (wrdCACHEDT1, 0, MaxLineLength+1+1);
memset (wrdCACHEDT2, 0, MaxLineLength+1+1);
memset (wrdCACHEDT3, 0, MaxLineLength+1+1);
memset (wrdCACHEDT4, 0, MaxLineLength+1+1);
memset (wrdCACHEDT5, 0, MaxLineLength+1+1);
memset (wrdCACHEDT6, 0, MaxLineLength+1+1);
memset (wrdCACHEDT7, 0, MaxLineLength+1+1);
memset (wrdCACHEDT8, 0, MaxLineLength+1+1);
memset (wrdCACHEDT9, 0, MaxLineLength+1+1);
memset (wrdCACHEDT0, 0, MaxLineLength+1+1);
memset (wrdCACHEDTa, 0, MaxLineLength+1+1);
memset (wrdCACHEDTb, 0, MaxLineLength+1+1);
memset (wrdCACHEDTc, 0, MaxLineLength+1+1);
memset (wrdCACHEDTd, 0, MaxLineLength+1+1);
memset (wrdCACHEDTe, 0, MaxLineLength+1+1);
memset (wrdCACHEDTf, 0, MaxLineLength+1+1);

if( ( fp_inLINE = fopen( argv[3-WILDCARD_IP_flag], "rb" ) ) == NULL )
{ printf( "Kazahana: Can't open %s file.\n", argv[3-WILDCARD_IP_flag] ); return( 1 ); }

if( ( fp_outLINE = fopen( "Kazahana.txt", "wb" ) ) == NULL )
{ printf( "Kazahana: Can't open Kazahana.txt file.\n" ); return( 1 ); }

#if defined(_WIN32_ENVIRONMENT_)
   // 64bit:
_lseeki64( fileno(fp_inLINE), 0L, SEEK_END );
size_inLINESIXFOUR = _telli64( fileno(fp_inLINE) );
_lseeki64( fileno(fp_inLINE), 0L, SEEK_SET );
#else
   // 64bit:
fseeko( fp_inLINE, 0L, SEEK_END );
size_inLINESIXFOUR = ftello( fp_inLINE );
fseeko( fp_inLINE, 0L, SEEK_SET );
#endif /* defined(_WIN32_ENVIRONMENT_)  */

// MT PARSING MT PARSING MT PARSING MT PARSING MT PARSING MT PARSING MT PARSING MT PARSING MT PARSING MT PARSING MT PARSING MT PARSING [
size_inLINESIXFOURleftforparsing = size_inLINESIXFOUR;
CACHEremainder = 0;
while (size_inLINESIXFOURleftforparsing >= CACHEsize-CACHEremainder) {
            clocks4 = clock();
#if defined(_icl_mumbo_jumbo_)
ticksStart = GetRDTSC();
#endif
```

```
            fread( xgamsCACHE+CACHEremainder, 1, CACHEsize-CACHEremainder, fp_inLINE );
#if defined(_icl_mumbo_jumbo_)
ticksTOTAL = ticksTOTAL + GetRDTSC() - ticksStart;
#endif
            clocks3 = clock();
            FREADclocks = FREADclocks + (clocks3-clocks4);
            size_inLINESIXFOURleftforparsing = size_inLINESIXFOURleftforparsing - (CACHEsize-CACHEremainder);
            Melnitchka = Melnitchka & 3; // 0 1 2 3: 00 01 10 11
            printf( "%s; Speed: %s bytes/clock; Traversed: %s bytes\r", Auberge[Melnitchka++], _ui64toaKAZEzerocomma(((size_inLINESIXFOUR-size_inLINESIXFOURleftforparsing)>>0)/((long)(clocks3 - clocks1 + 1)), llTOaDigits, 10) +12,
_ui64toaKAZEcomma((size_inLINESIXFOUR-size_inLINESIXFOURleftforparsing), llTOaDigits2, 10));
            CACHEremainder = 0;
            while ( xgamsCACHE[ CACHEsize-1-CACHEremainder ] != 10 ) {
                        if ( CACHEsize-1-CACHEremainder == 0 ) { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); }
                        CACHEremainder++;
            }
            // Working area: xgamsCACHE..xgamsCACHE+CACHEsize-1-CACHEremainder
            //fwrite( xgamsCACHE, 1, xgamsCACHE+CACHEsize-1-CACHEremainder - xgamsCACHE +1, fp_outLINE ); //DELDEL
            //...
// 1st thread 0..1*(MAXxgamsInCACHE/THREADSnumber)-1
// 2nd thread 1*(MAXxgamsInCACHE/THREADSnumber)-1+1..2*(MAXxgamsInCACHE/THREADSnumber)-1
// 3rd thread 2*(MAXxgamsInCACHE/THREADSnumber)-1+1..3*(MAXxgamsInCACHE/THREADSnumber)-1
            // WorkArea pair is the left/right offset in xgamsCACHE pool for each thread, offsets are better than pointers because they are 4bytes not 8
            // CAUTION: An uncrushed bug: the partitions can be without ASCII 010 at all! CRASH!

            //WorkAreaRedgeTe = 15*(CACHEsize/THREADSnumber)-1;
if ( 15*(CACHEsize/THREADSnumber)-1 >= CACHEsize-1-CACHEremainder ) { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); } // parse BUGfixed: ...nixFIX

             WorkAreaLedgeT1 = 0;
            WorkAreaRedgeT1 = 1*(CACHEsize/THREADSnumber)-1;
//if ( WorkAreaRedgeT1 >= CACHEsize-1-CACHEremainder ) { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); } // parse BUGfixed: ...nixFIX
            //while ( xgamsCACHE[ WorkAreaRedgeT1 ] != 10 ) WorkAreaRedgeT1++; // SLOW!
            xgamsCACHEMEMCHR = memchr(&xgamsCACHE[ WorkAreaRedgeT1 ], 10, CACHEsize/THREADSnumber/24*28);
            if ( xgamsCACHEMEMCHR == NULL )   { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); }
            // Line below triggers cast warning, no worries since the result of subtraction lies within 'unsigned int' i.e. chunk is always < 4GB:
            WorkAreaRedgeT1 = WorkAreaRedgeT1 + (unsigned int)( (unsigned long long)xgamsCACHEMEMCHR - (unsigned long long)(&xgamsCACHE[ WorkAreaRedgeT1 ]) );

            //printf("%s\n", argv[0] );
            //printf("%s\n", memchr(argv[0], 'K', 300) );
            //printf("%d\n", (unsigned int) (memchr(argv[0], 65, 300)) - (unsigned int)(argv[0]) );
            //Kazahana_r1-+_HEXADECAD-Threads_IntelV12.exe
            //Kazahana_r1-+_HEXADECAD-Threads_IntelV12.exe
            // 17

            WorkAreaLedgeT2 = WorkAreaRedgeT1+1;
            WorkAreaRedgeT2 = 2*(CACHEsize/THREADSnumber)-1;
//if ( WorkAreaRedgeT2 >= CACHEsize-1-CACHEremainder ) { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); } // parse BUGfixed: ...nixFIX
            //while ( xgamsCACHE[ WorkAreaRedgeT2 ] != 10 ) WorkAreaRedgeT2++;
            xgamsCACHEMEMCHR = memchr(&xgamsCACHE[ WorkAreaRedgeT2 ], 10, CACHEsize/THREADSnumber/24*28);
            if ( xgamsCACHEMEMCHR == NULL )   { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); }
            WorkAreaRedgeT2 = (unsigned int)( WorkAreaRedgeT2 + (unsigned long long)xgamsCACHEMEMCHR - (unsigned long long)(&xgamsCACHE[ WorkAreaRedgeT2 ]) );
            WorkAreaLedgeT3 = WorkAreaRedgeT2+1;
            WorkAreaRedgeT3 = 3*(CACHEsize/THREADSnumber)-1;
//if ( WorkAreaRedgeT3 >= CACHEsize-1-CACHEremainder ) { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); } // parse BUGfixed: ...nixFIX
            //while ( xgamsCACHE[ WorkAreaRedgeT3 ] != 10 ) WorkAreaRedgeT3++;
            xgamsCACHEMEMCHR = memchr(&xgamsCACHE[ WorkAreaRedgeT3 ], 10, CACHEsize/THREADSnumber/24*28);
            if ( xgamsCACHEMEMCHR == NULL )   { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); }
            WorkAreaRedgeT3 = (unsigned int)( WorkAreaRedgeT3 + (unsigned long long)xgamsCACHEMEMCHR - (unsigned long long)(&xgamsCACHE[ WorkAreaRedgeT3 ]) );
            WorkAreaLedgeT4 = WorkAreaRedgeT3+1;
            WorkAreaRedgeT4 = 4*(CACHEsize/THREADSnumber)-1;
//if ( WorkAreaRedgeT4 >= CACHEsize-1-CACHEremainder ) { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); } // parse BUGfixed: ...nixFIX
            //while ( xgamsCACHE[ WorkAreaRedgeT4 ] != 10 ) WorkAreaRedgeT4++;
            xgamsCACHEMEMCHR = memchr(&xgamsCACHE[ WorkAreaRedgeT4 ], 10, CACHEsize/THREADSnumber/24*28);
            if ( xgamsCACHEMEMCHR == NULL )   { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); }
            WorkAreaRedgeT4 = (unsigned int)( WorkAreaRedgeT4 + (unsigned long long)xgamsCACHEMEMCHR - (unsigned long long)(&xgamsCACHE[ WorkAreaRedgeT4 ]) );
            WorkAreaLedgeT5 = WorkAreaRedgeT4+1;
            WorkAreaRedgeT5 = 5*(CACHEsize/THREADSnumber)-1;
//if ( WorkAreaRedgeT5 >= CACHEsize-1-CACHEremainder ) { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); } // parse BUGfixed: ...nixFIX
            //while ( xgamsCACHE[ WorkAreaRedgeT5 ] != 10 ) WorkAreaRedgeT5++;
            xgamsCACHEMEMCHR = memchr(&xgamsCACHE[ WorkAreaRedgeT5 ], 10, CACHEsize/THREADSnumber/24*28);
```

```
            if ( xgamsCACHEMEMCHR == NULL )   { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); }
            WorkAreaRedgeT5 = (unsigned int)( WorkAreaRedgeT5 + (unsigned long long)xgamsCACHEMEMCHR - (unsigned long long)(&xgamsCACHE[ WorkAreaRedgeT5 ]) );
            WorkAreaLedgeT6 = WorkAreaRedgeT5+1;
            WorkAreaRedgeT6 = 6*(CACHEsize/THREADSnumber)-1;
//if ( WorkAreaRedgeT6 >= CACHEsize-1-CACHEremainder ) { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); } // parse BUGfixed: ...nixFIX
            //while ( xgamsCACHE[ WorkAreaRedgeT6 ] != 10 ) WorkAreaRedgeT6++;
            xgamsCACHEMEMCHR = memchr(&xgamsCACHE[ WorkAreaRedgeT6 ], 10, CACHEsize/THREADSnumber/24*28);
            if ( xgamsCACHEMEMCHR == NULL )   { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); }
            WorkAreaRedgeT6 = (unsigned int)( WorkAreaRedgeT6 + (unsigned long long)xgamsCACHEMEMCHR - (unsigned long long)(&xgamsCACHE[ WorkAreaRedgeT6 ]) );
            WorkAreaLedgeT7 = WorkAreaRedgeT6+1;
            WorkAreaRedgeT7 = 7*(CACHEsize/THREADSnumber)-1;
//if ( WorkAreaRedgeT7 >= CACHEsize-1-CACHEremainder ) { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); } // parse BUGfixed: ...nixFIX
            //while ( xgamsCACHE[ WorkAreaRedgeT7 ] != 10 ) WorkAreaRedgeT7++;
            xgamsCACHEMEMCHR = memchr(&xgamsCACHE[ WorkAreaRedgeT7 ], 10, CACHEsize/THREADSnumber/24*28);
            if ( xgamsCACHEMEMCHR == NULL )   { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); }
            WorkAreaRedgeT7 = (unsigned int)( WorkAreaRedgeT7 + (unsigned long long)xgamsCACHEMEMCHR - (unsigned long long)(&xgamsCACHE[ WorkAreaRedgeT7 ]) );
            WorkAreaLedgeT8 = WorkAreaRedgeT7+1;
            WorkAreaRedgeT8 = 8*(CACHEsize/THREADSnumber)-1;
//if ( WorkAreaRedgeT8 >= CACHEsize-1-CACHEremainder ) { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); } // parse BUGfixed: ...nixFIX
            //while ( xgamsCACHE[ WorkAreaRedgeT8 ] != 10 ) WorkAreaRedgeT8++;
            xgamsCACHEMEMCHR = memchr(&xgamsCACHE[ WorkAreaRedgeT8 ], 10, CACHEsize/THREADSnumber/24*28);
            if ( xgamsCACHEMEMCHR == NULL )   { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); }
            WorkAreaRedgeT8 = (unsigned int)( WorkAreaRedgeT8 + (unsigned long long)xgamsCACHEMEMCHR - (unsigned long long)(&xgamsCACHE[ WorkAreaRedgeT8 ]) );
            WorkAreaLedgeT9 = WorkAreaRedgeT8+1;
            WorkAreaRedgeT9 = 9*(CACHEsize/THREADSnumber)-1;
//if ( WorkAreaRedgeT9 >= CACHEsize-1-CACHEremainder ) { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); } // parse BUGfixed: ...nixFIX
            //while ( xgamsCACHE[ WorkAreaRedgeT9 ] != 10 ) WorkAreaRedgeT9++;
            xgamsCACHEMEMCHR = memchr(&xgamsCACHE[ WorkAreaRedgeT9 ], 10, CACHEsize/THREADSnumber/24*28);
            if ( xgamsCACHEMEMCHR == NULL )   { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); }
            WorkAreaRedgeT9 = (unsigned int)( WorkAreaRedgeT9 + (unsigned long long)xgamsCACHEMEMCHR - (unsigned long long)(&xgamsCACHE[ WorkAreaRedgeT9 ]) );
            WorkAreaLedgeT0 = WorkAreaRedgeT9+1;
            WorkAreaRedgeT0 = 10*(CACHEsize/THREADSnumber)-1;
//if ( WorkAreaRedgeT0 >= CACHEsize-1-CACHEremainder ) { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); } // parse BUGfixed: ...nixFIX
            //while ( xgamsCACHE[ WorkAreaRedgeT0 ] != 10 ) WorkAreaRedgeT0++;
            xgamsCACHEMEMCHR = memchr(&xgamsCACHE[ WorkAreaRedgeT0 ], 10, CACHEsize/THREADSnumber/24*28);
            if ( xgamsCACHEMEMCHR == NULL )   { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); }
            WorkAreaRedgeT0 = (unsigned int)( WorkAreaRedgeT0 + (unsigned long long)xgamsCACHEMEMCHR - (unsigned long long)(&xgamsCACHE[ WorkAreaRedgeT0 ]) );
            WorkAreaLedgeTa = WorkAreaRedgeT0+1;
            WorkAreaRedgeTa = 11*(CACHEsize/THREADSnumber)-1;
//if ( WorkAreaRedgeTa >= CACHEsize-1-CACHEremainder ) { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); } // parse BUGfixed: ...nixFIX
            //while ( xgamsCACHE[ WorkAreaRedgeTa ] != 10 ) WorkAreaRedgeTa++;
            xgamsCACHEMEMCHR = memchr(&xgamsCACHE[ WorkAreaRedgeTa ], 10, CACHEsize/THREADSnumber/24*28);
            if ( xgamsCACHEMEMCHR == NULL )   { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); }
            WorkAreaRedgeTa = (unsigned int)( WorkAreaRedgeTa + (unsigned long long)xgamsCACHEMEMCHR - (unsigned long long)(&xgamsCACHE[ WorkAreaRedgeTa ]) );
            WorkAreaLedgeTb = WorkAreaRedgeTa+1;
            WorkAreaRedgeTb = 12*(CACHEsize/THREADSnumber)-1;
//if ( WorkAreaRedgeTb >= CACHEsize-1-CACHEremainder ) { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); } // parse BUGfixed: ...nixFIX
            //while ( xgamsCACHE[ WorkAreaRedgeTb ] != 10 ) WorkAreaRedgeTb++;
            xgamsCACHEMEMCHR = memchr(&xgamsCACHE[ WorkAreaRedgeTb ], 10, CACHEsize/THREADSnumber/24*28);
            if ( xgamsCACHEMEMCHR == NULL )   { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); }
            WorkAreaRedgeTb = (unsigned int)( WorkAreaRedgeTb + (unsigned long long)xgamsCACHEMEMCHR - (unsigned long long)(&xgamsCACHE[ WorkAreaRedgeTb ]) );
            WorkAreaLedgeTc = WorkAreaRedgeTb+1;
            WorkAreaRedgeTc = 13*(CACHEsize/THREADSnumber)-1;
//if ( WorkAreaRedgeTc >= CACHEsize-1-CACHEremainder ) { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); } // parse BUGfixed: ...nixFIX
            //while ( xgamsCACHE[ WorkAreaRedgeTc ] != 10 ) WorkAreaRedgeTc++;
            xgamsCACHEMEMCHR = memchr(&xgamsCACHE[ WorkAreaRedgeTc ], 10, CACHEsize/THREADSnumber/24*28);
            if ( xgamsCACHEMEMCHR == NULL )   { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); }
            WorkAreaRedgeTc = (unsigned int)( WorkAreaRedgeTc + (unsigned long long)xgamsCACHEMEMCHR - (unsigned long long)(&xgamsCACHE[ WorkAreaRedgeTc ]) );
            WorkAreaLedgeTd = WorkAreaRedgeTc+1;
            WorkAreaRedgeTd = 14*(CACHEsize/THREADSnumber)-1;
//if ( WorkAreaRedgeTd >= CACHEsize-1-CACHEremainder ) { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); } // parse BUGfixed: ...nixFIX
            //while ( xgamsCACHE[ WorkAreaRedgeTd ] != 10 ) WorkAreaRedgeTd++;
            xgamsCACHEMEMCHR = memchr(&xgamsCACHE[ WorkAreaRedgeTd ], 10, CACHEsize/THREADSnumber/24*28);
            if ( xgamsCACHEMEMCHR == NULL )   { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); }
            WorkAreaRedgeTd = (unsigned int)( WorkAreaRedgeTd + (unsigned long long)xgamsCACHEMEMCHR - (unsigned long long)(&xgamsCACHE[ WorkAreaRedgeTd ]) );
            WorkAreaLedgeTe = WorkAreaRedgeTd+1;
            WorkAreaRedgeTe = 15*(CACHEsize/THREADSnumber)-1;
//if ( WorkAreaRedgeTe >= CACHEsize-1-CACHEremainder ) { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); } // parse BUGfixed: ...nixFIX
            //while ( xgamsCACHE[ WorkAreaRedgeTe ] != 10 ) WorkAreaRedgeTe++;
```

```
            xgamsCACHEMEMCHR = memchr(&xgamsCACHE[ WorkAreaRedgeTe ], 10, CACHEsize/THREADSnumber/24*28);
            if ( xgamsCACHEMEMCHR == NULL )   { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); }
            WorkAreaRedgeTe = (unsigned int)( WorkAreaRedgeTe + (unsigned long long)xgamsCACHEMEMCHR - (unsigned long long)(&xgamsCACHE[ WorkAreaRedgeTe ]) );
            WorkAreaLedgeTf = WorkAreaRedgeTe+1;
             WorkAreaRedgeTf = CACHEsize-1-CACHEremainder;
            if ( WorkAreaLedgeTf >= WorkAreaRedgeTf ) { printf( "\nKazahana: Failure! Too long line encountered. Increase Master-Buffer size.\n" ); return( 1 ); }
            //...

#ifdef Commence_OpenMP
#pragma omp parallel shared(fp_outLINE,n,AtMostLevenshteinDistance,MAXboth,WILDCARD_IP_flag,Exact_flag,WILDCARD_FAST_flag,EXHAUSTIVE_flag) private(wrdlen,k,workbyte,m,wrd,wrdLOW,i,j,l,BB,SkipHeuristic,StartingPosition,FOUNDinPTR)
#endif
{
#ifdef Commence_OpenMP
  #pragma omp sections nowait
#endif
    {

// 1st thread
#ifdef Commence_OpenMP
    #pragma omp section
#endif
            {

                                                if (Exact_flag) {
// WHOLE buffer at once not line-by-line [[[[[[[[ Since r.1-++
                                                            k = WorkAreaLedgeT1;
                                                            while ( k < WorkAreaRedgeT1 ) {
                                                            #ifdef RG7Gulliver
                                                                        //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_1(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT1-k+1, n);
                                                                        FOUNDinPTR = Railgun_Sekireigan_Wolfram_1(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT1-k+1, n);
                                                            #else
                                                                        FOUNDinPTR = Railgun_Quadruplet_7_1(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT1-k+1, n);
                                                            #endif
                                                            // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                            // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                            //if ((long)( FOUNDinPTR - &xgamsCACHE[k] )>=0) {
                                                            if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(&xgamsCACHE[k])) {
                                                                        i = k + (long)( FOUNDinPTR - &xgamsCACHE[k] ); j = i;
                                                                        while (xgamsCACHE[j] != 10) {++j;} // Works both on UNIX(LF) and Windows(CRLF)
                                                                        while (i > k && xgamsCACHE[i-1] != 10) {--i;}
                                                                        k = j+1; // Should "point" to first symbol after the dumped fragment.

                                                                        //fwrite( &xgamsCACHE[i], j - i + 1, 1, fp_outLINE ); DumpedLines1++;
                                                                        if (j - i + 1 <= 168*MaxLineLength) { // fix for 1-++
                                                                        memcpy(wrd, &xgamsCACHE[i], j - i + 1); wrd[j - i + 1]=0; DumpedLines1++;
                                                                        j--; if (wrd[j - i + 1]==10) wrd[j - i + 1]=0;
                                                                        j--; if (wrd[j - i + 1]==13) wrd[j - i + 1]=0;
                                                            // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                            #pragma omp critical
#endif
                                                            {
                                                                        if (Dump_flag)
                                                                        fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]);
                                                                        else
                                                                        fprintf( fp_outLINE, "%s\r\n", wrd);
                                                            }
                                                                        }
                                                            } else k = WorkAreaRedgeT1;
                                                            } // while
// WHOLE buffer at once not line-by-line ]]]]]]]] Since r.1-++
                                                } else { // if (Exact_flag) {

            //memset (wrdCACHEDT1, 0, MaxLineLength+1+1);
            i = 1;
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            wrdlen = 0;
        for( k = WorkAreaLedgeT1; k <= WorkAreaRedgeT1; k++ )
            {
```

```
                        workbyte = xgamsCACHE[k];

                         if( wrdlen < MAXboth) {
                                                  if (CaseSensitiveWildcardMatching_flag == 0)
                                    wrdLOW[ wrdlen ] = KAZE_tolower( workbyte );
                                                  else
                                    wrdLOW[ wrdlen ] = ( workbyte );
                                              wrd[ wrdlen ] = workbyte;
                                        }
                         if (workbyte == 10) {
                                              TotalLines1++;
// Wildcard search [
                         if ( 0 < wrdlen && wrdlen < MAXboth)
                                        {
                                              wrd[ wrdlen ] = 0;
                                              wrdLOW[ wrdlen ] = 0;
                                              if ( wrd[ wrdlen-1 ] == 13 ) //CR
                                                        {--wrdlen; wrd[ wrdlen ] = 0; wrdLOW[ wrdlen ] = 0;}

if (WILDCARD_IP_flag) {
// WILDCARD IP [
                                              WordsChecked1++;
                                              if (Exact_flag) {
                                                        //if ((long)( Railgun_Quadruplet_7Gulliver_1(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                        #ifdef RG7Gulliver
                                                                  //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_1(wrdLOW, wrdARG, wrdlen, n);
                                                                  FOUNDinPTR = Railgun_Sekireigan_Wolfram_1(wrdLOW, wrdARG, wrdlen, n);
                                                        #else
                                                                  FOUNDinPTR = Railgun_Quadruplet_7_1(wrdLOW, wrdARG, wrdlen, n);
                                                        #endif
                                                        // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                        // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                        //if ((long)( FOUNDinPTR - wrdLOW )>=0)
                                                        if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(wrdLOW))
                                                        //if ((long)( Railgun_Quadruplet_7_1(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                                  {
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                  DumpedLines1++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                                  }
                                              } else {
                                                        if (WILDCARD_FAST_flag) {
#if defined(_WildFastKaze_)
                                                        if ( WildcardMatch_Iterative_Kaze1(wrdARG, wrdLOW) ) {
#else
                                                        if ( IterativeWildcards1(wrdARG, wrdLOW) ) {
#endif
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                  DumpedLines1++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                        }
                                                        } else {
                                                        maskGLOBALlen = n;
                                                        nameGLOBALlen1 = wrdlen;
                                                        if ( EnhancedMaskTest_OrEmpty_AndNotEmpty_1(wrdARG, 0, wrdLOW, 0) ) { // Caution: Not lowercased as it should!
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                  DumpedLines1++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
```

```
                                                        }
                                                        }
                                                        }
                                                }
// WILDCARD IP ]
} else {

// A simple heuristic #1: Don't enter the nasty loops unless MaximumLevenshteinDistance >= ABS(m-n).
m = wrdlen; // strlen(wrd);
//if (m>MaxLineLength)
//{ printf( "\nKazahana: Incoming xgram exceeding the limit.\n" ); exit( 2 ); }
// Above two commented lines are too severe, changed with next line allowing to search into lines bigger than our needs:
if (m)
if (EXHAUSTIVE_flag == 1) {

// Here we'll walk through the whole length of 'm', ay-yaa.
// EXHAUSTIVE [[[[[[[[[[[[[[[[[[[[[[[[[[[[[
// Here the old fuzzy is replaced with exhaustive one, using BBs of the incoming string (here 'm').
// We need to factorize 'm' down to all 'n+AtMostLevenshteinDistance' long strings/BBs and to search into them ONE-BY-ONE - a gruelling task indeed!
//// Example:
// One of those 33/5 lines is a 46 bytes long string, i.e. m = 46:
// |Edelweiß || [[edelweiss]] || edelweiss flower
// Let's search fuzzily for "edelvais", pretending we don't know the right spelling:
// To find "edelweiss" we need at least Levenshtein distance 3, i.e. n = 8, AtMostLevenshteinDistance = 3:
// 'edelvais' vs 'edelweiss':
// - the mispelled one has one character less;
// - the mispelled one has 2 wrong characters: 'v' & 'a' instead of 'w' & 'e'.
// From above we need Building-Blocks of 46 bytes order 8+3.
// Inhere we are using order 11, 'm - Order + 1' is the number of total BBs for text 'm' bytes long: 46-11+1 = 36
// The 36 BBs:
// |Edelweiß |
// Edelweiß ||
// ...
// weiss flowe
// eiss flower
//
// Not so trivial, below, when the pattern is one more or one less char the dump fails to match the proper LD:
// Kazahana 2e "Meescha Tate" kazahana3.txt 1536
// Kazahana: Total/Checked/Dumped xgrams: 3/93/0
//
// Kazahana 3e "Meescha Tate" kazahana3.txt 1536
// Kazahana: Total/Checked/Dumped xgrams: 3/34/3
// type Kazahana.txt
// *  1977   &amp;ndash; [[Ameesha Patel]], Indian actress and producer
// * [[Ameesha Patel]]
// * [[Miesha Tate]], Mixed martial artist
//
// Kazahana 2e "Misha Tate" kazahana3.txt 1536
// Kazahana: Total/Checked/Dumped xgrams: 3/74/1
// type Kazahana.txt
// * [[Miesha Tate]], Mixed martial artist
//
// Kazahana 1e "Misha Tate" kazahana3.txt 1536
// Kazahana: Total/Checked/Dumped xgrams: 3/99/0
//
// Kazahana 1e "MieshaA Tate" kazahana3.txt 1536
// Kazahana: Total/Checked/Dumped xgrams: 3/93/0
// Kazahana 2e "MieshaAA Tate" kazahana3.txt 1536
// Kazahana: Total/Checked/Dumped xgrams: 3/90/0

// Slow! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
/*
                                for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                for (l=0; l < m-BB+1; l++) {
            WordsChecked1++;
// LD [
            SkipHeuristic=0;
            for(i=1;i<=BB;i++) {
                        for (j=1;j<=n;j++) {
```

```
                                if(wrdLOW[l+i-1] == wrdARG[j-1])
                                            LevenshteinT1[i][j] = LevenshteinT1[i-1][j-1];
                                else
#if defined(_WIN32ASM_)
                                            LevenshteinT1[i][j] = min_AF(LevenshteinT1[i-1][j]+1, LevenshteinT1[i][j-1]+1, LevenshteinT1[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                            LevenshteinT1[i][j] = MIN(MIN((LevenshteinT1[i-1][j]+1),(LevenshteinT1[i][j-1]+1)),(LevenshteinT1[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                            //{LevenshteinT1[i][j] = MIN(MIN(LevenshteinT1[i-1][j],LevenshteinT1[i][j-1]),LevenshteinT1[i-1][j-1]); --LevenshteinT1[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. BB) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinT1[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT1[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                            if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT1[BB][n]) {
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                        fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines1++;
                                                        }
                                                        // Once dumping the line we need double 'break' from BB and l 'for's:
                                                        goto EXHAUSTIVE1;
                                                        //BB=n+AtMostLevenshteinDistance+1; // Instead of 'goto' setting break for BB 'for', stupidesque
                                                        //break; // No need of further checking down the line, one dump only is needed.
                                                        //if ((DumpedLines & 0xff) == 0xff)
                                                        //         //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                        //         fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                            }
// LD ]
                                }
                                }
EXHAUSTIVE1:;
*/
// Slow! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Fast! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

                                for (l=0; l < m-(n-AtMostLevenshteinDistance)+1; l++) { // Here BB = n-AtMostLevenshteinDistance the smallest BB
                                for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                // From here on 'm' will be replaced by BB
                                if (l < m-BB+1) {

if (AtMostLevenshteinDistance >= MAX(BB,n)-MIN(BB,n)) // This is the only add-on for r.1+
{
            WordsChecked1++;
// LD [
StartingPosition = 1;
while (     wrdCACHEDT1[StartingPosition-1] && wrdLOW[l+StartingPosition-1] && wrdCACHEDT1[StartingPosition-1]==wrdLOW[l+StartingPosition-1] )
            StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

            SkipHeuristic=0;
            for(i=StartingPosition;i<=BB;i++) { // StartingPosition is in range 1..
                        for (j=1;j<=n;j++) {
                                if(wrdLOW[l+i-1] == wrdARG[j-1])
                                            LevenshteinT1[i][j] = LevenshteinT1[i-1][j-1];
                                else
#if defined(_WIN32ASM_)
                                            LevenshteinT1[i][j] = min_AF(LevenshteinT1[i-1][j]+1, LevenshteinT1[i][j-1]+1, LevenshteinT1[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                            LevenshteinT1[i][j] = MIN(MIN((LevenshteinT1[i-1][j]+1),(LevenshteinT1[i][j-1]+1)),(LevenshteinT1[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                            //{LevenshteinT1[i][j] = MIN(MIN(LevenshteinT1[i-1][j],LevenshteinT1[i][j-1]),LevenshteinT1[i-1][j-1]); --LevenshteinT1[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }
```

```
		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinT1[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT1[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT1[BB][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines1++;
					}
					// Once dumping the line we need double 'break' from BB and l 'for's:
					goto EXHAUSTIVE1;
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
					//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
				}
	// The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
	//memcpy(wrdCACHEDT1, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
	memcpy(wrdCACHEDT1, &wrdLOW[l], BB); wrdCACHEDT1[BB]=0;
// LD ]
}

			} // if (l < m-BB+1)
			}
			}
EXHAUSTIVE1:;

// Fast! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// EXHAUSTIVE ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]

} else {
if (m<=MaxLineLength)
if (AtMostLevenshteinDistance >= MAX(m,n)-MIN(m,n)) // This is the only add-on for r.1+
{
	WordsChecked1++;
// LD [
StartingPosition = 1;
while (	wrdCACHEDT1[StartingPosition-1] && wrdCACHEDT1[StartingPosition-1]==wrdLOW[StartingPosition-1] ) // No need of && wrd[StartingPosition-1]
	StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

	SkipHeuristic=0;
	for(i=StartingPosition;i<=m;i++) { // StartingPosition is in range 1..
		for (j=1;j<=n;j++) {
			if(wrdLOW[i-1] == wrdARG[j-1])
				LevenshteinT1[i][j] = LevenshteinT1[i-1][j-1];
			else
#if defined(_WIN32ASM_)
				LevenshteinT1[i][j] = min_AF(LevenshteinT1[i-1][j]+1, LevenshteinT1[i][j-1]+1, LevenshteinT1[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinT1[i][j] = MIN(MIN((LevenshteinT1[i-1][j]+1),(LevenshteinT1[i][j-1]+1)),(LevenshteinT1[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinT1[i][j] = MIN(MIN(LevenshteinT1[i-1][j],LevenshteinT1[i][j-1]),LevenshteinT1[i-1][j-1]); --LevenshteinT1[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
		}

		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinT1[i][n] - (m-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT1[i][n] - (m-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT1[m][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
```

```
                                                        fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines1++;
                                                        }
                                                        //if ((DumpedLines & 0xff) == 0xff)
                                                        //          //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                        //          fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                             }
           // The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
           memcpy(wrdCACHEDT1, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
// LD ]
}
} //if (EXHAUSTIVE_flag == 1)

} //if (WILDCARD_IP_flag)
                                  }
// Wildcard search ]
                                             wrdlen = 0;
                        }
                        else wrdlen++;
        } // k 'for'
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                             } // if (Exact_flag) {
           }

// 2nd thread
#ifdef Commence_OpenMP
    #pragma omp section
#endif
           {

                                             if (Exact_flag) {
// WHOLE buffer at once not line-by-line [[[[[[[[ Since r.1-++
                                                        k = WorkAreaLedgeT2;
                                                        while ( k < WorkAreaRedgeT2 ) {
                                                        #ifdef RG7Gulliver
                                                                   //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_2(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT2-k+1, n);
                                                                   FOUNDinPTR = Railgun_Sekireigan_Wolfram_2(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT2-k+1, n);
                                                        #else
                                                                   FOUNDinPTR = Railgun_Quadruplet_7_2(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT2-k+1, n);
                                                        #endif
                                                        // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                        // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                        //if ((long)( FOUNDinPTR - &xgamsCACHE[k] )>=0) {
                                                        if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(&xgamsCACHE[k])) {
                                                                   i = k + (long)( FOUNDinPTR - &xgamsCACHE[k] ); j = i;
                                                                   while (xgamsCACHE[j] != 10) {++j;} // Works both on UNIX(LF) and Windows(CRLF)
                                                                   while (i > k && xgamsCACHE[i-1] != 10) {--i;}
                                                                   k = j+1; // Should "point" to first symbol after the dumped fragment.

                                                                   //fwrite( &xgamsCACHE[i], j - i + 1, 1, fp_outLINE ); DumpedLines1++;
                                                                   if (j - i + 1 <= 168*MaxLineLength) { // fix for 1-++
                                                                   memcpy(wrd, &xgamsCACHE[i], j - i + 1); wrd[j - i + 1]=0; DumpedLines2++;
                                                                   j--; if (wrd[j - i + 1]==10) wrd[j - i + 1]=0;
                                                                   j--; if (wrd[j - i + 1]==13) wrd[j - i + 1]=0;
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                   if (Dump_flag)
                                                                   fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]);
                                                                   else
                                                                   fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                                   }
                                                        } else k = WorkAreaRedgeT2;
                                                        } // while
// WHOLE buffer at once not line-by-line ]]]]]]]] Since r.1-++
                                             } else { // if (Exact_flag) {
```

```
            //memset (wrdCACHEDT2, 0, MaxLineLength+1+1);
            i = 1;
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            wrdlen = 0;
        for( k = WorkAreaLedgeT2; k <= WorkAreaRedgeT2; k++ )
            {
                       workbyte = xgamsCACHE[k];

                         if( wrdlen < MAXboth) {
                                                          if (CaseSensitiveWildcardMatching_flag == 0)
                                    wrdLOW[ wrdlen ] = KAZE_tolower( workbyte );
                                                          else
                                    wrdLOW[ wrdlen ] = ( workbyte );
                                              wrd[ wrdlen ] = workbyte;
                                   }
                         if (workbyte == 10) {
                                              TotalLines2++;
// Wildcard search [
                         if ( 0 < wrdlen && wrdlen < MAXboth)
                                   {
                                              wrd[ wrdlen ] = 0;
                                              wrdLOW[ wrdlen ] = 0;
                                              if ( wrd[ wrdlen-1 ] == 13 ) //CR
                                                          {--wrdlen; wrd[ wrdlen ] = 0; wrdLOW[ wrdlen ] = 0;}

if (WILDCARD_IP_flag) {
// WILDCARD IP [
                                              WordsChecked2++;
                                              if (Exact_flag) {
                                                          //if ((long)( Railgun_Quadruplet_7Gulliver_2(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                          #ifdef RG7Gulliver
                                                                      //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_2(wrdLOW, wrdARG, wrdlen, n);
                                                                      FOUNDinPTR = Railgun_Sekireigan_Wolfram_2(wrdLOW, wrdARG, wrdlen, n);
                                                          #else
                                                                      FOUNDinPTR = Railgun_Quadruplet_7_2(wrdLOW, wrdARG, wrdlen, n);
                                                          #endif
                                                          // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                          // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                          //if ((long)( FOUNDinPTR - wrdLOW )>=0)
                                                          if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(wrdLOW))
                                                          //if ((long)( Railgun_Quadruplet_7_2(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                                      {
                                                          // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                          #pragma omp critical

#endif
                                                          {
                                                                      DumpedLines2++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                          }
                                                                      }
                                              } else {
                                                          if (WILDCARD_FAST_flag) {
#if defined(_WildFastKaze_)
                                                          if ( WildcardMatch_Iterative_Kaze2(wrdARG, wrdLOW) ) {

#else
                                                          if ( IterativeWildcards2(wrdARG, wrdLOW) ) {

#endif
                                                          // Below pragma is needed explicitly only for MinGW, grrr...

#ifdef Commence_OpenMP
                                                          #pragma omp critical

#endif
                                                          {
                                                                      DumpedLines2++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                          }
                                                          }
                                                          } else {
                                                          maskGLOBALlen = n;
                                                          nameGLOBALlen2 = wrdlen;
                                                          if ( EnhancedMaskTest_OrEmpty_AndNotEmpty_2(wrdARG, 0, wrdLOW, 0) ) { // Caution: Not lowercased as it should!
```

```
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                DumpedLines2++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                        }
                                                        }
                                                }
// WILDCARD IP ]
} else {

// A simple heuristic #1: Don't enter the nasty loops unless MaximumLevenshteinDistance >= ABS(m-n).
m = wrdlen; // strlen(wrd);
//if (m>MaxLineLength)
//{ printf( "\nKazahana: Incoming xgram exceeding the limit.\n" ); exit( 2 ); }
// Above two commented lines are too severe, changed with next line allowing to search into lines bigger than our needs:
if (m)
if (EXHAUSTIVE_flag == 1) {

// Here we'll walk through the whole length of 'm', ay-yaa.
// EXHAUSTIVE [[[[[[[[[[[[[[[[[[[[[[[[[[[[
// Here the old fuzzy is replaced with exhaustive one, using BBs of the incoming string (here 'm').
// We need to factorize 'm' down to all 'n+AtMostLevenshteinDistance' long strings/BBs and to search into them ONE-BY-ONE - a gruelling task indeed!
//// Example:
// One of those 33/5 lines is a 46 bytes long string, i.e. m = 46:
// |Edelweiß || [[edelweiss]] || edelweiss flower
// Let's search fuzzily for "edelvais", pretending we don't know the right spelling:
// To find "edelweiss" we need at least Levenshtein distance 3, i.e. n = 8, AtMostLevenshteinDistance = 3:
// 'edelvais' vs 'edelweiss':
// - the mispelled one has one character less;
// - the mispelled one has 2 wrong characters: 'v' & 'a' instead of 'w' & 'e'.
// From above we need Building-Blocks of 46 bytes order 8+3.
// Inhere we are using order 11, 'm - Order + 1' is the number of total BBs for text 'm' bytes long: 46-11+1 = 36
// The 36 BBs:
// |Edelweiß |
// Edelweiß ||
// ...
// weiss flowe
// eiss flower

// Slow! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
/*
                                for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                for (l=0; l < m-BB+1; l++) {
            WordsChecked2++;
// LD [
            SkipHeuristic=0;
            for(i=1;i<=BB;i++) {
                        for (j=1;j<=n;j++) {
                                    if(wrdLOW[l+i-1] == wrdARG[j-1])
                                                LevenshteinT2[i][j] = LevenshteinT2[i-1][j-1];
                                    else
#if defined(_WIN32ASM_)
                                                LevenshteinT2[i][j] = min_AF(LevenshteinT2[i-1][j]+1, LevenshteinT2[i][j-1]+1, LevenshteinT2[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                                LevenshteinT2[i][j] = MIN(MIN((LevenshteinT2[i-1][j]+1),(LevenshteinT2[i][j-1]+1)),(LevenshteinT2[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                                //{LevenshteinT2[i][j] = MIN(MIN(LevenshteinT2[i-1][j],LevenshteinT2[i][j-1]),LevenshteinT2[i-1][j-1]); --LevenshteinT2[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. BB) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinT2[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT2[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                                if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT2[BB][n]) {
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
```

```
                                                        #pragma omp critical
#endif
                                                        {
                                                        fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines2++;
                                                        }
                                                        goto EXHAUSTIVE2;
                                                        //break; // No need of further checking down the line, one dump only is needed.
                                                        //if ((DumpedLines & 0xff) == 0xff)
                                                        //        //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                        //        fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                }
// LD ]
                                        }
                                        }
EXHAUSTIVE2:;
*/
// Slow! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Fast! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

                                        for (l=0; l < m-(n-AtMostLevenshteinDistance)+1; l++) { // Here BB = n-AtMostLevenshteinDistance the smallest BB
                                        for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                        // From here on 'm' will be replaced by BB
                                        if (l < m-BB+1) {

if (AtMostLevenshteinDistance >= MAX(BB,n)-MIN(BB,n)) // This is the only add-on for r.1+
{
            WordsChecked2++;
// LD [
StartingPosition = 1;
while (     wrdCACHEDT2[StartingPosition-1] && wrdLOW[l+StartingPosition-1] && wrdCACHEDT2[StartingPosition-1]==wrdLOW[l+StartingPosition-1] )
            StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

            SkipHeuristic=0;
            for(i=StartingPosition;i<=BB;i++) { // StartingPosition is in range 1..
                        for (j=1;j<=n;j++) {
                                    if(wrdLOW[l+i-1] == wrdARG[j-1])
                                                LevenshteinT2[i][j] = LevenshteinT2[i-1][j-1];
                                    else
#if defined(_WIN32ASM_)
                                                LevenshteinT2[i][j] = min_AF(LevenshteinT2[i-1][j]+1, LevenshteinT2[i][j-1]+1, LevenshteinT2[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                                LevenshteinT2[i][j] = MIN(MIN((LevenshteinT2[i-1][j]+1),(LevenshteinT2[i][j-1]+1)),(LevenshteinT2[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                                //{LevenshteinT2[i][j] = MIN(MIN(LevenshteinT2[i-1][j],LevenshteinT2[i][j-1]),LevenshteinT2[i-1][j-1]); --LevenshteinT2[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinT2[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT2[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                                if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT2[BB][n]) {
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                        fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines2++;
                                                        }
                                                        // Once dumping the line we need double 'break' from BB and l 'for's:
                                                        goto EXHAUSTIVE2;
                                                        //if ((DumpedLines & 0xff) == 0xff)
                                                        //        //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                        //        fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                }
            // The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
            //memcpy(wrdCACHEDT1, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
```

```
	memcpy(wrdCACHEDT2, &wrdLOW[l], BB); wrdCACHEDT2[BB]=0;
// LD ]
}

			} // if (l < m-BB+1)
			}
			}
EXHAUSTIVE2:;

// Fast! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// EXHAUSTIVE ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]

} else {
if (m<=MaxLineLength)
if (AtMostLevenshteinDistance >= MAX(m,n)-MIN(m,n)) // This is the only add-on for r.1+
{
	WordsChecked2++;
// LD [
StartingPosition = 1;
while (	wrdCACHEDT2[StartingPosition-1] && wrdCACHEDT2[StartingPosition-1]==wrdLOW[StartingPosition-1] ) // No need of && wrd[StartingPosition-1]
	StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

	SkipHeuristic=0;
	for(i=StartingPosition;i<=m;i++) { // StartingPosition is in range 1..
		for (j=1;j<=n;j++) {
			if(wrdLOW[i-1] == wrdARG[j-1])
				LevenshteinT2[i][j] = LevenshteinT2[i-1][j-1];
			else
#if defined(_WIN32ASM_)
				LevenshteinT2[i][j] = min_AF(LevenshteinT2[i-1][j]+1, LevenshteinT2[i][j-1]+1, LevenshteinT2[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinT2[i][j] = MIN(MIN((LevenshteinT2[i-1][j]+1),(LevenshteinT2[i][j-1]+1)),(LevenshteinT2[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinT2[i][j] = MIN(MIN(LevenshteinT2[i-1][j],LevenshteinT2[i][j-1]),LevenshteinT2[i-1][j-1]); --LevenshteinT2[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! // 237,270 xgrams/s
#endif
		}

		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinT2[i][n] - (m-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT2[i][n] - (m-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT2[m][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines2++;
					}
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
					//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
				}
	// The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
	memcpy(wrdCACHEDT2, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
// LD ]
}
} //if (EXHAUSTIVE_flag == 1)

} //if (WILDCARD_IP_flag)
			}
// Wildcard search ]
				wrdlen = 0;
		 }
		 else wrdlen++;
        } // k 'for'
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
                                             } // if (Exact_flag) {
           }

// 3rd thread
#ifdef Commence_OpenMP
    #pragma omp section
#endif
           {

                                             if (Exact_flag) {
// WHOLE buffer at once not line-by-line [[[[[[[ Since r.1-++
                                                        k = WorkAreaLedgeT3;
                                                        while ( k < WorkAreaRedgeT3 ) {
                                                        #ifdef RG7Gulliver
                                                                   //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_3(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT3-k+1, n);
                                                                   FOUNDinPTR = Railgun_Sekireigan_Wolfram_3(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT3-k+1, n);
                                                        #else
                                                                   FOUNDinPTR = Railgun_Quadruplet_7_3(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT3-k+1, n);
                                                        #endif
                                                        // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                        // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                        //if ((long)( FOUNDinPTR - &xgamsCACHE[k] )>=0) {
                                                        if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(&xgamsCACHE[k])) {
                                                                   i = k + (long)( FOUNDinPTR - &xgamsCACHE[k] ); j = i;
                                                                   while (xgamsCACHE[j] != 10) {++j;} // Works both on UNIX(LF) and Windows(CRLF)
                                                                   while (i > k && xgamsCACHE[i-1] != 10) {--i;}
                                                                   k = j+1; // Should "point" to first symbol after the dumped fragment.

                                                                   //fwrite( &xgamsCACHE[i], j - i + 1, 1, fp_outLINE ); DumpedLines1++;
                                                                   if (j - i + 1 <= 168*MaxLineLength) { // fix for 1-++
                                                                   memcpy(wrd, &xgamsCACHE[i], j - i + 1); wrd[j - i + 1]=0; DumpedLines3++;
                                                                   j--; if (wrd[j - i + 1]==10) wrd[j - i + 1]=0;
                                                                   j--; if (wrd[j - i + 1]==13) wrd[j - i + 1]=0;
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP

                                                        #pragma omp critical

#endif

                                                        {
                                                                   if (Dump_flag)
                                                                   fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]);
                                                                   else
                                                                   fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                                   }
                                                        } else k = WorkAreaRedgeT3;
                                                        } // while
// WHOLE buffer at once not line-by-line ]]]]]]]] Since r.1-++
                                             } else { // if (Exact_flag) {

           //memset (wrdCACHEDT3, 0, MaxLineLength+1+1);
           i = 1;
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
           wrdlen = 0;
        for( k = WorkAreaLedgeT3; k <= WorkAreaRedgeT3; k++ )
           {
                      workbyte = xgamsCACHE[k];

                        if( wrdlen < MAXboth) {
                                                        if (CaseSensitiveWildcardMatching_flag == 0)
                                   wrdLOW[ wrdlen ] = KAZE_tolower( workbyte );
                                                        else
                                   wrdLOW[ wrdlen ] = ( workbyte );
                                             wrd[ wrdlen ] = workbyte;
                                 }
                        if (workbyte == 10) {
                                             TotalLines3++;
// Wildcard search [
                        if ( 0 < wrdlen && wrdlen < MAXboth)
                                 {
```

```
                                        wrd[ wrdlen ] = 0;
                                        wrdLOW[ wrdlen ] = 0;
                                        if ( wrd[ wrdlen-1 ] == 13 ) //CR
                                                {--wrdlen; wrd[ wrdlen ] = 0; wrdLOW[ wrdlen ] = 0;}

if (WILDCARD_IP_flag) {
// WILDCARD IP [
                                        WordsChecked3++;
                                        if (Exact_flag) {
                                                //if ((long)( Railgun_Quadruplet_7Gulliver_3(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                #ifdef RG7Gulliver
                                                        //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_3(wrdLOW, wrdARG, wrdlen, n);
                                                        FOUNDinPTR = Railgun_Sekireigan_Wolfram_3(wrdLOW, wrdARG, wrdlen, n);
                                                #else
                                                        FOUNDinPTR = Railgun_Quadruplet_7_3(wrdLOW, wrdARG, wrdlen, n);
                                                #endif
                                                // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                //if ((long)( FOUNDinPTR - wrdLOW )>=0)
                                                if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(wrdLOW))
                                                //if ((long)( Railgun_Quadruplet_7_3(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                        {
                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                #pragma omp critical
#endif
                                                {
                                                        DumpedLines3++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                }
                                                        }
                                        } else {
                                                if (WILDCARD_FAST_flag) {
#if defined(_WildFastKaze_)
                                                if ( WildcardMatch_Iterative_Kaze3(wrdARG, wrdLOW) ) {
#else
                                                if ( IterativeWildcards3(wrdARG, wrdLOW) ) {
#endif
                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                #pragma omp critical
#endif
                                                {
                                                        DumpedLines3++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                }
                                                }
                                                } else {
                                                maskGLOBALlen = n;
                                                nameGLOBALlen3 = wrdlen;
                                                if ( EnhancedMaskTest_OrEmpty_AndNotEmpty_3(wrdARG, 0, wrdLOW, 0) ) { // Caution: Not lowercased as it should!
                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                #pragma omp critical
#endif
                                                {
                                                        DumpedLines3++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                }
                                                }
                                                }
                                        }
// WILDCARD IP ]
} else {

// A simple heuristic #1: Don't enter the nasty loops unless MaximumLevenshteinDistance >= ABS(m-n).
m = wrdlen; // strlen(wrd);
//if (m>MaxLineLength)
//{ printf( "\nKazahana: Incoming xgram exceeding the limit.\n" ); exit( 2 ); }
// Above two commented lines are too severe, changed with next line allowing to search into lines bigger than our needs:
if (m)
if (EXHAUSTIVE_flag == 1) {
```

```
// Here we'll walk through the whole length of 'm', ay-yaa.
// EXHAUSTIVE [[[[[[[[[[[[[[[[[[[[[[[[[[[[[[
// Here the old fuzzy is replaced with exhaustive one, using BBs of the incoming string (here 'm').
// We need to factorize 'm' down to all 'n+AtMostLevenshteinDistance' long strings/BBs and to search into them ONE-BY-ONE - a gruelling task indeed!
//// Example:
// One of those 33/5 lines is a 46 bytes long string, i.e. m = 46:
// |Edelweiß || [[edelweiss]] || edelweiss flower
// Let's search fuzzily for "edelvais", pretending we don't know the right spelling:
// To find "edelweiss" we need at least Levenshtein distance 3, i.e. n = 8, AtMostLevenshteinDistance = 3:
// 'edelvais' vs 'edelweiss':
// - the mispelled one has one character less;
// - the mispelled one has 2 wrong characters: 'v' & 'a' instead of 'w' & 'e'.
// From above we need Building-Blocks of 46 bytes order 8+3.
// Inhere we are using order 11, 'm - Order + 1' is the number of total BBs for text 'm' bytes long: 46-11+1 = 36
// The 36 BBs:
// |Edelweiß |
// Edelweiß ||
// ...
// weiss flowe
// eiss flower

// Slow! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
/*
                                for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                for (l=0; l < m-BB+1; l++) {
            WordsChecked3++;
// LD [
            SkipHeuristic=0;
            for(i=1;i<=BB;i++) {
                        for (j=1;j<=n;j++) {
                                if(wrdLOW[l+i-1] == wrdARG[j-1])
                                            LevenshteinT3[i][j] = LevenshteinT3[i-1][j-1];
                                else
#if defined(_WIN32ASM_)
                                            LevenshteinT3[i][j] = min_AF(LevenshteinT3[i-1][j]+1, LevenshteinT3[i][j-1]+1, LevenshteinT3[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                            LevenshteinT3[i][j] = MIN(MIN((LevenshteinT3[i-1][j]+1),(LevenshteinT3[i][j-1]+1)),(LevenshteinT3[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                            //{LevenshteinT3[i][j] = MIN(MIN(LevenshteinT3[i-1][j],LevenshteinT3[i][j-1]),LevenshteinT3[i-1][j-1]); --LevenshteinT3[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. BB) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinT3[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT3[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                            if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT3[BB][n]) {
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                        fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines3++;
                                                        }
                                                        goto EXHAUSTIVE3;
                                                        //break; // No need of further checking down the line, one dump only is needed.
                                                        //if ((DumpedLines & 0xff) == 0xff)
                                                        //         //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                        //         fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                            }
// LD ]
                                }
                                }
EXHAUSTIVE3:;
*/
// Slow! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Fast! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

```
			for (l=0; l < m-(n-AtMostLevenshteinDistance)+1; l++) { // Here BB = n-AtMostLevenshteinDistance the smallest BB
			for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
			// From here on 'm' will be replaced by BB
			if (l < m-BB+1) {

if (AtMostLevenshteinDistance >= MAX(BB,n)-MIN(BB,n)) // This is the only add-on for r.1+
{
	WordsChecked3++;
// LD [
StartingPosition = 1;
while (	wrdCACHEDT3[StartingPosition-1] && wrdLOW[l+StartingPosition-1] && wrdCACHEDT3[StartingPosition-1]==wrdLOW[l+StartingPosition-1] )
	StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

	SkipHeuristic=0;
	for(i=StartingPosition;i<=BB;i++) { // StartingPosition is in range 1..
		for (j=1;j<=n;j++) {
			if(wrdLOW[l+i-1] == wrdARG[j-1])
				LevenshteinT3[i][j] = LevenshteinT3[i-1][j-1];
			else
#if defined(_WIN32ASM_)
				LevenshteinT3[i][j] = min_AF(LevenshteinT3[i-1][j]+1, LevenshteinT3[i][j-1]+1, LevenshteinT3[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinT3[i][j] = MIN(MIN((LevenshteinT3[i-1][j]+1),(LevenshteinT3[i][j-1]+1)),(LevenshteinT3[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinT3[i][j] = MIN(MIN(LevenshteinT3[i-1][j],LevenshteinT3[i][j-1]),LevenshteinT3[i-1][j-1]); --LevenshteinT3[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
		}

		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinT3[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT3[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT3[BB][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines3++;
					}
					// Once dumping the line we need double 'break' from BB and l 'for's:
					goto EXHAUSTIVE3;
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
					//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
				}
	// The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
	//memcpy(wrdCACHEDT3, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
	memcpy(wrdCACHEDT3, &wrdLOW[l], BB); wrdCACHEDT3[BB]=0;
// LD ]
}

			} // if (l < m-BB+1)
			}
			}
EXHAUSTIVE3:;

// Fast! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// EXHAUSTIVE ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]

} else {
if (m<=MaxLineLength)
if (AtMostLevenshteinDistance >= MAX(m,n)-MIN(m,n)) // This is the only add-on for r.1+
{
	WordsChecked3++;
// LD [
StartingPosition = 1;
```

```
while (    wrdCACHEDT3[StartingPosition-1] && wrdCACHEDT3[StartingPosition-1]==wrdLOW[StartingPosition-1] ) // No need of && wrd[StartingPosition-1]
           StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

           SkipHeuristic=0;
           for(i=StartingPosition;i<=m;i++) { // StartingPosition is in range 1..
                      for (j=1;j<=n;j++) {
                                 if(wrdLOW[i-1] == wrdARG[j-1])
                                            LevenshteinT3[i][j] = LevenshteinT3[i-1][j-1];
                                 else
#if defined(_WIN32ASM_)
                                            LevenshteinT3[i][j] = min_AF(LevenshteinT3[i-1][j]+1, LevenshteinT3[i][j-1]+1, LevenshteinT3[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                            LevenshteinT3[i][j] = MIN(MIN((LevenshteinT3[i-1][j]+1),(LevenshteinT3[i][j-1]+1)),(LevenshteinT3[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                            //{LevenshteinT3[i][j] = MIN(MIN(LevenshteinT3[i-1][j],LevenshteinT3[i][j-1]),LevenshteinT3[i-1][j-1]); --LevenshteinT3[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                      }

                      // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                      if ( LevenshteinT3[i][n] - (m-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT3[i][n] - (m-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
           }
                                            if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT3[m][n]) {
                                                       // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                       #pragma omp critical
#endif
                                                       {
                                                       fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines3++;
                                                       }
                                                       //if ((DumpedLines & 0xff) == 0xff)
                                                       //         //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                       //         fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                            }
           // The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
           memcpy(wrdCACHEDT3, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
// LD ]
}
} //if (EXHAUSTIVE_flag == 1)

} //if (WILDCARD_IP_flag)
                                 }
// Wildcard search ]
                                            wrdlen = 0;
                       }
                       else wrdlen++;
        } // k 'for'
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                            } // if (Exact_flag) {
           }

// 4th thread
#ifdef Commence_OpenMP
    #pragma omp section
#endif
           {

                                            if (Exact_flag) {
// WHOLE buffer at once not line-by-line [[[[[[[[ Since r.1-++
                                                       k = WorkAreaLedgeT4;
                                                       while ( k < WorkAreaRedgeT4 ) {
                                                       #ifdef RG7Gulliver
                                                                  //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_4(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT4-k+1, n);
                                                                  FOUNDinPTR = Railgun_Sekireigan_Wolfram_4(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT4-k+1, n);
                                                       #else
                                                                  FOUNDinPTR = Railgun_Quadruplet_7_4(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT4-k+1, n);
                                                       #endif
                                                       // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
```

```
                                                        // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                        //if ((long)( FOUNDinPTR - &xgamsCACHE[k] )>=0) {
                                                        if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(&xgamsCACHE[k])) {
                                                                   i = k + (long)( FOUNDinPTR - &xgamsCACHE[k] ); j = i;
                                                                   while (xgamsCACHE[j] != 10) {++j;} // Works both on UNIX(LF) and Windows(CRLF)
                                                                   while (i > k && xgamsCACHE[i-1] != 10) {--i;}
                                                                   k = j+1; // Should "point" to first symbol after the dumped fragment.

                                                                   //fwrite( &xgamsCACHE[i], j - i + 1, 1, fp_outLINE ); DumpedLines1++;
                                                                   if (j - i + 1 <= 168*MaxLineLength) { // fix for 1-++
                                                                   memcpy(wrd, &xgamsCACHE[i], j - i + 1); wrd[j - i + 1]=0; DumpedLines4++;
                                                                   j--; if (wrd[j - i + 1]==10) wrd[j - i + 1]=0;
                                                                   j--; if (wrd[j - i + 1]==13) wrd[j - i + 1]=0;
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                   if (Dump_flag)
                                                                   fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]);
                                                                   else
                                                                   fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                                   }
                                                        } else k = WorkAreaRedgeT4;
                                                        } // while
// WHOLE buffer at once not line-by-line ]]]]]]]] Since r.1-++
                                             } else { // if (Exact_flag) {

           //memset (wrdCACHEDT4, 0, MaxLineLength+1+1);
           i = 1;
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
           wrdlen = 0;
        for( k = WorkAreaLedgeT4; k <= WorkAreaRedgeT4; k++ )
           {
                       workbyte = xgamsCACHE[k];

                        if( wrdlen < MAXboth) {
                                                        if (CaseSensitiveWildcardMatching_flag == 0)
                                    wrdLOW[ wrdlen ] = KAZE_tolower( workbyte );
                                                        else
                                    wrdLOW[ wrdlen ] = ( workbyte );
                                                 wrd[ wrdlen ] = workbyte;
                                              }
                        if (workbyte == 10) {
                                                 TotalLines4++;
// Wildcard search [
                        if ( 0 < wrdlen && wrdlen < MAXboth)
                                              {
                                                 wrd[ wrdlen ] = 0;
                                                 wrdLOW[ wrdlen ] = 0;
                                                 if ( wrd[ wrdlen-1 ] == 13 ) //CR
                                                        {--wrdlen; wrd[ wrdlen ] = 0; wrdLOW[ wrdlen ] = 0;}

if (WILDCARD_IP_flag) {
// WILDCARD IP [
                                                 WordsChecked4++;
                                                 if (Exact_flag) {
                                                        //if ((long)( Railgun_Quadruplet_7Gulliver_4(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                        #ifdef RG7Gulliver
                                                                   //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_4(wrdLOW, wrdARG, wrdlen, n);
                                                                   FOUNDinPTR = Railgun_Sekireigan_Wolfram_4(wrdLOW, wrdARG, wrdlen, n);
                                                        #else
                                                                   FOUNDinPTR = Railgun_Quadruplet_7_4(wrdLOW, wrdARG, wrdlen, n);
                                                        #endif
                                                        // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                        // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                        //if ((long)( FOUNDinPTR - wrdLOW )>=0)
                                                        if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(wrdLOW))
```

```
                                                //if ((long)( Railgun_Quadruplet_7_4(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                                {
                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                #pragma omp critical
#endif
                                                {
                                                                DumpedLines4++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                }
                                                                }
                                        } else {
                                                if (WILDCARD_FAST_flag) {
#if defined(_WildFastKaze_)
                                                if ( WildcardMatch_Iterative_Kaze4(wrdARG, wrdLOW) ) {
#else
                                                if ( IterativeWildcards4(wrdARG, wrdLOW) ) {
#endif
                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                #pragma omp critical
#endif
                                                {
                                                                DumpedLines4++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                }
                                                }
                                                } else {
                                                maskGLOBALlen = n;
                                                nameGLOBALlen4 = wrdlen;
                                                if ( EnhancedMaskTest_OrEmpty_AndNotEmpty_4(wrdARG, 0, wrdLOW, 0) ) { // Caution: Not lowercased as it should!
                                                                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                #pragma omp critical
#endif
                                                {
                                                                DumpedLines4++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                }
                                                }
                                                }
                                        }
// WILDCARD IP ]
} else {

// A simple heuristic #1: Don't enter the nasty loops unless MaximumLevenshteinDistance >= ABS(m-n).
m = wrdlen; // strlen(wrd);
//if (m>MaxLineLength)
//{ printf( "\nKazahana: Incoming xgram exceeding the limit.\n" ); exit( 2 ); }
// Above two commented lines are too severe, changed with next line allowing to search into lines bigger than our needs:
if (m)
if (EXHAUSTIVE_flag == 1) {

// Here we'll walk through the whole length of 'm', ay-yaa.
// EXHAUSTIVE [[[[[[[[[[[[[[[[[[[[[[[[[[[[[[
// Here the old fuzzy is replaced with exhaustive one, using BBs of the incoming string (here 'm').
// We need to factorize 'm' down to all 'n+AtMostLevenshteinDistance' long strings/BBs and to search into them ONE-BY-ONE - a gruelling task indeed!
//// Example:
// One of those 33/5 lines is a 46 bytes long string, i.e. m = 46:
// |Edelweiß || [[edelweiss]] || edelweiss flower
// Let's search fuzzily for "edelvais", pretending we don't know the right spelling:
// To find "edelweiss" we need at least Levenshtein distance 3, i.e. n = 8, AtMostLevenshteinDistance = 3:
// 'edelvais' vs 'edelweiss':
// - the mispelled one has one character less;
// - the mispelled one has 2 wrong characters: 'v' & 'a' instead of 'w' & 'e'.
// From above we need Building-Blocks of 46 bytes order 8+3.
// Inhere we are using order 11, 'm - Order + 1' is the number of total BBs for text 'm' bytes long: 46-11+1 = 36
// The 36 BBs:
// |Edelweiß |
// Edelweiß ||
// ...
// weiss flowe
```

```
// eiss flower

// Slow! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
/*
			for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
			for (l=0; l < m-BB+1; l++) {
	WordsChecked4++;
// LD [
	SkipHeuristic=0;
	for(i=1;i<=BB;i++) {
		for (j=1;j<=n;j++) {
			if(wrdLOW[l+i-1] == wrdARG[j-1])
				LevenshteinT4[i][j] = LevenshteinT4[i-1][j-1];
			else
#if defined(_WIN32ASM_)
				LevenshteinT4[i][j] = min_AF(LevenshteinT4[i-1][j]+1, LevenshteinT4[i][j-1]+1, LevenshteinT4[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinT4[i][j] = MIN(MIN((LevenshteinT4[i-1][j]+1),(LevenshteinT4[i][j-1]+1)),(LevenshteinT4[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinT4[i][j] = MIN(MIN(LevenshteinT4[i-1][j],LevenshteinT4[i][j-1]),LevenshteinT4[i-1][j-1]); --LevenshteinT4[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
		}

		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. BB) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinT4[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT4[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT4[BB][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines4++;
					}
					goto EXHAUSTIVE4;
					//break; // No need of further checking down the line, one dump only is needed.
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
					//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
				}
// LD ]
			}
			}
EXHAUSTIVE4:;
*/
// Slow! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Fast! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

			for (l=0; l < m-(n-AtMostLevenshteinDistance)+1; l++) { // Here BB = n-AtMostLevenshteinDistance the smallest BB
			for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
			// From here on 'm' will be replaced by BB
			if (l < m-BB+1) {

if (AtMostLevenshteinDistance >= MAX(BB,n)-MIN(BB,n)) // This is the only add-on for r.1+
{
	WordsChecked4++;
// LD [
StartingPosition = 1;
while (	wrdCACHEDT4[StartingPosition-1] && wrdLOW[l+StartingPosition-1] && wrdCACHEDT4[StartingPosition-1]==wrdLOW[l+StartingPosition-1] )
	StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

	SkipHeuristic=0;
	for(i=StartingPosition;i<=BB;i++) { // StartingPosition is in range 1..
		for (j=1;j<=n;j++) {
			if(wrdLOW[l+i-1] == wrdARG[j-1])
				LevenshteinT4[i][j] = LevenshteinT4[i-1][j-1];
```

```
			else
#if defined(_WIN32ASM_)
				LevenshteinT4[i][j] = min_AF(LevenshteinT4[i-1][j]+1, LevenshteinT4[i][j-1]+1, LevenshteinT4[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinT4[i][j] = MIN(MIN((LevenshteinT4[i-1][j]+1),(LevenshteinT4[i][j-1]+1)),(LevenshteinT4[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinT4[i][j] = MIN(MIN(LevenshteinT4[i-1][j],LevenshteinT4[i][j-1]),LevenshteinT4[i-1][j-1]); --LevenshteinT4[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! // 
237,270 xgrams/s
#endif
		}

		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinT4[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT4[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT4[BB][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines4++;
					}
					// Once dumping the line we need double 'break' from BB and l 'for's:
					goto EXHAUSTIVE4;
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
					//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
				}
	// The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
	//memcpy(wrdCACHEDT4, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
	memcpy(wrdCACHEDT4, &wrdLOW[l], BB); wrdCACHEDT4[BB]=0;
// LD ]
}

			} // if (l < m-BB+1)
			}
			}
EXHAUSTIVE4:;

// Fast! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// EXHAUSTIVE ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]

} else {
if (m<=MaxLineLength)
if (AtMostLevenshteinDistance >= MAX(m,n)-MIN(m,n)) // This is the only add-on for r.1+
{
	WordsChecked4++;
// LD [
StartingPosition = 1;
while (	wrdCACHEDT4[StartingPosition-1] && wrdCACHEDT4[StartingPosition-1]==wrdLOW[StartingPosition-1] ) // No need of && wrd[StartingPosition-1]
	StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

	SkipHeuristic=0;
	for(i=StartingPosition;i<=m;i++) { // StartingPosition is in range 1..
		for (j=1;j<=n;j++) {
			if(wrdLOW[i-1] == wrdARG[j-1])
				LevenshteinT4[i][j] = LevenshteinT4[i-1][j-1];
			else
#if defined(_WIN32ASM_)
				LevenshteinT4[i][j] = min_AF(LevenshteinT4[i-1][j]+1, LevenshteinT4[i][j-1]+1, LevenshteinT4[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinT4[i][j] = MIN(MIN((LevenshteinT4[i-1][j]+1),(LevenshteinT4[i][j-1]+1)),(LevenshteinT4[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinT4[i][j] = MIN(MIN(LevenshteinT4[i-1][j],LevenshteinT4[i][j-1]),LevenshteinT4[i-1][j-1]); --LevenshteinT4[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! // 
237,270 xgrams/s
#endif
		}
```

```
                      // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                      if ( LevenshteinT4[i][n] - (m-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT4[i][n] - (m-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
           }
                                             if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT4[m][n]) {
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                        fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines4++;
                                                        }
                                                        //if ((DumpedLines & 0xff) == 0xff)
                                                        //         //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                        //         fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                             }
           // The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
           memcpy(wrdCACHEDT4, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
// LD ]
}
} //if (EXHAUSTIVE_flag == 1)

} //if (WILDCARD_IP_flag)
                                  }
// Wildcard search ]
                                             wrdlen = 0;
                        }
                        else wrdlen++;
        } // k 'for'
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                             } // if (Exact_flag) {
           }

// 5th thread
#ifdef Commence_OpenMP
    #pragma omp section
#endif
           {

                                             if (Exact_flag) {
// WHOLE buffer at once not line-by-line [[[[[[[[ Since r.1-++
                                                        k = WorkAreaLedgeT5;
                                                        while ( k < WorkAreaRedgeT5 ) {
                                                        #ifdef RG7Gulliver
                                                                   //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_5(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT5-k+1, n);
                                                                   FOUNDinPTR = Railgun_Sekireigan_Wolfram_5(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT5-k+1, n);
                                                        #else
                                                                   FOUNDinPTR = Railgun_Quadruplet_7_5(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT5-k+1, n);
                                                        #endif
                                                        // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                        // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                        //if ((long)( FOUNDinPTR - &xgamsCACHE[k] )>=0) {
                                                        if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(&xgamsCACHE[k])) {
                                                                   i = k + (long)( FOUNDinPTR - &xgamsCACHE[k] ); j = i;
                                                                   while (xgamsCACHE[j] != 10) {++j;} // Works both on UNIX(LF) and Windows(CRLF)
                                                                   while (i > k && xgamsCACHE[i-1] != 10) {--i;}
                                                                   k = j+1; // Should "point" to first symbol after the dumped fragment.

                                                                   //fwrite( &xgamsCACHE[i], j - i + 1, 1, fp_outLINE ); DumpedLines1++;
                                                                   if (j - i + 1 <= 168*MaxLineLength) { // fix for 1-++
                                                                   memcpy(wrd, &xgamsCACHE[i], j - i + 1); wrd[j - i + 1]=0; DumpedLines5++;
                                                                   j--; if (wrd[j - i + 1]==10) wrd[j - i + 1]=0;
                                                                   j--; if (wrd[j - i + 1]==13) wrd[j - i + 1]=0;
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                   if (Dump_flag)
                                                                   fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]);
```

```
                                                                 else
                                                                 fprintf( fp_outLINE, "%s\r\n", wrd);
                                                      }
                                                                 }
                                                      } else k = WorkAreaRedgeT5;
                                                      } // while
// WHOLE buffer at once not line-by-line ]]]]]]]] Since r.1-++
                                           } else { // if (Exact_flag) {

           //memset (wrdCACHEDT5, 0, MaxLineLength+1+1);
           i = 1;
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
           wrdlen = 0;
        for( k = WorkAreaLedgeT5; k <= WorkAreaRedgeT5; k++ )
           {
                      workbyte = xgamsCACHE[k];

                       if( wrdlen < MAXboth) {
                                                      if (CaseSensitiveWildcardMatching_flag == 0)
                                  wrdLOW[ wrdlen ] = KAZE_tolower( workbyte );
                                                      else
                                  wrdLOW[ wrdlen ] = ( workbyte );
                                           wrd[ wrdlen ] = workbyte;
                                 }
                       if (workbyte == 10) {
                                           TotalLines5++;
// Wildcard search [
                       if ( 0 < wrdlen && wrdlen < MAXboth)
                                 {
                                           wrd[ wrdlen ] = 0;
                                           wrdLOW[ wrdlen ] = 0;
                                           if ( wrd[ wrdlen-1 ] == 13 ) //CR
                                                      {--wrdlen; wrd[ wrdlen ] = 0; wrdLOW[ wrdlen ] = 0;}

if (WILDCARD_IP_flag) {
// WILDCARD IP [
                                           WordsChecked5++;
                                           if (Exact_flag) {
                                                      //if ((long)( Railgun_Quadruplet_7Gulliver_5(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                      #ifdef RG7Gulliver
                                                                 //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_5(wrdLOW, wrdARG, wrdlen, n);
                                                                 FOUNDinPTR = Railgun_Sekireigan_Wolfram_5(wrdLOW, wrdARG, wrdlen, n);
                                                      #else
                                                                 FOUNDinPTR = Railgun_Quadruplet_7_5(wrdLOW, wrdARG, wrdlen, n);
                                                      #endif
                                                      // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                      // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                      //if ((long)( FOUNDinPTR - wrdLOW )>=0)
                                                      if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(wrdLOW))
                                                      //if ((long)( Railgun_Quadruplet_7_5(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                                 {
                                                      // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                      #pragma omp critical
#endif
                                                      {
                                                                 DumpedLines5++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                      }
                                                                 }
                                           } else {
                                                      if (WILDCARD_FAST_flag) {
#if defined(_WildFastKaze_)
                                                      if ( WildcardMatch_Iterative_Kaze5(wrdARG, wrdLOW) ) {
#else
                                                      if ( IterativeWildcards5(wrdARG, wrdLOW) ) {
#endif
                                                      // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                      #pragma omp critical
```

```
#endif
                                                        {
                                                                DumpedLines5++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                        }
                                                        } else {
                                                        maskGLOBALlen = n;
                                                        nameGLOBALlen5 = wrdlen;
                                                        if ( EnhancedMaskTest_OrEmpty_AndNotEmpty_5(wrdARG, 0, wrdLOW, 0) ) { // Caution: Not lowercased as it should!
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                DumpedLines5++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                        }
                                                        }
                                                }
// WILDCARD IP ]
} else {

// A simple heuristic #1: Don't enter the nasty loops unless MaximumLevenshteinDistance >= ABS(m-n).
m = wrdlen; // strlen(wrd);
//if (m>MaxLineLength)
//{ printf( "\nKazahana: Incoming xgram exceeding the limit.\n" ); exit( 2 ); }
// Above two commented lines are too severe, changed with next line allowing to search into lines bigger than our needs:
if (m)
if (EXHAUSTIVE_flag == 1) {

// Here we'll walk through the whole length of 'm', ay-yaa.
// EXHAUSTIVE [[[[[[[[[[[[[[[[[[[[[[[[[[[[[[
// Here the old fuzzy is replaced with exhaustive one, using BBs of the incoming string (here 'm').
// We need to factorize 'm' down to all 'n+AtMostLevenshteinDistance' long strings/BBs and to search into them ONE-BY-ONE - a gruelling task indeed!
//// Example:
// One of those 33/5 lines is a 46 bytes long string, i.e. m = 46:
// |Edelweiß || [[edelweiss]] || edelweiss flower
// Let's search fuzzily for "edelvais", pretending we don't know the right spelling:
// To find "edelweiss" we need at least Levenshtein distance 3, i.e. n = 8, AtMostLevenshteinDistance = 3:
// 'edelvais' vs 'edelweiss':
// - the mispelled one has one character less;
// - the mispelled one has 2 wrong characters: 'v' & 'a' instead of 'w' & 'e'.
// From above we need Building-Blocks of 46 bytes order 8+3.
// Inhere we are using order 11, 'm - Order + 1' is the number of total BBs for text 'm' bytes long: 46-11+1 = 36
// The 36 BBs:
// |Edelweiß |
// Edelweiß ||
// ...
// weiss flowe
// eiss flower

// Slow! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
/*
                                for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                for (l=0; l < m-BB+1; l++) {
            WordsChecked5++;
// LD [
            SkipHeuristic=0;
            for(i=1;i<=BB;i++) {
                        for (j=1;j<=n;j++) {
                                if(wrdLOW[l+i-1] == wrdARG[j-1])
                                        LevenshteinT5[i][j] = LevenshteinT5[i-1][j-1];
                                else
#if defined(_WIN32ASM_)
                                        LevenshteinT5[i][j] = min_AF(LevenshteinT5[i-1][j]+1, LevenshteinT5[i][j-1]+1, LevenshteinT5[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                        LevenshteinT5[i][j] = MIN(MIN((LevenshteinT5[i-1][j]+1),(LevenshteinT5[i][j-1]+1)),(LevenshteinT5[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                        //{LevenshteinT5[i][j] = MIN(MIN(LevenshteinT5[i-1][j],LevenshteinT5[i][j-1]),LevenshteinT5[i-1][j-1]); --LevenshteinT5[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
```

```
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. BB) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinT5[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT5[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                                    if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT5[BB][n]) {
                                                              // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                              #pragma omp critical
#endif
                                                              {
                                                              fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines5++;
                                                              }
                                                              goto EXHAUSTIVE5;
                                                              //break; // No need of further checking down the line, one dump only is needed.
                                                              //if ((DumpedLines & 0xff) == 0xff)
                                                              //         //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                              //         fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                    }
// LD ]
                                          }
                                          }
EXHAUSTIVE5:;
*/
// Slow! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Fast! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

                                          for (l=0; l < m-(n-AtMostLevenshteinDistance)+1; l++) { // Here BB = n-AtMostLevenshteinDistance the smallest BB
                                          for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                          // From here on 'm' will be replaced by BB
                                          if (l < m-BB+1) {

if (AtMostLevenshteinDistance >= MAX(BB,n)-MIN(BB,n)) // This is the only add-on for r.1+
{
            WordsChecked5++;
// LD [
StartingPosition = 1;
while (     wrdCACHEDT5[StartingPosition-1] && wrdLOW[l+StartingPosition-1] && wrdCACHEDT5[StartingPosition-1]==wrdLOW[l+StartingPosition-1] )
            StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

            SkipHeuristic=0;
            for(i=StartingPosition;i<=BB;i++) { // StartingPosition is in range 1..
                        for (j=1;j<=n;j++) {
                                    if(wrdLOW[l+i-1] == wrdARG[j-1])
                                                LevenshteinT5[i][j] = LevenshteinT5[i-1][j-1];
                                    else
#if defined(_WIN32ASM_)
                                                LevenshteinT5[i][j] = min_AF(LevenshteinT5[i-1][j]+1, LevenshteinT5[i][j-1]+1, LevenshteinT5[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                                LevenshteinT5[i][j] = MIN(MIN((LevenshteinT5[i-1][j]+1),(LevenshteinT5[i][j-1]+1)),(LevenshteinT5[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                                //{LevenshteinT5[i][j] = MIN(MIN(LevenshteinT5[i-1][j],LevenshteinT5[i][j-1]),LevenshteinT5[i-1][j-1]); --LevenshteinT5[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinT5[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT5[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                                    if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT5[BB][n]) {
                                                              // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                              #pragma omp critical
#endif
                                                              {
                                                              fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines5++;
```

```
					}
					// Once dumping the line we need double 'break' from BB and l 'for's:
					goto EXHAUSTIVE5;
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
					//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
				}
	// The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
	//memcpy(wrdCACHEDT5, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
	memcpy(wrdCACHEDT5, &wrdLOW[l], BB); wrdCACHEDT5[BB]=0;
// LD ]
}

			} // if (l < m-BB+1)
			}
			}
EXHAUSTIVE5:;

// Fast! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// EXHAUSTIVE ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]

} else {
if (m<=MaxLineLength)
if (AtMostLevenshteinDistance >= MAX(m,n)-MIN(m,n)) // This is the only add-on for r.1+
{
	WordsChecked5++;
// LD [
StartingPosition = 1;
while (	wrdCACHEDT5[StartingPosition-1] && wrdCACHEDT5[StartingPosition-1]==wrdLOW[StartingPosition-1] ) // No need of && wrd[StartingPosition-1]
	StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

	SkipHeuristic=0;
	for(i=StartingPosition;i<=m;i++) { // StartingPosition is in range 1..
		for (j=1;j<=n;j++) {
			if(wrdLOW[i-1] == wrdARG[j-1])
				LevenshteinT5[i][j] = LevenshteinT5[i-1][j-1];
			else
#if defined(_WIN32ASM_)
				LevenshteinT5[i][j] = min_AF(LevenshteinT5[i-1][j]+1, LevenshteinT5[i][j-1]+1, LevenshteinT5[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinT5[i][j] = MIN(MIN((LevenshteinT5[i-1][j]+1),(LevenshteinT5[i][j-1]+1)),(LevenshteinT5[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinT5[i][j] = MIN(MIN(LevenshteinT5[i-1][j],LevenshteinT5[i][j-1]),LevenshteinT5[i-1][j-1]); --LevenshteinT5[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
		}

		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinT5[i][n] - (m-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT5[i][n] - (m-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT5[m][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines5++;
					}
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
					//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
				}
	// The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
	memcpy(wrdCACHEDT5, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
// LD ]
}
} //if (EXHAUSTIVE_flag == 1)
```

```
} //if (WILDCARD_IP_flag)
                                        }
// Wildcard search ]
                                                wrdlen = 0;
                                }
                                else wrdlen++;
        } // k 'for'
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                                } // if (Exact_flag) {
           }

// 6th thread
#ifdef Commence_OpenMP
    #pragma omp section
#endif
           {

                                                if (Exact_flag) {
// WHOLE buffer at once not line-by-line [[[[[[[[ Since r.1-++
                                                        k = WorkAreaLedgeT6;
                                                        while ( k < WorkAreaRedgeT6 ) {
                                                        #ifdef RG7Gulliver
                                                                //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_6(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT6-k+1, n);
                                                                FOUNDinPTR = Railgun_Sekireigan_Wolfram_6(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT6-k+1, n);
                                                        #else
                                                                FOUNDinPTR = Railgun_Quadruplet_7_6(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT6-k+1, n);
                                                        #endif
                                                        // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                        // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                        //if ((long)( FOUNDinPTR - &xgamsCACHE[k] )>=0) {
                                                        if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(&xgamsCACHE[k])) {
                                                                i = k + (long)( FOUNDinPTR - &xgamsCACHE[k] ); j = i;
                                                                while (xgamsCACHE[j] != 10) {++j;} // Works both on UNIX(LF) and Windows(CRLF)
                                                                while (i > k && xgamsCACHE[i-1] != 10) {--i;}
                                                                k = j+1; // Should "point" to first symbol after the dumped fragment.

                                                                //fwrite( &xgamsCACHE[i], j - i + 1, 1, fp_outLINE ); DumpedLines1++;
                                                                if (j - i + 1 <= 168*MaxLineLength) { // fix for 1-++
                                                                memcpy(wrd, &xgamsCACHE[i], j - i + 1); wrd[j - i + 1]=0; DumpedLines6++;
                                                                j--; if (wrd[j - i + 1]==10) wrd[j - i + 1]=0;
                                                                j--; if (wrd[j - i + 1]==13) wrd[j - i + 1]=0;
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                if (Dump_flag)
                                                                fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]);
                                                                else
                                                                fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                                }
                                                        } else k = WorkAreaRedgeT6;
                                                        } // while
// WHOLE buffer at once not line-by-line ]]]]]]]] Since r.1-++
                                                } else { // if (Exact_flag) {

           //memset (wrdCACHEDT6, 0, MaxLineLength+1+1);
           i = 1;
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
           wrdlen = 0;
        for( k = WorkAreaLedgeT6; k <= WorkAreaRedgeT6; k++ )
           {
                        workbyte = xgamsCACHE[k];

                         if( wrdlen < MAXboth) {
                                                        if (CaseSensitiveWildcardMatching_flag == 0)
                                        wrdLOW[ wrdlen ] = KAZE_tolower( workbyte );
```

```
                                                        else
                                        wrdLOW[ wrdlen ] = ( workbyte );
                                                wrd[ wrdlen ] = workbyte;
                                        }
                                if (workbyte == 10) {
                                                TotalLines6++;
// Wildcard search [
                                if ( 0 < wrdlen && wrdlen < MAXboth)
                                        {
                                                wrd[ wrdlen ] = 0;
                                                wrdLOW[ wrdlen ] = 0;
                                                if ( wrd[ wrdlen-1 ] == 13 ) //CR
                                                        {--wrdlen; wrd[ wrdlen ] = 0; wrdLOW[ wrdlen ] = 0;}

if (WILDCARD_IP_flag) {
// WILDCARD IP [
                                                WordsChecked6++;
                                                if (Exact_flag) {
                                                        //if ((long)( Railgun_Quadruplet_7Gulliver_6(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                        #ifdef RG7Gulliver
                                                                //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_6(wrdLOW, wrdARG, wrdlen, n);
                                                                FOUNDinPTR = Railgun_Sekireigan_Wolfram_6(wrdLOW, wrdARG, wrdlen, n);
                                                        #else
                                                                FOUNDinPTR = Railgun_Quadruplet_7_6(wrdLOW, wrdARG, wrdlen, n);
                                                        #endif
                                                        // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                        // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                        //if ((long)( FOUNDinPTR - wrdLOW )>=0)
                                                        if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(wrdLOW))
                                                        //if ((long)( Railgun_Quadruplet_7_6(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                                {
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                DumpedLines6++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                                }
                                                } else {
                                                        if (WILDCARD_FAST_flag) {
#if defined(_WildFastKaze_)
                                                        if ( WildcardMatch_Iterative_Kaze6(wrdARG, wrdLOW) ) {
#else
                                                        if ( IterativeWildcards6(wrdARG, wrdLOW) ) {
#endif
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                DumpedLines6++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                        }
                                                        } else {
                                                        maskGLOBALlen = n;
                                                        nameGLOBALlen6 = wrdlen;
                                                        if ( EnhancedMaskTest_OrEmpty_AndNotEmpty_6(wrdARG, 0, wrdLOW, 0) ) { // Caution: Not lowercased as it should!
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                DumpedLines6++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                        }
                                                        }
                                                }
// WILDCARD IP ]
```

```
} else {

// A simple heuristic #1: Don't enter the nasty loops unless MaximumLevenshteinDistance >= ABS(m-n).
m = wrdlen; // strlen(wrd);
//if (m>MaxLineLength)
//{ printf( "\nKazahana: Incoming xgram exceeding the limit.\n" ); exit( 2 ); }
// Above two commented lines are too severe, changed with next line allowing to search into lines bigger than our needs:
if (m)
if (EXHAUSTIVE_flag == 1) {

// Here we'll walk through the whole length of 'm', ay-yaa.
// EXHAUSTIVE [[[[[[[[[[[[[[[[[[[[[[[[[[[[[[
// Here the old fuzzy is replaced with exhaustive one, using BBs of the incoming string (here 'm').
// We need to factorize 'm' down to all 'n+AtMostLevenshteinDistance' long strings/BBs and to search into them ONE-BY-ONE - a gruelling task indeed!
//// Example:
// One of those 33/5 lines is a 46 bytes long string, i.e. m = 46:
// |Edelweiß || [[edelweiss]] || edelweiss flower
// Let's search fuzzily for "edelvais", pretending we don't know the right spelling:
// To find "edelweiss" we need at least Levenshtein distance 3, i.e. n = 8, AtMostLevenshteinDistance = 3:
// 'edelvais' vs 'edelweiss':
// - the mispelled one has one character less;
// - the mispelled one has 2 wrong characters: 'v' & 'a' instead of 'w' & 'e'.
// From above we need Building-Blocks of 46 bytes order 8+3.
// Inhere we are using order 11, 'm - Order + 1' is the number of total BBs for text 'm' bytes long: 46-11+1 = 36
// The 36 BBs:
// |Edelweiß |
// Edelweiß ||
// ...
// weiss flowe
// eiss flower

// Slow! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
/*
                                for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                for (l=0; l < m-BB+1; l++) {
            WordsChecked6++;
// LD [
            SkipHeuristic=0;
            for(i=1;i<=BB;i++) {
                        for (j=1;j<=n;j++) {
                                    if(wrdLOW[l+i-1] == wrdARG[j-1])
                                                LevenshteinT6[i][j] = LevenshteinT6[i-1][j-1];
                                    else
#if defined(_WIN32ASM_)
                                                LevenshteinT6[i][j] = min_AF(LevenshteinT6[i-1][j]+1, LevenshteinT6[i][j-1]+1, LevenshteinT6[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                                LevenshteinT6[i][j] = MIN(MIN((LevenshteinT6[i-1][j]+1),(LevenshteinT6[i][j-1]+1)),(LevenshteinT6[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                                //{LevenshteinT6[i][j] = MIN(MIN(LevenshteinT6[i-1][j],LevenshteinT6[i][j-1]),LevenshteinT6[i-1][j-1]); --LevenshteinT6[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. BB) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinT6[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT6[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                                if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT6[BB][n]) {
                                                            // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                            #pragma omp critical
#endif
                                                            {
                                                            fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines6++;
                                                            }
                                                            goto EXHAUSTIVE6;
                                                            //break; // No need of further checking down the line, one dump only is needed.
                                                            //if ((DumpedLines & 0xff) == 0xff)
                                                            //          //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                            //          fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                }
```

```
// LD ]
                                        }
                                        }
EXHAUSTIVE6:;
*/
// Slow! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Fast! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

                                        for (l=0; l < m-(n-AtMostLevenshteinDistance)+1; l++) { // Here BB = n-AtMostLevenshteinDistance the smallest BB
                                        for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                        // From here on 'm' will be replaced by BB
                                        if (l < m-BB+1) {

if (AtMostLevenshteinDistance >= MAX(BB,n)-MIN(BB,n)) // This is the only add-on for r.1+
{
            WordsChecked6++;
// LD [
StartingPosition = 1;
while (     wrdCACHEDT6[StartingPosition-1] && wrdLOW[l+StartingPosition-1] && wrdCACHEDT6[StartingPosition-1]==wrdLOW[l+StartingPosition-1] )
            StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

            SkipHeuristic=0;
            for(i=StartingPosition;i<=BB;i++) { // StartingPosition is in range 1..
                        for (j=1;j<=n;j++) {
                                    if(wrdLOW[l+i-1] == wrdARG[j-1])
                                                LevenshteinT6[i][j] = LevenshteinT6[i-1][j-1];
                                    else
#if defined(_WIN32ASM_)
                                                LevenshteinT6[i][j] = min_AF(LevenshteinT6[i-1][j]+1, LevenshteinT6[i][j-1]+1, LevenshteinT6[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                                LevenshteinT6[i][j] = MIN(MIN((LevenshteinT6[i-1][j]+1),(LevenshteinT6[i][j-1]+1)),(LevenshteinT6[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                                //{LevenshteinT6[i][j] = MIN(MIN(LevenshteinT6[i-1][j],LevenshteinT6[i][j-1]),LevenshteinT6[i-1][j-1]); --LevenshteinT6[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinT6[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT6[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                                if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT6[BB][n]) {
                                                            // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                            #pragma omp critical
#endif
                                                            {
                                                            fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines6++;
                                                            }
                                                            // Once dumping the line we need double 'break' from BB and l 'for's:
                                                            goto EXHAUSTIVE6;
                                                            //if ((DumpedLines & 0xff) == 0xff)
                                                            //         //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                            //         fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                }
            // The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
            //memcpy(wrdCACHEDT6, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
            memcpy(wrdCACHEDT6, &wrdLOW[l], BB); wrdCACHEDT6[BB]=0;
// LD ]
}

                                        } // if (l < m-BB+1)
                                        }
                                        }
EXHAUSTIVE6:;

// Fast! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

```
// EXHAUSTIVE ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]

} else {
if (m<=MaxLineLength)
if (AtMostLevenshteinDistance >= MAX(m,n)-MIN(m,n)) // This is the only add-on for r.1+
{
	WordsChecked6++;
// LD [
StartingPosition = 1;
while (	wrdCACHEDT6[StartingPosition-1] && wrdCACHEDT6[StartingPosition-1]==wrdLOW[StartingPosition-1] ) // No need of && wrd[StartingPosition-1]
	StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

	SkipHeuristic=0;
	for(i=StartingPosition;i<=m;i++) { // StartingPosition is in range 1..
		for (j=1;j<=n;j++) {
			if(wrdLOW[i-1] == wrdARG[j-1])
				LevenshteinT6[i][j] = LevenshteinT6[i-1][j-1];
			else
#if defined(_WIN32ASM_)
				LevenshteinT6[i][j] = min_AF(LevenshteinT6[i-1][j]+1, LevenshteinT6[i][j-1]+1, LevenshteinT6[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinT6[i][j] = MIN(MIN((LevenshteinT6[i-1][j]+1),(LevenshteinT6[i][j-1]+1)),(LevenshteinT6[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinT6[i][j] = MIN(MIN(LevenshteinT6[i-1][j],LevenshteinT6[i][j-1]),LevenshteinT6[i-1][j-1]); --LevenshteinT6[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
		}

		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinT6[i][n] - (m-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT6[i][n] - (m-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT6[m][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines6++;
					}
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
					//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
				}
	// The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
	memcpy(wrdCACHEDT6, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
// LD ]
}
} //if (EXHAUSTIVE_flag == 1)

} //if (WILDCARD_IP_flag)
			}
// Wildcard search ]
				wrdlen = 0;
		 }
		 else wrdlen++;
        } // k 'for'
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
				} // if (Exact_flag) {
	}

// 7th thread
#ifdef Commence_OpenMP
    #pragma omp section
#endif
	{

				if (Exact_flag) {
// WHOLE buffer at once not line-by-line [[[[[[[[ Since r.1-++
```

```
                                                k = WorkAreaLedgeT7;
                                                while ( k < WorkAreaRedgeT7 ) {
                                                #ifdef RG7Gulliver
                                                          //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_7(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT7-k+1, n);
                                                          FOUNDinPTR = Railgun_Sekireigan_Wolfram_7(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT7-k+1, n);
                                                #else
                                                          FOUNDinPTR = Railgun_Quadruplet_7_7(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT7-k+1, n);
                                                #endif
                                                // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                //if ((long)( FOUNDinPTR - &xgamsCACHE[k] )>=0) {
                                                if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(&xgamsCACHE[k])) {
                                                          i = k + (long)( FOUNDinPTR - &xgamsCACHE[k] ); j = i;
                                                          while (xgamsCACHE[j] != 10) {++j;} // Works both on UNIX(LF) and Windows(CRLF)
                                                          while (i > k && xgamsCACHE[i-1] != 10) {--i;}
                                                          k = j+1; // Should "point" to first symbol after the dumped fragment.

                                                          //fwrite( &xgamsCACHE[i], j - i + 1, 1, fp_outLINE ); DumpedLines1++;
                                                          if (j - i + 1 <= 168*MaxLineLength) { // fix for 1-++
                                                          memcpy(wrd, &xgamsCACHE[i], j - i + 1); wrd[j - i + 1]=0; DumpedLines7++;
                                                          j--; if (wrd[j - i + 1]==10) wrd[j - i + 1]=0;
                                                          j--; if (wrd[j - i + 1]==13) wrd[j - i + 1]=0;
                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                #pragma omp critical
#endif
                                                {
                                                          if (Dump_flag)
                                                          fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]);
                                                          else
                                                          fprintf( fp_outLINE, "%s\r\n", wrd);
                                                }
                                                          }
                                                } else k = WorkAreaRedgeT7;
                                                } // while
// WHOLE buffer at once not line-by-line ]]]]]]]] Since r.1-++
                                          } else { // if (Exact_flag) {

           //memset (wrdCACHEDT7, 0, MaxLineLength+1+1);
           i = 1;
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
           wrdlen = 0;
        for( k = WorkAreaLedgeT7; k <= WorkAreaRedgeT7; k++ )
           {
                     workbyte = xgamsCACHE[k];

                      if( wrdlen < MAXboth) {
                                                if (CaseSensitiveWildcardMatching_flag == 0)
                                 wrdLOW[ wrdlen ] = KAZE_tolower( workbyte );
                                                else
                                 wrdLOW[ wrdlen ] = ( workbyte );
                                           wrd[ wrdlen ] = workbyte;
                                 }
                      if (workbyte == 10) {
                                           TotalLines7++;
// Wildcard search [
                      if ( 0 < wrdlen && wrdlen < MAXboth)
                                 {
                                           wrd[ wrdlen ] = 0;
                                           wrdLOW[ wrdlen ] = 0;
                                           if ( wrd[ wrdlen-1 ] == 13 ) //CR
                                                {--wrdlen; wrd[ wrdlen ] = 0; wrdLOW[ wrdlen ] = 0;}

if (WILDCARD_IP_flag) {
// WILDCARD IP [
                                           WordsChecked7++;
                                           if (Exact_flag) {
                                                //if ((long)( Railgun_Quadruplet_7Gulliver_7(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                #ifdef RG7Gulliver
```

```
                                                        //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_7(wrdLOW, wrdARG, wrdlen, n);
                                                        FOUNDinPTR = Railgun_Sekireigan_Wolfram_7(wrdLOW, wrdARG, wrdlen, n);
                                                #else
                                                        FOUNDinPTR = Railgun_Quadruplet_7_7(wrdLOW, wrdARG, wrdlen, n);
                                                #endif
                                                // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                //if ((long)( FOUNDinPTR - wrdLOW )>=0)
                                                if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(wrdLOW))
                                                //if ((long)( Railgun_Quadruplet_7_7(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                        {
                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                #pragma omp critical
#endif
                                                {
                                                        DumpedLines7++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                }
                                                        }
                                        } else {
                                                if (WILDCARD_FAST_flag) {
#if defined(_WildFastKaze_)
                                                if ( WildcardMatch_Iterative_Kaze7(wrdARG, wrdLOW) ) {
#else
                                                if ( IterativeWildcards7(wrdARG, wrdLOW) ) {
#endif
                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                #pragma omp critical
#endif
                                                {
                                                        DumpedLines7++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                }
                                                }
                                                } else {
                                                maskGLOBALlen = n;
                                                nameGLOBALlen7 = wrdlen;
                                                if ( EnhancedMaskTest_OrEmpty_AndNotEmpty_7(wrdARG, 0, wrdLOW, 0) ) { // Caution: Not lowercased as it should!
                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                #pragma omp critical
#endif
                                                {
                                                        DumpedLines7++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                }
                                                }
                                                }
                                        }
// WILDCARD IP ]
} else {

// A simple heuristic #1: Don't enter the nasty loops unless MaximumLevenshteinDistance >= ABS(m-n).
m = wrdlen; // strlen(wrd);
//if (m>MaxLineLength)
//{ printf( "\nKazahana: Incoming xgram exceeding the limit.\n" ); exit( 2 ); }
// Above two commented lines are too severe, changed with next line allowing to search into lines bigger than our needs:
if (m)
if (EXHAUSTIVE_flag == 1) {

// Here we'll walk through the whole length of 'm', ay-yaa.
// EXHAUSTIVE [[[[[[[[[[[[[[[[[[[[[[[[[[[[[[
// Here the old fuzzy is replaced with exhaustive one, using BBs of the incoming string (here 'm').
// We need to factorize 'm' down to all 'n+AtMostLevenshteinDistance' long strings/BBs and to search into them ONE-BY-ONE - a gruelling task indeed!
//// Example:
// One of those 33/5 lines is a 46 bytes long string, i.e. m = 46:
// |Edelweiß || [[edelweiss]] || edelweiss flower
// Let's search fuzzily for "edelvais", pretending we don't know the right spelling:
// To find "edelweiss" we need at least Levenshtein distance 3, i.e. n = 8, AtMostLevenshteinDistance = 3:
// 'edelvais' vs 'edelweiss':
```

```
// - the mispelled one has one character less;
// - the mispelled one has 2 wrong characters: 'v' & 'a' instead of 'w' & 'e'.
// From above we need Building-Blocks of 46 bytes order 8+3.
// Inhere we are using order 11, 'm - Order + 1' is the number of total BBs for text 'm' bytes long: 46-11+1 = 36
// The 36 BBs:
// |Edelweiß |
// Edelweiß ||
// ...
// weiss flowe
// eiss flower

// Slow! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
/*
                                for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                for (l=0; l < m-BB+1; l++) {
            WordsChecked7++;
// LD [
            SkipHeuristic=0;
            for(i=1;i<=BB;i++) {
                        for (j=1;j<=n;j++) {
                                    if(wrdLOW[l+i-1] == wrdARG[j-1])
                                                LevenshteinT7[i][j] = LevenshteinT7[i-1][j-1];
                                    else
#if defined(_WIN32ASM_)
                                                LevenshteinT7[i][j] = min_AF(LevenshteinT7[i-1][j]+1, LevenshteinT7[i][j-1]+1, LevenshteinT7[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                                LevenshteinT7[i][j] = MIN(MIN((LevenshteinT7[i-1][j]+1),(LevenshteinT7[i][j-1]+1)),(LevenshteinT7[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                                //{LevenshteinT7[i][j] = MIN(MIN(LevenshteinT7[i-1][j],LevenshteinT7[i][j-1]),LevenshteinT7[i-1][j-1]); --LevenshteinT7[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. BB) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinT7[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT7[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                                if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT7[BB][n]) {
                                                            // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                            #pragma omp critical
#endif
                                                            {
                                                            fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines7++;
                                                            }
                                                            goto EXHAUSTIVE7;
                                                            //break; // No need of further checking down the line, one dump only is needed.
                                                            //if ((DumpedLines & 0xff) == 0xff)
                                                            //         //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                            //         fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                }
// LD ]
                                }
                                }
EXHAUSTIVE7:;
*/
// Slow! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Fast! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

                                for (l=0; l < m-(n-AtMostLevenshteinDistance)+1; l++) { // Here BB = n-AtMostLevenshteinDistance the smallest BB
                                for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                // From here on 'm' will be replaced by BB
                                if (l < m-BB+1) {

if (AtMostLevenshteinDistance >= MAX(BB,n)-MIN(BB,n)) // This is the only add-on for r.1+
{
            WordsChecked7++;
// LD [
StartingPosition = 1;
while (     wrdCACHEDT7[StartingPosition-1] && wrdLOW[l+StartingPosition-1] && wrdCACHEDT7[StartingPosition-1]==wrdLOW[l+StartingPosition-1] )
```

```
		StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

		SkipHeuristic=0;
		for(i=StartingPosition;i<=BB;i++) { // StartingPosition is in range 1..
			for (j=1;j<=n;j++) {
				if(wrdLOW[l+i-1] == wrdARG[j-1])
					LevenshteinT7[i][j] = LevenshteinT7[i-1][j-1];
				else
#if defined(_WIN32ASM_)
					LevenshteinT7[i][j] = min_AF(LevenshteinT7[i-1][j]+1, LevenshteinT7[i][j-1]+1, LevenshteinT7[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
					LevenshteinT7[i][j] = MIN(MIN((LevenshteinT7[i-1][j]+1),(LevenshteinT7[i][j-1]+1)),(LevenshteinT7[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
					//{LevenshteinT7[i][j] = MIN(MIN(LevenshteinT7[i-1][j],LevenshteinT7[i][j-1]),LevenshteinT7[i-1][j-1]); --LevenshteinT7[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! // 237,270 xgrams/s
#endif
			}

			// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
			if ( LevenshteinT7[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT7[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
		}
					if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT7[BB][n]) {
						// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
						#pragma omp critical
#endif
						{
						fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines7++;
						}
						// Once dumping the line we need double 'break' from BB and l 'for's:
						goto EXHAUSTIVE7;
						//if ((DumpedLines & 0xff) == 0xff)
						//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
						//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
					}
		// The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
		//memcpy(wrdCACHEDT7, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
		memcpy(wrdCACHEDT7, &wrdLOW[l], BB); wrdCACHEDT7[BB]=0;
// LD ]
}

				} // if (l < m-BB+1)
				}
				}
EXHAUSTIVE7:;

// Fast! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// EXHAUSTIVE ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]

} else {
if (m<=MaxLineLength)
if (AtMostLevenshteinDistance >= MAX(m,n)-MIN(m,n)) // This is the only add-on for r.1+
{
		WordsChecked7++;
// LD [
StartingPosition = 1;
while (	wrdCACHEDT7[StartingPosition-1] && wrdCACHEDT7[StartingPosition-1]==wrdLOW[StartingPosition-1] ) // No need of && wrd[StartingPosition-1]
		StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

		SkipHeuristic=0;
		for(i=StartingPosition;i<=m;i++) { // StartingPosition is in range 1..
			for (j=1;j<=n;j++) {
				if(wrdLOW[i-1] == wrdARG[j-1])
					LevenshteinT7[i][j] = LevenshteinT7[i-1][j-1];
				else
```

```
#if defined(_WIN32ASM_)
                                             LevenshteinT7[i][j] = min_AF(LevenshteinT7[i-1][j]+1, LevenshteinT7[i][j-1]+1, LevenshteinT7[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                             LevenshteinT7[i][j] = MIN(MIN((LevenshteinT7[i-1][j]+1),(LevenshteinT7[i][j-1]+1)),(LevenshteinT7[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                             //{LevenshteinT7[i][j] = MIN(MIN(LevenshteinT7[i-1][j],LevenshteinT7[i][j-1]),LevenshteinT7[i-1][j-1]); --LevenshteinT7[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                       }

                       // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                       if ( LevenshteinT7[i][n] - (m-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT7[i][n] - (m-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
           }
                                             if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT7[m][n]) {
                                                         // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                         #pragma omp critical
#endif
                                                         {
                                                         fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines7++;
                                                         }
                                                         //if ((DumpedLines & 0xff) == 0xff)
                                                         //         //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                         //         fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                             }
           // The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
           memcpy(wrdCACHEDT7, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
// LD ]
}
} //if (EXHAUSTIVE_flag == 1)

} //if (WILDCARD_IP_flag)
                                  }
// Wildcard search ]
                                             wrdlen = 0;
                        }
                        else wrdlen++;
        } // k 'for'
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                             } // if (Exact_flag) {
           }

// 8th thread
#ifdef Commence_OpenMP
    #pragma omp section
#endif
           {

                                             if (Exact_flag) {
// WHOLE buffer at once not line-by-line [[[[[[[ Since r.1-++
                                                         k = WorkAreaLedgeT8;
                                                         while ( k < WorkAreaRedgeT8 ) {
                                                         #ifdef RG7Gulliver
                                                                    //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_8(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT8-k+1, n);
                                                                    FOUNDinPTR = Railgun_Sekireigan_Wolfram_8(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT8-k+1, n);
                                                         #else
                                                                    FOUNDinPTR = Railgun_Quadruplet_7_8(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT8-k+1, n);
                                                         #endif
                                                         // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                         // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                         //if ((long)( FOUNDinPTR - &xgamsCACHE[k] )>=0) {
                                                         if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(&xgamsCACHE[k])) {
                                                                    i = k + (long)( FOUNDinPTR - &xgamsCACHE[k] ); j = i;
                                                                    while (xgamsCACHE[j] != 10) {++j;} // Works both on UNIX(LF) and Windows(CRLF)
                                                                    while (i > k && xgamsCACHE[i-1] != 10) {--i;}
                                                                    k = j+1; // Should "point" to first symbol after the dumped fragment.

                                                                    //fwrite( &xgamsCACHE[i], j - i + 1, 1, fp_outLINE ); DumpedLines1++;
                                                                    if (j - i + 1 <= 168*MaxLineLength) { // fix for 1-++
                                                                    memcpy(wrd, &xgamsCACHE[i], j - i + 1); wrd[j - i + 1]=0; DumpedLines8++;
```

```
                                                            j--; if (wrd[j - i + 1]==10) wrd[j - i + 1]=0;
                                                            j--; if (wrd[j - i + 1]==13) wrd[j - i + 1]=0;
                                                  // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                  #pragma omp critical
#endif
                                                  {
                                                            if (Dump_flag)
                                                            fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]);
                                                            else
                                                            fprintf( fp_outLINE, "%s\r\n", wrd);
                                                  }
                                                            }
                                                  } else k = WorkAreaRedgeT8;
                                                  } // while
// WHOLE buffer at once not line-by-line ]]]]]]]] Since r.1-++
                                        } else { // if (Exact_flag) {

          //memset (wrdCACHEDT8, 0, MaxLineLength+1+1);
          i = 1;
       //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
          wrdlen = 0;
       for( k = WorkAreaLedgeT8; k <= WorkAreaRedgeT8; k++ )
          {
                     workbyte = xgamsCACHE[k];

                      if( wrdlen < MAXboth) {
                                                  if (CaseSensitiveWildcardMatching_flag == 0)
                                 wrdLOW[ wrdlen ] = KAZE_tolower( workbyte );
                                                  else
                                 wrdLOW[ wrdlen ] = ( workbyte );
                                           wrd[ wrdlen ] = workbyte;
                                 }
                      if (workbyte == 10) {
                                           TotalLines8++;
// Wildcard search [
                      if ( 0 < wrdlen && wrdlen < MAXboth)
                                 {
                                           wrd[ wrdlen ] = 0;
                                           wrdLOW[ wrdlen ] = 0;
                                           if ( wrd[ wrdlen-1 ] == 13 ) //CR
                                                  {--wrdlen; wrd[ wrdlen ] = 0; wrdLOW[ wrdlen ] = 0;}

if (WILDCARD_IP_flag) {
// WILDCARD IP [
                                           WordsChecked8++;
                                           if (Exact_flag) {
                                                  //if ((long)( Railgun_Quadruplet_7Gulliver_8(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                  #ifdef RG7Gulliver
                                                            //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_8(wrdLOW, wrdARG, wrdlen, n);
                                                            FOUNDinPTR = Railgun_Sekireigan_Wolfram_8(wrdLOW, wrdARG, wrdlen, n);
                                                  #else
                                                            FOUNDinPTR = Railgun_Quadruplet_7_8(wrdLOW, wrdARG, wrdlen, n);
                                                  #endif
                                                  // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                  // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                  //if ((long)( FOUNDinPTR - wrdLOW )>=0)
                                                  if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(wrdLOW))
                                                  //if ((long)( Railgun_Quadruplet_7_8(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                            {
                                                  // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                  #pragma omp critical
#endif
                                                  {
                                                            DumpedLines8++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                  }
                                                            }
                                           } else {
```

```
                                                        if (WILDCARD_FAST_flag) {
#if defined(_WildFastKaze_)
                                                        if ( WildcardMatch_Iterative_Kaze8(wrdARG, wrdLOW) ) {
#else
                                                        if ( IterativeWildcards8(wrdARG, wrdLOW) ) {
#endif
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                DumpedLines8++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                        }
                                                        } else {
                                                        maskGLOBALlen = n;
                                                        nameGLOBALlen8 = wrdlen;
                                                        if ( EnhancedMaskTest_OrEmpty_AndNotEmpty_8(wrdARG, 0, wrdLOW, 0) ) { // Caution: Not lowercased as it should!
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                DumpedLines8++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                        }
                                                        }
                                                }
// WILDCARD IP ]
} else {

// A simple heuristic #1: Don't enter the nasty loops unless MaximumLevenshteinDistance >= ABS(m-n).
m = wrdlen; // strlen(wrd);
//if (m>MaxLineLength)
//{ printf( "\nKazahana: Incoming xgram exceeding the limit.\n" ); exit( 2 ); }
// Above two commented lines are too severe, changed with next line allowing to search into lines bigger than our needs:
if (m)
if (EXHAUSTIVE_flag == 1) {

// Here we'll walk through the whole length of 'm', ay-yaa.
// EXHAUSTIVE [[[[[[[[[[[[[[[[[[[[[[[[[[[[[
// Here the old fuzzy is replaced with exhaustive one, using BBs of the incoming string (here 'm').
// We need to factorize 'm' down to all 'n+AtMostLevenshteinDistance' long strings/BBs and to search into them ONE-BY-ONE - a gruelling task indeed!
//// Example:
// One of those 33/5 lines is a 46 bytes long string, i.e. m = 46:
// |Edelweiß || [[edelweiss]] || edelweiss flower
// Let's search fuzzily for "edelvais", pretending we don't know the right spelling:
// To find "edelweiss" we need at least Levenshtein distance 3, i.e. n = 8, AtMostLevenshteinDistance = 3:
// 'edelvais' vs 'edelweiss':
// - the mispelled one has one character less;
// - the mispelled one has 2 wrong characters: 'v' & 'a' instead of 'w' & 'e'.
// From above we need Building-Blocks of 46 bytes order 8+3.
// Inhere we are using order 11, 'm - Order + 1' is the number of total BBs for text 'm' bytes long: 46-11+1 = 36
// The 36 BBs:
// |Edelweiß |
// Edelweiß ||
// ...
// weiss flowe
// eiss flower

// Slow! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
/*
                                for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                for (l=0; l < m-BB+1; l++) {
            WordsChecked8++;
// LD [
            SkipHeuristic=0;
            for(i=1;i<=BB;i++) {
                        for (j=1;j<=n;j++) {
```

```
                                if(wrdLOW[l+i-1] == wrdARG[j-1])
                                            LevenshteinT8[i][j] = LevenshteinT8[i-1][j-1];
                                else
#if defined(_WIN32ASM_)
                                            LevenshteinT8[i][j] = min_AF(LevenshteinT8[i-1][j]+1, LevenshteinT8[i][j-1]+1, LevenshteinT8[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                            LevenshteinT8[i][j] = MIN(MIN((LevenshteinT8[i-1][j]+1),(LevenshteinT8[i][j-1]+1)),(LevenshteinT8[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                            //{LevenshteinT8[i][j] = MIN(MIN(LevenshteinT8[i-1][j],LevenshteinT8[i][j-1]),LevenshteinT8[i-1][j-1]); --LevenshteinT8[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. BB) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinT8[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT8[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                            if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT8[BB][n]) {
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                        fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines8++;
                                                        }
                                                        goto EXHAUSTIVE8;
                                                        //break; // No need of further checking down the line, one dump only is needed.
                                                        //if ((DumpedLines & 0xff) == 0xff)
                                                        //        //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                        //        fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                            }
// LD ]
                                }
                                }
EXHAUSTIVE8:;
*/
// Slow! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Fast! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

                                for (l=0; l < m-(n-AtMostLevenshteinDistance)+1; l++) { // Here BB = n-AtMostLevenshteinDistance the smallest BB
                                for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                // From here on 'm' will be replaced by BB
                                if (l < m-BB+1) {

if (AtMostLevenshteinDistance >= MAX(BB,n)-MIN(BB,n)) // This is the only add-on for r.1+
{
            WordsChecked8++;
// LD [
StartingPosition = 1;
while (     wrdCACHEDT8[StartingPosition-1] && wrdLOW[l+StartingPosition-1] && wrdCACHEDT8[StartingPosition-1]==wrdLOW[l+StartingPosition-1] )
            StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

            SkipHeuristic=0;
            for(i=StartingPosition;i<=BB;i++) { // StartingPosition is in range 1..
                        for (j=1;j<=n;j++) {
                                if(wrdLOW[l+i-1] == wrdARG[j-1])
                                            LevenshteinT8[i][j] = LevenshteinT8[i-1][j-1];
                                else
#if defined(_WIN32ASM_)
                                            LevenshteinT8[i][j] = min_AF(LevenshteinT8[i-1][j]+1, LevenshteinT8[i][j-1]+1, LevenshteinT8[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                            LevenshteinT8[i][j] = MIN(MIN((LevenshteinT8[i-1][j]+1),(LevenshteinT8[i][j-1]+1)),(LevenshteinT8[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                            //{LevenshteinT8[i][j] = MIN(MIN(LevenshteinT8[i-1][j],LevenshteinT8[i][j-1]),LevenshteinT8[i-1][j-1]); --LevenshteinT8[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
```

```
		if ( LevenshteinT8[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT8[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT8[BB][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines8++;
					}
					// Once dumping the line we need double 'break' from BB and l 'for's:
					goto EXHAUSTIVE8;
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
					//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
				}
	// The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
	//memcpy(wrdCACHEDT8, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
	memcpy(wrdCACHEDT8, &wrdLOW[l], BB); wrdCACHEDT8[BB]=0;
// LD ]
}

			} // if (l < m-BB+1)
			}
			}
EXHAUSTIVE8:;

// Fast! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// EXHAUSTIVE ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]

} else {
if (m<=MaxLineLength)
if (AtMostLevenshteinDistance >= MAX(m,n)-MIN(m,n)) // This is the only add-on for r.1+
{
	WordsChecked8++;
// LD [
StartingPosition = 1;
while (	wrdCACHEDT8[StartingPosition-1] && wrdCACHEDT8[StartingPosition-1]==wrdLOW[StartingPosition-1] ) // No need of && wrd[StartingPosition-1]
	StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

	SkipHeuristic=0;
	for(i=StartingPosition;i<=m;i++) { // StartingPosition is in range 1..
		for (j=1;j<=n;j++) {
			if(wrdLOW[i-1] == wrdARG[j-1])
				LevenshteinT8[i][j] = LevenshteinT8[i-1][j-1];
			else
#if defined(_WIN32ASM_)
				LevenshteinT8[i][j] = min_AF(LevenshteinT8[i-1][j]+1, LevenshteinT8[i][j-1]+1, LevenshteinT8[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinT8[i][j] = MIN(MIN((LevenshteinT8[i-1][j]+1),(LevenshteinT8[i][j-1]+1)),(LevenshteinT8[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinT8[i][j] = MIN(MIN(LevenshteinT8[i-1][j],LevenshteinT8[i][j-1]),LevenshteinT8[i-1][j-1]); --LevenshteinT8[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! // 
237,270 xgrams/s
#endif
		}

		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinT8[i][n] - (m-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT8[i][n] - (m-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT8[m][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines8++;
					}
```

```
                                                         //if ((DumpedLines & 0xff) == 0xff)
                                                         //         //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                         //         fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                              }
            // The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
            memcpy(wrdCACHEDT8, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
// LD ]
}
} //if (EXHAUSTIVE_flag == 1)

} //if (WILDCARD_IP_flag)
                                  }
// Wildcard search ]
                                              wrdlen = 0;
                        }
                        else wrdlen++;
        } // k 'for'
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                              } // if (Exact_flag) {
            }

// 9th thread
#ifdef Commence_OpenMP
    #pragma omp section
#endif
            {

                                              if (Exact_flag) {
// WHOLE buffer at once not line-by-line [[[[[[[[ Since r.1-++
                                                         k = WorkAreaLedgeT9;
                                                         while ( k < WorkAreaRedgeT9 ) {
                                                         #ifdef RG7Gulliver
                                                                    //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_9(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT9-k+1, n);
                                                                    FOUNDinPTR = Railgun_Sekireigan_Wolfram_9(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT9-k+1, n);
                                                         #else
                                                                    FOUNDinPTR = Railgun_Quadruplet_7_9(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT9-k+1, n);
                                                         #endif
                                                         // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                         // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                         //if ((long)( FOUNDinPTR - &xgamsCACHE[k] )>=0) {
                                                         if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(&xgamsCACHE[k])) {
                                                                    i = k + (long)( FOUNDinPTR - &xgamsCACHE[k] ); j = i;
                                                                    while (xgamsCACHE[j] != 10) {++j;} // Works both on UNIX(LF) and Windows(CRLF)
                                                                    while (i > k && xgamsCACHE[i-1] != 10) {--i;}
                                                                    k = j+1; // Should "point" to first symbol after the dumped fragment.

                                                                    //fwrite( &xgamsCACHE[i], j - i + 1, 1, fp_outLINE ); DumpedLines1++;
                                                                    if (j - i + 1 <= 168*MaxLineLength) { // fix for 1-++
                                                                    memcpy(wrd, &xgamsCACHE[i], j - i + 1); wrd[j - i + 1]=0; DumpedLines9++;
                                                                    j--; if (wrd[j - i + 1]==10) wrd[j - i + 1]=0;
                                                                    j--; if (wrd[j - i + 1]==13) wrd[j - i + 1]=0;
                                                         // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP

                                                         #pragma omp critical

#endif

                                                         {
                                                                    if (Dump_flag)
                                                                    fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]);
                                                                    else
                                                                    fprintf( fp_outLINE, "%s\r\n", wrd);
                                                         }
                                                                    }
                                                         } else k = WorkAreaRedgeT9;
                                                         } // while
// WHOLE buffer at once not line-by-line ]]]]]]]] Since r.1-++
                                              } else { // if (Exact_flag) {

            //memset (wrdCACHEDT9, 0, MaxLineLength+1+1);
            i = 1;
```

```
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            wrdlen = 0;
        for( k = WorkAreaLedgeT9; k <= WorkAreaRedgeT9; k++ )
            {
                       workbyte = xgamsCACHE[k];

                        if( wrdlen < MAXboth) {
                                                         if (CaseSensitiveWildcardMatching_flag == 0)
                                    wrdLOW[ wrdlen ] = KAZE_tolower( workbyte );
                                                         else
                                    wrdLOW[ wrdlen ] = ( workbyte );
                                              wrd[ wrdlen ] = workbyte;
                                  }
                        if (workbyte == 10) {
                                              TotalLines9++;
// Wildcard search [
                        if ( 0 < wrdlen && wrdlen < MAXboth)
                                  {
                                              wrd[ wrdlen ] = 0;
                                              wrdLOW[ wrdlen ] = 0;
                                              if ( wrd[ wrdlen-1 ] == 13 ) //CR
                                                         {--wrdlen; wrd[ wrdlen ] = 0; wrdLOW[ wrdlen ] = 0;}

if (WILDCARD_IP_flag) {
// WILDCARD IP [
                                              WordsChecked9++;
                                              if (Exact_flag) {
                                                         //if ((long)( Railgun_Quadruplet_7Gulliver_9(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                         #ifdef RG7Gulliver
                                                                    //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_9(wrdLOW, wrdARG, wrdlen, n);
                                                                    FOUNDinPTR = Railgun_Sekireigan_Wolfram_9(wrdLOW, wrdARG, wrdlen, n);
                                                         #else
                                                                    FOUNDinPTR = Railgun_Quadruplet_7_9(wrdLOW, wrdARG, wrdlen, n);
                                                         #endif
                                                         // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                         // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                         //if ((long)( FOUNDinPTR - wrdLOW )>=0)
                                                         if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(wrdLOW))
                                                         //if ((long)( Railgun_Quadruplet_7_9(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                                    {
                                                         // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                         #pragma omp critical
#endif
                                                         {
                                                                    DumpedLines9++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                         }
                                                                    }
                                              } else {
                                                         if (WILDCARD_FAST_flag) {
#if defined(_WildFastKaze_)
                                                         if ( WildcardMatch_Iterative_Kaze9(wrdARG, wrdLOW) ) {
#else
                                                         if ( IterativeWildcards9(wrdARG, wrdLOW) ) {
#endif
                                                         // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                         #pragma omp critical
#endif
                                                         {
                                                                    DumpedLines9++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                         }
                                                         }
                                                         } else {
                                                         maskGLOBALlen = n;
                                                         nameGLOBALlen9 = wrdlen;
                                                         if ( EnhancedMaskTest_OrEmpty_AndNotEmpty_9(wrdARG, 0, wrdLOW, 0) ) { // Caution: Not lowercased as it should!
                                                         // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
```

```
                                                                #pragma omp critical
#endif
                                                                {
                                                                          DumpedLines9++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                                }
                                                                }
                                                                }
                                                      }
// WILDCARD IP ]
} else {

// A simple heuristic #1: Don't enter the nasty loops unless MaximumLevenshteinDistance >= ABS(m-n).
m = wrdlen; // strlen(wrd);
//if (m>MaxLineLength)
//{ printf( "\nKazahana: Incoming xgram exceeding the limit.\n" ); exit( 2 ); }
// Above two commented lines are too severe, changed with next line allowing to search into lines bigger than our needs:
if (m)
if (EXHAUSTIVE_flag == 1) {

// Here we'll walk through the whole length of 'm', ay-yaa.
// EXHAUSTIVE [[[[[[[[[[[[[[[[[[[[[[[[[[[[
// Here the old fuzzy is replaced with exhaustive one, using BBs of the incoming string (here 'm').
// We need to factorize 'm' down to all 'n+AtMostLevenshteinDistance' long strings/BBs and to search into them ONE-BY-ONE - a gruelling task indeed!
//// Example:
// One of those 33/5 lines is a 46 bytes long string, i.e. m = 46:
// |Edelweiß || [[edelweiss]] || edelweiss flower
// Let's search fuzzily for "edelvais", pretending we don't know the right spelling:
// To find "edelweiss" we need at least Levenshtein distance 3, i.e. n = 8, AtMostLevenshteinDistance = 3:
// 'edelvais' vs 'edelweiss':
// - the mispelled one has one character less;
// - the mispelled one has 2 wrong characters: 'v' & 'a' instead of 'w' & 'e'.
// From above we need Building-Blocks of 46 bytes order 8+3.
// Inhere we are using order 11, 'm - Order + 1' is the number of total BBs for text 'm' bytes long: 46-11+1 = 36
// The 36 BBs:
// |Edelweiß |
// Edelweiß ||
// ...
// weiss flowe
// eiss flower

// Slow! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
/*
                                  for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                  for (l=0; l < m-BB+1; l++) {
            WordsChecked9++;
// LD [
            SkipHeuristic=0;
            for(i=1;i<=BB;i++) {
                        for (j=1;j<=n;j++) {
                                  if(wrdLOW[l+i-1] == wrdARG[j-1])
                                            LevenshteinT9[i][j] = LevenshteinT9[i-1][j-1];
                                  else
#if defined(_WIN32ASM_)
                                            LevenshteinT9[i][j] = min_AF(LevenshteinT9[i-1][j]+1, LevenshteinT9[i][j-1]+1, LevenshteinT9[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                            LevenshteinT9[i][j] = MIN(MIN((LevenshteinT9[i-1][j]+1),(LevenshteinT9[i][j-1]+1)),(LevenshteinT9[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                            //{LevenshteinT9[i][j] = MIN(MIN(LevenshteinT9[i-1][j],LevenshteinT9[i][j-1]),LevenshteinT9[i-1][j-1]); --LevenshteinT9[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. BB) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinT9[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT9[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                            if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT9[BB][n]) {
                                                      // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                      #pragma omp critical
#endif
```

```
                                                        {
                                                        fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines9++;
                                                        }
                                                        goto EXHAUSTIVE9;
                                                        //break; // No need of further checking down the line, one dump only is needed.
                                                        //if ((DumpedLines & 0xff) == 0xff)
                                                        //        //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                        //        fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                }
// LD ]
                                        }
                                        }
EXHAUSTIVE9:;
*/
// Slow! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Fast! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

                                        for (l=0; l < m-(n-AtMostLevenshteinDistance)+1; l++) { // Here BB = n-AtMostLevenshteinDistance the smallest BB
                                        for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                        // From here on 'm' will be replaced by BB
                                        if (l < m-BB+1) {

if (AtMostLevenshteinDistance >= MAX(BB,n)-MIN(BB,n)) // This is the only add-on for r.1+
{
                WordsChecked9++;
// LD [
StartingPosition = 1;
while (     wrdCACHEDT9[StartingPosition-1] && wrdLOW[l+StartingPosition-1] && wrdCACHEDT9[StartingPosition-1]==wrdLOW[l+StartingPosition-1] )
                StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

                SkipHeuristic=0;
                for(i=StartingPosition;i<=BB;i++) { // StartingPosition is in range 1..
                        for (j=1;j<=n;j++) {
                                        if(wrdLOW[l+i-1] == wrdARG[j-1])
                                                LevenshteinT9[i][j] = LevenshteinT9[i-1][j-1];
                                        else
#if defined(_WIN32ASM_)
                                                LevenshteinT9[i][j] = min_AF(LevenshteinT9[i-1][j]+1, LevenshteinT9[i][j-1]+1, LevenshteinT9[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                                LevenshteinT9[i][j] = MIN(MIN((LevenshteinT9[i-1][j]+1),(LevenshteinT9[i][j-1]+1)),(LevenshteinT9[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                                //{LevenshteinT9[i][j] = MIN(MIN(LevenshteinT9[i-1][j],LevenshteinT9[i][j-1]),LevenshteinT9[i-1][j-1]); --LevenshteinT9[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinT9[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT9[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
                }
                                                if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT9[BB][n]) {
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                        fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines9++;
                                                        }
                                                        // Once dumping the line we need double 'break' from BB and l 'for's:
                                                        goto EXHAUSTIVE9;
                                                        //if ((DumpedLines & 0xff) == 0xff)
                                                        //        //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                        //        fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                }
                // The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
                //memcpy(wrdCACHEDT9, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
                memcpy(wrdCACHEDT9, &wrdLOW[l], BB); wrdCACHEDT9[BB]=0;
// LD ]
```

```
}

                                    } // if (l < m-BB+1)
                                    }
                                    }
EXHAUSTIVE9:;

// Fast! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// EXHAUSTIVE ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]

} else {
if (m<=MaxLineLength)
if (AtMostLevenshteinDistance >= MAX(m,n)-MIN(m,n)) // This is the only add-on for r.1+
{
            WordsChecked9++;
// LD [
StartingPosition = 1;
while (     wrdCACHEDT9[StartingPosition-1] && wrdCACHEDT9[StartingPosition-1]==wrdLOW[StartingPosition-1] ) // No need of && wrd[StartingPosition-1]
            StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

            SkipHeuristic=0;
            for(i=StartingPosition;i<=m;i++) { // StartingPosition is in range 1..
                        for (j=1;j<=n;j++) {
                                    if(wrdLOW[i-1] == wrdARG[j-1])
                                                LevenshteinT9[i][j] = LevenshteinT9[i-1][j-1];
                                    else
#if defined(_WIN32ASM_)
                                                LevenshteinT9[i][j] = min_AF(LevenshteinT9[i-1][j]+1, LevenshteinT9[i][j-1]+1, LevenshteinT9[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                                LevenshteinT9[i][j] = MIN(MIN((LevenshteinT9[i-1][j]+1),(LevenshteinT9[i][j-1]+1)),(LevenshteinT9[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                                //{LevenshteinT9[i][j] = MIN(MIN(LevenshteinT9[i-1][j],LevenshteinT9[i][j-1]),LevenshteinT9[i-1][j-1]); --LevenshteinT9[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinT9[i][n] - (m-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT9[i][n] - (m-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                                if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT9[m][n]) {
                                                            // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                            #pragma omp critical
#endif
                                                            {
                                                            fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines9++;
                                                            }
                                                            //if ((DumpedLines & 0xff) == 0xff)
                                                            //          //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                            //          fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                }
            // The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
            memcpy(wrdCACHEDT9, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
// LD ]
}
} //if (EXHAUSTIVE_flag == 1)

} //if (WILDCARD_IP_flag)
                                    }
// Wildcard search ]
                                                wrdlen = 0;
                         }
                         else wrdlen++;
        } // k 'for'
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                                } // if (Exact_flag) {
            }
```

```
// 0th thread
#ifdef Commence_OpenMP
    #pragma omp section
#endif
          {

                                                  if (Exact_flag) {
// WHOLE buffer at once not line-by-line [[[[[[[[ Since r.1-++
                                                            k = WorkAreaLedgeT0;
                                                            while ( k < WorkAreaRedgeT0 ) {
                                                            #ifdef RG7Gulliver
                                                                      //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_0(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT0-k+1, n);
                                                                      FOUNDinPTR = Railgun_Sekireigan_Wolfram_0(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT0-k+1, n);
                                                            #else
                                                                      FOUNDinPTR = Railgun_Quadruplet_7_0(&xgamsCACHE[k], wrdARG, WorkAreaRedgeT0-k+1, n);
                                                            #endif
                                                            // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                            // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                            //if ((long)( FOUNDinPTR - &xgamsCACHE[k] )>=0) {
                                                            if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(&xgamsCACHE[k])) {
                                                                      i = k + (long)( FOUNDinPTR - &xgamsCACHE[k] ); j = i;
                                                                      while (xgamsCACHE[j] != 10) {++j;} // Works both on UNIX(LF) and Windows(CRLF)
                                                                      while (i > k && xgamsCACHE[i-1] != 10) {--i;}
                                                                      k = j+1; // Should "point" to first symbol after the dumped fragment.

                                                                      //fwrite( &xgamsCACHE[i], j - i + 1, 1, fp_outLINE ); DumpedLines1++;
                                                                      if (j - i + 1 <= 168*MaxLineLength) { // fix for 1-++
                                                                      memcpy(wrd, &xgamsCACHE[i], j - i + 1); wrd[j - i + 1]=0; DumpedLines0++;
                                                                      j--; if (wrd[j - i + 1]==10) wrd[j - i + 1]=0;
                                                                      j--; if (wrd[j - i + 1]==13) wrd[j - i + 1]=0;
                                                            // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                            #pragma omp critical
#endif
                                                            {
                                                                      if (Dump_flag)
                                                                      fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]);
                                                                      else
                                                                      fprintf( fp_outLINE, "%s\r\n", wrd);
                                                            }
                                                                      }
                                                            } else k = WorkAreaRedgeT0;
                                                            } // while
// WHOLE buffer at once not line-by-line ]]]]]]]] Since r.1-++
                                                  } else { // if (Exact_flag) {

          //memset (wrdCACHEDT0, 0, MaxLineLength+1+1);
          i = 1;
       //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
          wrdlen = 0;
       for( k = WorkAreaLedgeT0; k <= WorkAreaRedgeT0; k++ )
          {
                    workbyte = xgamsCACHE[k];

                     if( wrdlen < MAXboth) {
                                                            if (CaseSensitiveWildcardMatching_flag == 0)
                                 wrdLOW[ wrdlen ] = KAZE_tolower( workbyte );
                                                            else
                                 wrdLOW[ wrdlen ] = ( workbyte );
                                           wrd[ wrdlen ] = workbyte;
                                 }
                     if (workbyte == 10) {
                                           TotalLines0++;
// Wildcard search [
                     if ( 0 < wrdlen && wrdlen < MAXboth)
                                 {
                                           wrd[ wrdlen ] = 0;
                                           wrdLOW[ wrdlen ] = 0;
```

```
                                        if ( wrd[ wrdlen-1 ] == 13 ) //CR
                                                {--wrdlen; wrd[ wrdlen ] = 0; wrdLOW[ wrdlen ] = 0;}

if (WILDCARD_IP_flag) {
// WILDCARD IP [
                                        WordsChecked0++;
                                        if (Exact_flag) {
                                                //if ((long)( Railgun_Quadruplet_7Gulliver_0(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                #ifdef RG7Gulliver
                                                        //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_0(wrdLOW, wrdARG, wrdlen, n);
                                                        FOUNDinPTR = Railgun_Sekireigan_Wolfram_0(wrdLOW, wrdARG, wrdlen, n);
                                                #else
                                                        FOUNDinPTR = Railgun_Quadruplet_7_0(wrdLOW, wrdARG, wrdlen, n);
                                                #endif
                                                // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                //if ((long)( FOUNDinPTR - wrdLOW )>=0)
                                                if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(wrdLOW))
                                                //if ((long)( Railgun_Quadruplet_7_0(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                        {
                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                #pragma omp critical
#endif
                                                {
                                                        DumpedLines0++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                }
                                                        }
                                        } else {
                                                if (WILDCARD_FAST_flag) {
#if defined(_WildFastKaze_)
                                                if ( WildcardMatch_Iterative_Kaze0(wrdARG, wrdLOW) ) {
#else
                                                if ( IterativeWildcards0(wrdARG, wrdLOW) ) {
#endif
                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                #pragma omp critical
#endif
                                                {
                                                        DumpedLines0++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                }
                                                }
                                                } else {
                                                maskGLOBALlen = n;
                                                nameGLOBALlen0 = wrdlen;
                                                if ( EnhancedMaskTest_OrEmpty_AndNotEmpty_0(wrdARG, 0, wrdLOW, 0) ) { // Caution: Not lowercased as it should!
                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                #pragma omp critical
#endif
                                                {
                                                        DumpedLines0++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                }
                                                }
                                                }
                                        }
// WILDCARD IP ]
} else {

// A simple heuristic #1: Don't enter the nasty loops unless MaximumLevenshteinDistance >= ABS(m-n).
m = wrdlen; // strlen(wrd);
//if (m>MaxLineLength)
//{ printf( "\nKazahana: Incoming xgram exceeding the limit.\n" ); exit( 2 ); }
// Above two commented lines are too severe, changed with next line allowing to search into lines bigger than our needs:
if (m)
if (EXHAUSTIVE_flag == 1) {

// Here we'll walk through the whole length of 'm', ay-yaa.
```

```
// EXHAUSTIVE [[[[[[[[[[[[[[[[[[[[[[[[[[[[
// Here the old fuzzy is replaced with exhaustive one, using BBs of the incoming string (here 'm').
// We need to factorize 'm' down to all 'n+AtMostLevenshteinDistance' long strings/BBs and to search into them ONE-BY-ONE - a gruelling task indeed!
//// Example:
// One of those 33/5 lines is a 46 bytes long string, i.e. m = 46:
// |Edelweiß || [[edelweiss]] || edelweiss flower
// Let's search fuzzily for "edelvais", pretending we don't know the right spelling:
// To find "edelweiss" we need at least Levenshtein distance 3, i.e. n = 8, AtMostLevenshteinDistance = 3:
// 'edelvais' vs 'edelweiss':
// - the mispelled one has one character less;
// - the mispelled one has 2 wrong characters: 'v' & 'a' instead of 'w' & 'e'.
// From above we need Building-Blocks of 46 bytes order 8+3.
// Inhere we are using order 11, 'm - Order + 1' is the number of total BBs for text 'm' bytes long: 46-11+1 = 36
// The 36 BBs:
// |Edelweiß |
// Edelweiß ||
// ...
// weiss flowe
// eiss flower

// Slow! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
/*
                                for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                for (l=0; l < m-BB+1; l++) {
            WordsChecked0++;
// LD [
            SkipHeuristic=0;
            for(i=1;i<=BB;i++) {
                        for (j=1;j<=n;j++) {
                                    if(wrdLOW[l+i-1] == wrdARG[j-1])
                                                LevenshteinT0[i][j] = LevenshteinT0[i-1][j-1];
                                    else
#if defined(_WIN32ASM_)
                                                LevenshteinT0[i][j] = min_AF(LevenshteinT0[i-1][j]+1, LevenshteinT0[i][j-1]+1, LevenshteinT0[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                                LevenshteinT0[i][j] = MIN(MIN((LevenshteinT0[i-1][j]+1),(LevenshteinT0[i][j-1]+1)),(LevenshteinT0[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                                //{LevenshteinT0[i][j] = MIN(MIN(LevenshteinT0[i-1][j],LevenshteinT0[i][j-1]),LevenshteinT0[i-1][j-1]); --LevenshteinT0[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. BB) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinT0[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT0[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                                if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT0[BB][n]) {
                                                            // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                            #pragma omp critical
#endif
                                                            {
                                                            fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines0++;
                                                            }
                                                            goto EXHAUSTIVE0;
                                                            //break; // No need of further checking down the line, one dump only is needed.
                                                            //if ((DumpedLines & 0xff) == 0xff)
                                                            //         //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                            //         fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                }
// LD ]
                                    }
                                    }
EXHAUSTIVE0:;
*/
// Slow! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Fast! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

                                    for (l=0; l < m-(n-AtMostLevenshteinDistance)+1; l++) { // Here BB = n-AtMostLevenshteinDistance the smallest BB
                                    for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
```

```
			// From here on 'm' will be replaced by BB
			if (l < m-BB+1) {

if (AtMostLevenshteinDistance >= MAX(BB,n)-MIN(BB,n)) // This is the only add-on for r.1+
{
	WordsChecked0++;
// LD [
StartingPosition = 1;
while (	wrdCACHEDT0[StartingPosition-1] && wrdLOW[l+StartingPosition-1] && wrdCACHEDT0[StartingPosition-1]==wrdLOW[l+StartingPosition-1] )
	StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

	SkipHeuristic=0;
	for(i=StartingPosition;i<=BB;i++) { // StartingPosition is in range 1..
		for (j=1;j<=n;j++) {
			if(wrdLOW[l+i-1] == wrdARG[j-1])
				LevenshteinT0[i][j] = LevenshteinT0[i-1][j-1];
			else
#if defined(_WIN32ASM_)
				LevenshteinT0[i][j] = min_AF(LevenshteinT0[i-1][j]+1, LevenshteinT0[i][j-1]+1, LevenshteinT0[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinT0[i][j] = MIN(MIN((LevenshteinT0[i-1][j]+1),(LevenshteinT0[i][j-1]+1)),(LevenshteinT0[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinT0[i][j] = MIN(MIN(LevenshteinT0[i-1][j],LevenshteinT0[i][j-1]),LevenshteinT0[i-1][j-1]); --LevenshteinT0[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
		}

		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinT0[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT0[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT0[BB][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines0++;
					}
					// Once dumping the line we need double 'break' from BB and l 'for's:
					goto EXHAUSTIVE0;
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
					//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
				}
	// The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
	//memcpy(wrdCACHEDT0, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
	memcpy(wrdCACHEDT0, &wrdLOW[l], BB); wrdCACHEDT0[BB]=0;
// LD ]
}

			} // if (l < m-BB+1)
			}
			}
EXHAUSTIVE0:;

// Fast! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// EXHAUSTIVE ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]

} else {
if (m<=MaxLineLength)
if (AtMostLevenshteinDistance >= MAX(m,n)-MIN(m,n)) // This is the only add-on for r.1+
{
	WordsChecked0++;
// LD [
StartingPosition = 1;
while (	wrdCACHEDT0[StartingPosition-1] && wrdCACHEDT0[StartingPosition-1]==wrdLOW[StartingPosition-1] ) // No need of && wrd[StartingPosition-1]
	StartingPosition++;
```

```
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

           SkipHeuristic=0;
           for(i=StartingPosition;i<=m;i++) { // StartingPosition is in range 1..
                       for (j=1;j<=n;j++) {
                                  if(wrdLOW[i-1] == wrdARG[j-1])
                                             LevenshteinT0[i][j] = LevenshteinT0[i-1][j-1];
                                  else
#if defined(_WIN32ASM_)
                                             LevenshteinT0[i][j] = min_AF(LevenshteinT0[i-1][j]+1, LevenshteinT0[i][j-1]+1, LevenshteinT0[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                             LevenshteinT0[i][j] = MIN(MIN((LevenshteinT0[i-1][j]+1),(LevenshteinT0[i][j-1]+1)),(LevenshteinT0[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                             //{LevenshteinT0[i][j] = MIN(MIN(LevenshteinT0[i-1][j],LevenshteinT0[i][j-1]),LevenshteinT0[i-1][j-1]); --LevenshteinT0[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                       }

                       // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                       if ( LevenshteinT0[i][n] - (m-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT0[i][n] - (m-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
           }
                                             if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT0[m][n]) {
                                                         // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                         #pragma omp critical
#endif
                                                         {
                                                         fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines0++;
                                                         }
                                                         //if ((DumpedLines & 0xff) == 0xff)
                                                         //          //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                         //          fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                             }
           // The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
           memcpy(wrdCACHEDT0, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
// LD ]
}
} //if (EXHAUSTIVE_flag == 1)

} //if (WILDCARD_IP_flag)
                                  }
// Wildcard search ]
                                             wrdlen = 0;
                        }
                        else wrdlen++;
        } // k 'for'
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                             } // if (Exact_flag) {
           }

// Ath thread
#ifdef Commence_OpenMP
    #pragma omp section
#endif
           {

                                             if (Exact_flag) {
// WHOLE buffer at once not line-by-line [[[[[[[ Since r.1-++
                                                         k = WorkAreaLedgeTa;
                                                         while ( k < WorkAreaRedgeTa ) {
                                                         #ifdef RG7Gulliver
                                                                    //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_a(&xgamsCACHE[k], wrdARG, WorkAreaRedgeTa-k+1, n);
                                                                    FOUNDinPTR = Railgun_Sekireigan_Wolfram_a(&xgamsCACHE[k], wrdARG, WorkAreaRedgeTa-k+1, n);
                                                         #else
                                                                    FOUNDinPTR = Railgun_Quadruplet_7_a(&xgamsCACHE[k], wrdARG, WorkAreaRedgeTa-k+1, n);
                                                         #endif
                                                         // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                         // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                         //if ((long)( FOUNDinPTR - &xgamsCACHE[k] )>=0) {
```

```
                                                        if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(&xgamsCACHE[k])) {
                                                                    i = k + (long)( FOUNDinPTR - &xgamsCACHE[k] ); j = i;
                                                                    while (xgamsCACHE[j] != 10) {++j;} // Works both on UNIX(LF) and Windows(CRLF)
                                                                    while (i > k && xgamsCACHE[i-1] != 10) {--i;}
                                                                    k = j+1; // Should "point" to first symbol after the dumped fragment.

                                                                    //fwrite( &xgamsCACHE[i], j - i + 1, 1, fp_outLINE ); DumpedLines1++;
                                                                    if (j - i + 1 <= 168*MaxLineLength) { // fix for 1-++
                                                                    memcpy(wrd, &xgamsCACHE[i], j - i + 1); wrd[j - i + 1]=0; DumpedLinesa++;
                                                                    j--; if (wrd[j - i + 1]==10) wrd[j - i + 1]=0;
                                                                    j--; if (wrd[j - i + 1]==13) wrd[j - i + 1]=0;
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP

                                                        #pragma omp critical

#endif

                                                        {
                                                                    if (Dump_flag)
                                                                    fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]);
                                                                    else
                                                                    fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                                    }
                                                        } else k = WorkAreaRedgeTa;
                                                        } // while
// WHOLE buffer at once not line-by-line ]]]]]]]] Since r.1-++
                                             } else { // if (Exact_flag) {

           //memset (wrdCACHEDTa, 0, MaxLineLength+1+1);
           i = 1;
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
           wrdlen = 0;
        for( k = WorkAreaLedgeTa; k <= WorkAreaRedgeTa; k++ )
           {
                      workbyte = xgamsCACHE[k];

                        if( wrdlen < MAXboth) {
                                                        if (CaseSensitiveWildcardMatching_flag == 0)
                                   wrdLOW[ wrdlen ] = KAZE_tolower( workbyte );
                                                        else
                                   wrdLOW[ wrdlen ] = ( workbyte );
                                             wrd[ wrdlen ] = workbyte;
                                  }
                        if (workbyte == 10) {
                                             TotalLinesa++;
// Wildcard search [
                        if ( 0 < wrdlen && wrdlen < MAXboth)
                                  {
                                             wrd[ wrdlen ] = 0;
                                             wrdLOW[ wrdlen ] = 0;
                                             if ( wrd[ wrdlen-1 ] == 13 ) //CR
                                                        {--wrdlen; wrd[ wrdlen ] = 0; wrdLOW[ wrdlen ] = 0;}

if (WILDCARD_IP_flag) {
// WILDCARD IP [

                                             WordsCheckeda++;
                                             if (Exact_flag) {
                                                        //if ((long)( Railgun_Quadruplet_7Gulliver_a(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                        #ifdef RG7Gulliver
                                                                    //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_a(wrdLOW, wrdARG, wrdlen, n);
                                                                    FOUNDinPTR = Railgun_Sekireigan_Wolfram_a(wrdLOW, wrdARG, wrdlen, n);
                                                        #else
                                                                    FOUNDinPTR = Railgun_Quadruplet_7_a(wrdLOW, wrdARG, wrdlen, n);
                                                        #endif
                                                        // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                        // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                        //if ((long)( FOUNDinPTR - wrdLOW )>=0)
                                                        if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(wrdLOW))
                                                        //if ((long)( Railgun_Quadruplet_7_a(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                                    {
```

```
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                DumpedLinesa++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                                }
                                                } else {
                                                        if (WILDCARD_FAST_flag) {
#if defined(_WildFastKaze_)
                                                        if ( WildcardMatch_Iterative_Kazea(wrdARG, wrdLOW) ) {
#else
                                                        if ( IterativeWildcardsa(wrdARG, wrdLOW) ) {
#endif
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                DumpedLinesa++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                        }
                                                        } else {
                                                        maskGLOBALlen = n;
                                                        nameGLOBALlena = wrdlen;
                                                        if ( EnhancedMaskTest_OrEmpty_AndNotEmpty_a(wrdARG, 0, wrdLOW, 0) ) { // Caution: Not lowercased as it should!
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                DumpedLinesa++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                        }
                                                        }
                                                }
// WILDCARD IP ]
} else {

// A simple heuristic #1: Don't enter the nasty loops unless MaximumLevenshteinDistance >= ABS(m-n).
m = wrdlen; // strlen(wrd);
//if (m>MaxLineLength)
//{ printf( "\nKazahana: Incoming xgram exceeding the limit.\n" ); exit( 2 ); }
// Above two commented lines are too severe, changed with next line allowing to search into lines bigger than our needs:
if (m)
if (EXHAUSTIVE_flag == 1) {

// Here we'll walk through the whole length of 'm', ay-yaa.
// EXHAUSTIVE [[[[[[[[[[[[[[[[[[[[[[[[[[[[[[
// Here the old fuzzy is replaced with exhaustive one, using BBs of the incoming string (here 'm').
// We need to factorize 'm' down to all 'n+AtMostLevenshteinDistance' long strings/BBs and to search into them ONE-BY-ONE - a gruelling task indeed!
//// Example:
// One of those 33/5 lines is a 46 bytes long string, i.e. m = 46:
// |Edelweiß || [[edelweiss]] || edelweiss flower
// Let's search fuzzily for "edelvais", pretending we don't know the right spelling:
// To find "edelweiss" we need at least Levenshtein distance 3, i.e. n = 8, AtMostLevenshteinDistance = 3:
// 'edelvais' vs 'edelweiss':
// - the mispelled one has one character less;
// - the mispelled one has 2 wrong characters: 'v' & 'a' instead of 'w' & 'e'.
// From above we need Building-Blocks of 46 bytes order 8+3.
// Inhere we are using order 11, 'm - Order + 1' is the number of total BBs for text 'm' bytes long: 46-11+1 = 36
// The 36 BBs:
// |Edelweiß |
// Edelweiß ||
// ...
// weiss flowe
// eiss flower
```

```
// Slow! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
/*
			for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
			for (l=0; l < m-BB+1; l++) {
	WordsCheckeda++;
// LD [
	SkipHeuristic=0;
	for(i=1;i<=BB;i++) {
		for (j=1;j<=n;j++) {
			if(wrdLOW[l+i-1] == wrdARG[j-1])
				LevenshteinTa[i][j] = LevenshteinTa[i-1][j-1];
			else
#if defined(_WIN32ASM_)
				LevenshteinTa[i][j] = min_AF(LevenshteinTa[i-1][j]+1, LevenshteinTa[i][j-1]+1, LevenshteinTa[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinTa[i][j] = MIN(MIN((LevenshteinTa[i-1][j]+1),(LevenshteinTa[i][j-1]+1)),(LevenshteinTa[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinTa[i][j] = MIN(MIN(LevenshteinTa[i-1][j],LevenshteinTa[i][j-1]),LevenshteinTa[i-1][j-1]); --LevenshteinTa[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! // 237,270 xgrams/s
#endif
		}

		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. BB) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinTa[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinTa[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinTa[BB][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLinesa++;
					}
					goto EXHAUSTIVEa;
					//break; // No need of further checking down the line, one dump only is needed.
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
					//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
				}
// LD ]
			}
			}
EXHAUSTIVEa:;
*/
// Slow! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Fast! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

			for (l=0; l < m-(n-AtMostLevenshteinDistance)+1; l++) { // Here BB = n-AtMostLevenshteinDistance the smallest BB
			for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
			// From here on 'm' will be replaced by BB
			if (l < m-BB+1) {

if (AtMostLevenshteinDistance >= MAX(BB,n)-MIN(BB,n)) // This is the only add-on for r.1+
{
	WordsCheckeda++;
// LD [
StartingPosition = 1;
while (	wrdCACHEDTa[StartingPosition-1] && wrdLOW[l+StartingPosition-1] && wrdCACHEDTa[StartingPosition-1]==wrdLOW[l+StartingPosition-1] )
	StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

	SkipHeuristic=0;
	for(i=StartingPosition;i<=BB;i++) { // StartingPosition is in range 1..
		for (j=1;j<=n;j++) {
			if(wrdLOW[l+i-1] == wrdARG[j-1])
				LevenshteinTa[i][j] = LevenshteinTa[i-1][j-1];
			else
#if defined(_WIN32ASM_)
```

```
				LevenshteinTa[i][j] = min_AF(LevenshteinTa[i-1][j]+1, LevenshteinTa[i][j-1]+1, LevenshteinTa[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinTa[i][j] = MIN(MIN((LevenshteinTa[i-1][j]+1),(LevenshteinTa[i][j-1]+1)),(LevenshteinTa[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinTa[i][j] = MIN(MIN(LevenshteinTa[i-1][j],LevenshteinTa[i][j-1]),LevenshteinTa[i-1][j-1]); --LevenshteinTa[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
		}

		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinTa[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinTa[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinTa[BB][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLinesa++;
					}
					// Once dumping the line we need double 'break' from BB and l 'for's:
					goto EXHAUSTIVEa;
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
					//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
				}
	// The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
	//memcpy(wrdCACHEDTa, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
	memcpy(wrdCACHEDTa, &wrdLOW[l], BB); wrdCACHEDTa[BB]=0;
// LD ]
}

			} // if (l < m-BB+1)
			}
			}
EXHAUSTIVEa:;

// Fast! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// EXHAUSTIVE ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]

} else {
if (m<=MaxLineLength)
if (AtMostLevenshteinDistance >= MAX(m,n)-MIN(m,n)) // This is the only add-on for r.1+
{
	WordsCheckeda++;
// LD [
StartingPosition = 1;
while (	wrdCACHEDTa[StartingPosition-1] && wrdCACHEDTa[StartingPosition-1]==wrdLOW[StartingPosition-1] ) // No need of && wrd[StartingPosition-1]
	StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

	SkipHeuristic=0;
	for(i=StartingPosition;i<=m;i++) { // StartingPosition is in range 1..
		for (j=1;j<=n;j++) {
			if(wrdLOW[i-1] == wrdARG[j-1])
				LevenshteinTa[i][j] = LevenshteinTa[i-1][j-1];
			else
#if defined(_WIN32ASM_)
				LevenshteinTa[i][j] = min_AF(LevenshteinTa[i-1][j]+1, LevenshteinTa[i][j-1]+1, LevenshteinTa[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinTa[i][j] = MIN(MIN((LevenshteinTa[i-1][j]+1),(LevenshteinTa[i][j-1]+1)),(LevenshteinTa[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinTa[i][j] = MIN(MIN(LevenshteinTa[i-1][j],LevenshteinTa[i][j-1]),LevenshteinTa[i-1][j-1]); --LevenshteinTa[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
		}

		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinTa[i][n] - (m-i) >= 0 && AtMostLevenshteinDistance < LevenshteinTa[i][n] - (m-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
```

```
             }
                                                   if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinTa[m][n]) {
                                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                                #pragma omp critical
#endif
                                                                {
                                                                fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLinesa++;
                                                                }
                                                                //if ((DumpedLines & 0xff) == 0xff)
                                                                //           //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                                //           fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                   }
             // The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
             memcpy(wrdCACHEDTa, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
// LD ]
}
} //if (EXHAUSTIVE_flag == 1)

} //if (WILDCARD_IP_flag)
                                       }
// Wildcard search ]
                                                   wrdlen = 0;
                           }
                           else wrdlen++;
         } // k 'for'
         //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                                   } // if (Exact_flag) {
             }

// Bth thread
#ifdef Commence_OpenMP
    #pragma omp section
#endif
             {

                                                   if (Exact_flag) {
// WHOLE buffer at once not line-by-line [[[[[[[ Since r.1-++
                                                                k = WorkAreaLedgeTb;
                                                                while ( k < WorkAreaRedgeTb ) {
                                                                #ifdef RG7Gulliver
                                                                             //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_b(&xgamsCACHE[k], wrdARG, WorkAreaRedgeTb-k+1, n);
                                                                             FOUNDinPTR = Railgun_Sekireigan_Wolfram_b(&xgamsCACHE[k], wrdARG, WorkAreaRedgeTb-k+1, n);
                                                                #else
                                                                             FOUNDinPTR = Railgun_Quadruplet_7_b(&xgamsCACHE[k], wrdARG, WorkAreaRedgeTb-k+1, n);
                                                                #endif
                                                                // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                                // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                                //if ((long)( FOUNDinPTR - &xgamsCACHE[k] )>=0) {
                                                                if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(&xgamsCACHE[k])) {
                                                                             i = k + (long)( FOUNDinPTR - &xgamsCACHE[k] ); j = i;
                                                                             while (xgamsCACHE[j] != 10) {++j;} // Works both on UNIX(LF) and Windows(CRLF)
                                                                             while (i > k && xgamsCACHE[i-1] != 10) {--i;}
                                                                             k = j+1; // Should "point" to first symbol after the dumped fragment.

                                                                             //fwrite( &xgamsCACHE[i], j - i + 1, 1, fp_outLINE ); DumpedLines1++;
                                                                             if (j - i + 1 <= 168*MaxLineLength) { // fix for 1-++
                                                                             memcpy(wrd, &xgamsCACHE[i], j - i + 1); wrd[j - i + 1]=0; DumpedLinesb++;
                                                                             j--; if (wrd[j - i + 1]==10) wrd[j - i + 1]=0;
                                                                             j--; if (wrd[j - i + 1]==13) wrd[j - i + 1]=0;
                                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                                #pragma omp critical
#endif
                                                                {
                                                                             if (Dump_flag)
                                                                             fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]);
                                                                             else
                                                                             fprintf( fp_outLINE, "%s\r\n", wrd);
```

```
                                                        }
                                                                }
                                                        } else k = WorkAreaRedgeTb;
                                                        } // while
// WHOLE buffer at once not line-by-line ]]]]]]]] Since r.1-++
                                                } else { // if (Exact_flag) {

           //memset (wrdCACHEDTb, 0, MaxLineLength+1+1);
           i = 1;
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
           wrdlen = 0;
        for( k = WorkAreaLedgeTb; k <= WorkAreaRedgeTb; k++ )
           {
                        workbyte = xgamsCACHE[k];

                         if( wrdlen < MAXboth) {
                                                        if (CaseSensitiveWildcardMatching_flag == 0)
                                         wrdLOW[ wrdlen ] = KAZE_tolower( workbyte );
                                                        else
                                         wrdLOW[ wrdlen ] = ( workbyte );
                                                  wrd[ wrdlen ] = workbyte;
                                        }
                         if (workbyte == 10) {
                                                  TotalLinesb++;
// Wildcard search [
                         if ( 0 < wrdlen && wrdlen < MAXboth)
                                        {
                                                  wrd[ wrdlen ] = 0;
                                                  wrdLOW[ wrdlen ] = 0;
                                                  if ( wrd[ wrdlen-1 ] == 13 ) //CR
                                                        {--wrdlen; wrd[ wrdlen ] = 0; wrdLOW[ wrdlen ] = 0;}

if (WILDCARD_IP_flag) {
// WILDCARD IP [
                                                  WordsCheckedb++;
                                                  if (Exact_flag) {
                                                        //if ((long)( Railgun_Quadruplet_7Gulliver_b(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                        #ifdef RG7Gulliver
                                                                //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_b(wrdLOW, wrdARG, wrdlen, n);
                                                                FOUNDinPTR = Railgun_Sekireigan_Wolfram_b(wrdLOW, wrdARG, wrdlen, n);
                                                        #else
                                                                FOUNDinPTR = Railgun_Quadruplet_7_b(wrdLOW, wrdARG, wrdlen, n);
                                                        #endif
                                                        // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                        // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                        //if ((long)( FOUNDinPTR - wrdLOW )>=0)
                                                        if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(wrdLOW))
                                                        //if ((long)( Railgun_Quadruplet_7_b(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                                {
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                DumpedLinesb++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                                }
                                                  } else {
                                                        if (WILDCARD_FAST_flag) {
#if defined(_WildFastKaze_)
                                                        if ( WildcardMatch_Iterative_Kazeb(wrdARG, wrdLOW) ) {
#else
                                                        if ( IterativeWildcardsb(wrdARG, wrdLOW) ) {
#endif
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
```

```
						DumpedLinesb++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
					}
					}
					} else {
					maskGLOBALlen = n;
					nameGLOBALlenb = wrdlen;
					if ( EnhancedMaskTest_OrEmpty_AndNotEmpty_b(wrdARG, 0, wrdLOW, 0) ) { // Caution: Not lowercased as it should!
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
						DumpedLinesb++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
					}
					}
					}
				}
// WILDCARD IP ]
} else {

// A simple heuristic #1: Don't enter the nasty loops unless MaximumLevenshteinDistance >= ABS(m-n).
m = wrdlen; // strlen(wrd);
//if (m>MaxLineLength)
//{ printf( "\nKazahana: Incoming xgram exceeding the limit.\n" ); exit( 2 ); }
// Above two commented lines are too severe, changed with next line allowing to search into lines bigger than our needs:
if (m)
if (EXHAUSTIVE_flag == 1) {

// Here we'll walk through the whole length of 'm', ay-yaa.
// EXHAUSTIVE [[[[[[[[[[[[[[[[[[[[[[[[[[[[[[
// Here the old fuzzy is replaced with exhaustive one, using BBs of the incoming string (here 'm').
// We need to factorize 'm' down to all 'n+AtMostLevenshteinDistance' long strings/BBs and to search into them ONE-BY-ONE - a gruelling task indeed!
//// Example:
// One of those 33/5 lines is a 46 bytes long string, i.e. m = 46:
// |Edelweiß || [[edelweiss]] || edelweiss flower
// Let's search fuzzily for "edelvais", pretending we don't know the right spelling:
// To find "edelweiss" we need at least Levenshtein distance 3, i.e. n = 8, AtMostLevenshteinDistance = 3:
// 'edelvais' vs 'edelweiss':
// - the mispelled one has one character less;
// - the mispelled one has 2 wrong characters: 'v' & 'a' instead of 'w' & 'e'.
// From above we need Building-Blocks of 46 bytes order 8+3.
// Inhere we are using order 11, 'm - Order + 1' is the number of total BBs for text 'm' bytes long: 46-11+1 = 36
// The 36 BBs:
// |Edelweiß |
// Edelweiß ||
// ...
// weiss flowe
// eiss flower

// Slow! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
/*
			for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
			for (l=0; l < m-BB+1; l++) {
	WordsCheckedb++;
// LD [
	SkipHeuristic=0;
	for(i=1;i<=BB;i++) {
		for (j=1;j<=n;j++) {
			if(wrdLOW[l+i-1] == wrdARG[j-1])
				LevenshteinTb[i][j] = LevenshteinTb[i-1][j-1];
			else
#if defined(_WIN32ASM_)
				LevenshteinTb[i][j] = min_AF(LevenshteinTb[i-1][j]+1, LevenshteinTb[i][j-1]+1, LevenshteinTb[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinTb[i][j] = MIN(MIN((LevenshteinTb[i-1][j]+1),(LevenshteinTb[i][j-1]+1)),(LevenshteinTb[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinTb[i][j] = MIN(MIN(LevenshteinTb[i-1][j],LevenshteinTb[i][j-1]),LevenshteinTb[i-1][j-1]); --LevenshteinTb[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
		}
```

```
                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. BB) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinTb[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinTb[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                                    if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinTb[BB][n]) {
                                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                                #pragma omp critical
#endif
                                                                {
                                                                fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLinesb++;
                                                                }
                                                                goto EXHAUSTIVEb;
                                                                //break; // No need of further checking down the line, one dump only is needed.
                                                                //if ((DumpedLines & 0xff) == 0xff)
                                                                //         //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                                //         fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                    }
// LD ]
                                        }
                                        }
EXHAUSTIVEb:;
*/
// Slow! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Fast! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

                                        for (l=0; l < m-(n-AtMostLevenshteinDistance)+1; l++) { // Here BB = n-AtMostLevenshteinDistance the smallest BB
                                        for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                        // From here on 'm' will be replaced by BB
                                        if (l < m-BB+1) {

if (AtMostLevenshteinDistance >= MAX(BB,n)-MIN(BB,n)) // This is the only add-on for r.1+
{
            WordsCheckedb++;
// LD [
StartingPosition = 1;
while (     wrdCACHEDTb[StartingPosition-1] && wrdLOW[l+StartingPosition-1] && wrdCACHEDTb[StartingPosition-1]==wrdLOW[l+StartingPosition-1] )
            StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

            SkipHeuristic=0;
            for(i=StartingPosition;i<=BB;i++) { // StartingPosition is in range 1..
                        for (j=1;j<=n;j++) {
                                    if(wrdLOW[l+i-1] == wrdARG[j-1])
                                                LevenshteinTb[i][j] = LevenshteinTb[i-1][j-1];
                                    else
#if defined(_WIN32ASM_)
                                                LevenshteinTb[i][j] = min_AF(LevenshteinTb[i-1][j]+1, LevenshteinTb[i][j-1]+1, LevenshteinTb[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                                LevenshteinTb[i][j] = MIN(MIN((LevenshteinTb[i-1][j]+1),(LevenshteinTb[i][j-1]+1)),(LevenshteinTb[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                                //{LevenshteinTb[i][j] = MIN(MIN(LevenshteinTb[i-1][j],LevenshteinTb[i][j-1]),LevenshteinTb[i-1][j-1]); --LevenshteinTb[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinTb[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinTb[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                                    if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinTb[BB][n]) {
                                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                                #pragma omp critical
#endif
                                                                {
                                                                fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLinesb++;
                                                                }
                                                                // Once dumping the line we need double 'break' from BB and l 'for's:
```

```
					goto EXHAUSTIVEb;
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
					//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
				}
	// The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
	//memcpy(wrdCACHEDTb, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
	memcpy(wrdCACHEDTb, &wrdLOW[l], BB); wrdCACHEDTb[BB]=0;
// LD ]
}

			} // if (l < m-BB+1)
			}
			}
EXHAUSTIVEb:;

// Fast! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// EXHAUSTIVE ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]

} else {
if (m<=MaxLineLength)
if (AtMostLevenshteinDistance >= MAX(m,n)-MIN(m,n)) // This is the only add-on for r.1+
{
	WordsCheckedb++;
// LD [
StartingPosition = 1;
while (	wrdCACHEDTb[StartingPosition-1] && wrdCACHEDTb[StartingPosition-1]==wrdLOW[StartingPosition-1] ) // No need of && wrd[StartingPosition-1]
	StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

	SkipHeuristic=0;
	for(i=StartingPosition;i<=m;i++) { // StartingPosition is in range 1..
		for (j=1;j<=n;j++) {
			if(wrdLOW[i-1] == wrdARG[j-1])
				LevenshteinTb[i][j] = LevenshteinTb[i-1][j-1];
			else
#if defined(_WIN32ASM_)
				LevenshteinTb[i][j] = min_AF(LevenshteinTb[i-1][j]+1, LevenshteinTb[i][j-1]+1, LevenshteinTb[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinTb[i][j] = MIN(MIN((LevenshteinTb[i-1][j]+1),(LevenshteinTb[i][j-1]+1)),(LevenshteinTb[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinTb[i][j] = MIN(MIN(LevenshteinTb[i-1][j],LevenshteinTb[i][j-1]),LevenshteinTb[i-1][j-1]); --LevenshteinTb[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! // 237,270 xgrams/s
#endif
		}

		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinTb[i][n] - (m-i) >= 0 && AtMostLevenshteinDistance < LevenshteinTb[i][n] - (m-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinTb[m][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLinesb++;
					}
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
					//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
				}
	// The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
	memcpy(wrdCACHEDTb, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
// LD ]
}
} //if (EXHAUSTIVE_flag == 1)

} //if (WILDCARD_IP_flag)
```

```
                                       }
// Wildcard search ]
                                                    wrdlen = 0;
                           }
                           else wrdlen++;
         } // k 'for'
         //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                                    } // if (Exact_flag) {
             }

// Cth thread
#ifdef Commence_OpenMP
    #pragma omp section
#endif
             {

                                                    if (Exact_flag) {
// WHOLE buffer at once not line-by-line [[[[[[[[ Since r.1-++
                                                                 k = WorkAreaLedgeTc;
                                                                 while ( k < WorkAreaRedgeTc ) {
                                                                 #ifdef RG7Gulliver
                                                                              //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_c(&xgamsCACHE[k], wrdARG, WorkAreaRedgeTc-k+1, n);
                                                                              FOUNDinPTR = Railgun_Sekireigan_Wolfram_c(&xgamsCACHE[k], wrdARG, WorkAreaRedgeTc-k+1, n);
                                                                 #else
                                                                              FOUNDinPTR = Railgun_Quadruplet_7_c(&xgamsCACHE[k], wrdARG, WorkAreaRedgeTc-k+1, n);
                                                                 #endif
                                                                 // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                                 // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                                 //if ((long)( FOUNDinPTR - &xgamsCACHE[k] )>=0) {
                                                                 if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(&xgamsCACHE[k])) {
                                                                              i = k + (long)( FOUNDinPTR - &xgamsCACHE[k] ); j = i;
                                                                              while (xgamsCACHE[j] != 10) {++j;} // Works both on UNIX(LF) and Windows(CRLF)
                                                                              while (i > k && xgamsCACHE[i-1] != 10) {--i;}
                                                                              k = j+1; // Should "point" to first symbol after the dumped fragment.

                                                                              //fwrite( &xgamsCACHE[i], j - i + 1, 1, fp_outLINE ); DumpedLines1++;
                                                                              if (j - i + 1 <= 168*MaxLineLength) { // fix for 1-++
                                                                              memcpy(wrd, &xgamsCACHE[i], j - i + 1); wrd[j - i + 1]=0; DumpedLinesc++;
                                                                              j--; if (wrd[j - i + 1]==10) wrd[j - i + 1]=0;
                                                                              j--; if (wrd[j - i + 1]==13) wrd[j - i + 1]=0;
                                                                 // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                                 #pragma omp critical
#endif
                                                                 {
                                                                              if (Dump_flag)
                                                                              fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]);
                                                                              else
                                                                              fprintf( fp_outLINE, "%s\r\n", wrd);
                                                                 }
                                                                              }
                                                                 } else k = WorkAreaRedgeTc;
                                                                 } // while
// WHOLE buffer at once not line-by-line ]]]]]]]] Since r.1-++
                                                    } else { // if (Exact_flag) {

             //memset (wrdCACHEDTc, 0, MaxLineLength+1+1);
             i = 1;
         //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
             wrdlen = 0;
         for( k = WorkAreaLedgeTc; k <= WorkAreaRedgeTc; k++ )
             {
                          workbyte = xgamsCACHE[k];

                           if( wrdlen < MAXboth) {
                                                                 if (CaseSensitiveWildcardMatching_flag == 0)
                                        wrdLOW[ wrdlen ] = KAZE_tolower( workbyte );
                                                                 else
                                        wrdLOW[ wrdlen ] = ( workbyte );
```

```
                                        wrd[ wrdlen ] = workbyte;
                                }
                        if (workbyte == 10) {
                                        TotalLinesc++;
// Wildcard search [
                        if ( 0 < wrdlen && wrdlen < MAXboth)
                                {
                                        wrd[ wrdlen ] = 0;
                                        wrdLOW[ wrdlen ] = 0;
                                        if ( wrd[ wrdlen-1 ] == 13 ) //CR
                                                {--wrdlen; wrd[ wrdlen ] = 0; wrdLOW[ wrdlen ] = 0;}

if (WILDCARD_IP_flag) {
// WILDCARD IP [
                                        WordsCheckedc++;
                                        if (Exact_flag) {
                                                //if ((long)( Railgun_Quadruplet_7Gulliver_c(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                #ifdef RG7Gulliver
                                                        //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_c(wrdLOW, wrdARG, wrdlen, n);
                                                        FOUNDinPTR = Railgun_Sekireigan_Wolfram_c(wrdLOW, wrdARG, wrdlen, n);
                                                #else
                                                        FOUNDinPTR = Railgun_Quadruplet_7_c(wrdLOW, wrdARG, wrdlen, n);
                                                #endif
                                                // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                //if ((long)( FOUNDinPTR - wrdLOW )>=0)
                                                if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(wrdLOW))
                                                //if ((long)( Railgun_Quadruplet_7_c(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                        {
                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                #pragma omp critical
#endif
                                                {
                                                        DumpedLinesc++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                }
                                                        }
                                        } else {
                                                if (WILDCARD_FAST_flag) {
#if defined(_WildFastKaze_)
                                                if ( WildcardMatch_Iterative_Kazec(wrdARG, wrdLOW) ) {
#else
                                                if ( IterativeWildcardsc(wrdARG, wrdLOW) ) {
#endif
                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                #pragma omp critical
#endif
                                                {
                                                        DumpedLinesc++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                }
                                                }
                                                } else {
                                                maskGLOBALlen = n;
                                                nameGLOBALlenc = wrdlen;
                                                if ( EnhancedMaskTest_OrEmpty_AndNotEmpty_c(wrdARG, 0, wrdLOW, 0) ) { // Caution: Not lowercased as it should!
                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                #pragma omp critical
#endif
                                                {
                                                        DumpedLinesc++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                }
                                                }
                                                }
                                        }
// WILDCARD IP ]
} else {
```

```
// A simple heuristic #1: Don't enter the nasty loops unless MaximumLevenshteinDistance >= ABS(m-n).
m = wrdlen; // strlen(wrd);
//if (m>MaxLineLength)
//{ printf( "\nKazahana: Incoming xgram exceeding the limit.\n" ); exit( 2 ); }
// Above two commented lines are too severe, changed with next line allowing to search into lines bigger than our needs:
if (m)
if (EXHAUSTIVE_flag == 1) {

// Here we'll walk through the whole length of 'm', ay-yaa.
// EXHAUSTIVE [[[[[[[[[[[[[[[[[[[[[[[[[[[[
// Here the old fuzzy is replaced with exhaustive one, using BBs of the incoming string (here 'm').
// We need to factorize 'm' down to all 'n+AtMostLevenshteinDistance' long strings/BBs and to search into them ONE-BY-ONE - a gruelling task indeed!
//// Example:
// One of those 33/5 lines is a 46 bytes long string, i.e. m = 46:
// |Edelweiß || [[edelweiss]] || edelweiss flower
// Let's search fuzzily for "edelvais", pretending we don't know the right spelling:
// To find "edelweiss" we need at least Levenshtein distance 3, i.e. n = 8, AtMostLevenshteinDistance = 3:
// 'edelvais' vs 'edelweiss':
// - the mispelled one has one character less;
// - the mispelled one has 2 wrong characters: 'v' & 'a' instead of 'w' & 'e'.
// From above we need Building-Blocks of 46 bytes order 8+3.
// Inhere we are using order 11, 'm - Order + 1' is the number of total BBs for text 'm' bytes long: 46-11+1 = 36
// The 36 BBs:
// |Edelweiß |
// Edelweiß ||
// ...
// weiss flowe
// eiss flower

// Slow! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
/*
                                for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                for (l=0; l < m-BB+1; l++) {
          WordsCheckedc++;
// LD [
          SkipHeuristic=0;
          for(i=1;i<=BB;i++) {
                    for (j=1;j<=n;j++) {
                                if(wrdLOW[l+i-1] == wrdARG[j-1])
                                          LevenshteinTc[i][j] = LevenshteinTc[i-1][j-1];
                                else
#if defined(_WIN32ASM_)
                                          LevenshteinTc[i][j] = min_AF(LevenshteinTc[i-1][j]+1, LevenshteinTc[i][j-1]+1, LevenshteinTc[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                          LevenshteinTc[i][j] = MIN(MIN((LevenshteinTc[i-1][j]+1),(LevenshteinTc[i][j-1]+1)),(LevenshteinTc[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                          //{LevenshteinTc[i][j] = MIN(MIN(LevenshteinTc[i-1][j],LevenshteinTc[i][j-1]),LevenshteinTc[i-1][j-1]); --LevenshteinTc[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                    }

                    // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. BB) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                    if ( LevenshteinTc[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinTc[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
          }
                                          if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinTc[BB][n]) {
                                                    // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                    #pragma omp critical
#endif
                                                    {
                                                    fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLinesc++;
                                                    }
                                                    goto EXHAUSTIVEc;
                                                    //break; // No need of further checking down the line, one dump only is needed.
                                                    //if ((DumpedLines & 0xff) == 0xff)
                                                    //         //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                    //         fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                          }
// LD ]
                                }
```

```
                        }
EXHAUSTIVEc:;
*/
// Slow! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Fast! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

                        for (l=0; l < m-(n-AtMostLevenshteinDistance)+1; l++) { // Here BB = n-AtMostLevenshteinDistance the smallest BB
                        for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                        // From here on 'm' will be replaced by BB
                        if (l < m-BB+1) {

if (AtMostLevenshteinDistance >= MAX(BB,n)-MIN(BB,n)) // This is the only add-on for r.1+
{
        WordsCheckedc++;
// LD [
StartingPosition = 1;
while ( wrdCACHEDTc[StartingPosition-1] && wrdLOW[l+StartingPosition-1] && wrdCACHEDTc[StartingPosition-1]==wrdLOW[l+StartingPosition-1] )
        StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

        SkipHeuristic=0;
        for(i=StartingPosition;i<=BB;i++) { // StartingPosition is in range 1..
                for (j=1;j<=n;j++) {
                        if(wrdLOW[l+i-1] == wrdARG[j-1])
                                LevenshteinTc[i][j] = LevenshteinTc[i-1][j-1];
                        else
#if defined(_WIN32ASM_)
                                LevenshteinTc[i][j] = min_AF(LevenshteinTc[i-1][j]+1, LevenshteinTc[i][j-1]+1, LevenshteinTc[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                LevenshteinTc[i][j] = MIN(MIN((LevenshteinTc[i-1][j]+1),(LevenshteinTc[i][j-1]+1)),(LevenshteinTc[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                //{LevenshteinTc[i][j] = MIN(MIN(LevenshteinTc[i-1][j],LevenshteinTc[i][j-1]),LevenshteinTc[i-1][j-1]); --LevenshteinTc[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! // 237,270 xgrams/s
#endif
                }

                // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                if ( LevenshteinTc[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinTc[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
        }
                                if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinTc[BB][n]) {
                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                        #pragma omp critical
#endif
                                        {
                                        fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLinesc++;
                                        }
                                        // Once dumping the line we need double 'break' from BB and l 'for's:
                                        goto EXHAUSTIVEc;
                                        //if ((DumpedLines & 0xff) == 0xff)
                                        //      //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                        //      fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                }
        // The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
        //memcpy(wrdCACHEDTc, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
        memcpy(wrdCACHEDTc, &wrdLOW[l], BB); wrdCACHEDTc[BB]=0;
// LD ]
}

                        } // if (l < m-BB+1)
                        }
                        }
EXHAUSTIVEc:;

// Fast! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// EXHAUSTIVE ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]
```

```
} else {
if (m<=MaxLineLength)
if (AtMostLevenshteinDistance >= MAX(m,n)-MIN(m,n)) // This is the only add-on for r.1+
{
	WordsCheckedc++;
// LD [
StartingPosition = 1;
while (	wrdCACHEDTc[StartingPosition-1] && wrdCACHEDTc[StartingPosition-1]==wrdLOW[StartingPosition-1] ) // No need of && wrd[StartingPosition-1]
	StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

	SkipHeuristic=0;
	for(i=StartingPosition;i<=m;i++) { // StartingPosition is in range 1..
		for (j=1;j<=n;j++) {
			if(wrdLOW[i-1] == wrdARG[j-1])
				LevenshteinTc[i][j] = LevenshteinTc[i-1][j-1];
			else
#if defined(_WIN32ASM_)
				LevenshteinTc[i][j] = min_AF(LevenshteinTc[i-1][j]+1, LevenshteinTc[i][j-1]+1, LevenshteinTc[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinTc[i][j] = MIN(MIN((LevenshteinTc[i-1][j]+1),(LevenshteinTc[i][j-1]+1)),(LevenshteinTc[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinTc[i][j] = MIN(MIN(LevenshteinTc[i-1][j],LevenshteinTc[i][j-1]),LevenshteinTc[i-1][j-1]); --LevenshteinTc[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
		}

		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinTc[i][n] - (m-i) >= 0 && AtMostLevenshteinDistance < LevenshteinTc[i][n] - (m-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinTc[m][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLinesc++;
					}
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
					//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
				}
	// The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
	memcpy(wrdCACHEDTc, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
// LD ]
}
} //if (EXHAUSTIVE_flag == 1)

} //if (WILDCARD_IP_flag)
			}
// Wildcard search ]
				wrdlen = 0;
                        }
                        else wrdlen++;
        } // k 'for'
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
				} // if (Exact_flag) {
	}

// Dth thread
#ifdef Commence_OpenMP
    #pragma omp section
#endif
	{

				if (Exact_flag) {
// WHOLE buffer at once not line-by-line [[[[[[[[ Since r.1-++
					k = WorkAreaLedgeTd;
					while ( k < WorkAreaRedgeTd ) {
```

```
                                                        #ifdef RG7Gulliver
                                                                   //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_d(&xgamsCACHE[k], wrdARG, WorkAreaRedgeTd-k+1, n);
                                                                   FOUNDinPTR = Railgun_Sekireigan_Wolfram_d(&xgamsCACHE[k], wrdARG, WorkAreaRedgeTd-k+1, n);
                                                        #else
                                                                   FOUNDinPTR = Railgun_Quadruplet_7_d(&xgamsCACHE[k], wrdARG, WorkAreaRedgeTd-k+1, n);
                                                        #endif
                                                        // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                        // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                        //if ((long)( FOUNDinPTR - &xgamsCACHE[k] )>=0) {
                                                        if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(&xgamsCACHE[k])) {
                                                                   i = k + (long)( FOUNDinPTR - &xgamsCACHE[k] ); j = i;
                                                                   while (xgamsCACHE[j] != 10) {++j;} // Works both on UNIX(LF) and Windows(CRLF)
                                                                   while (i > k && xgamsCACHE[i-1] != 10) {--i;}
                                                                   k = j+1; // Should "point" to first symbol after the dumped fragment.

                                                                   //fwrite( &xgamsCACHE[i], j - i + 1, 1, fp_outLINE ); DumpedLines1++;
                                                                   if (j - i + 1 <= 168*MaxLineLength) { // fix for 1-++
                                                                   memcpy(wrd, &xgamsCACHE[i], j - i + 1); wrd[j - i + 1]=0; DumpedLinesd++;
                                                                   j--; if (wrd[j - i + 1]==10) wrd[j - i + 1]=0;
                                                                   j--; if (wrd[j - i + 1]==13) wrd[j - i + 1]=0;
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP

                                                        #pragma omp critical

#endif
                                                        {
                                                                   if (Dump_flag)
                                                                   fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]);
                                                                   else
                                                                   fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                                   }
                                                        } else k = WorkAreaRedgeTd;
                                                        } // while
// WHOLE buffer at once not line-by-line ]]]]]]]] Since r.1-++
                                             } else { // if (Exact_flag) {

           //memset (wrdCACHEDTd, 0, MaxLineLength+1+1);
           i = 1;
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
           wrdlen = 0;
        for( k = WorkAreaLedgeTd; k <= WorkAreaRedgeTd; k++ )
           {
                       workbyte = xgamsCACHE[k];

                        if( wrdlen < MAXboth) {
                                                        if (CaseSensitiveWildcardMatching_flag == 0)
                                   wrdLOW[ wrdlen ] = KAZE_tolower( workbyte );
                                                        else
                                   wrdLOW[ wrdlen ] = ( workbyte );
                                             wrd[ wrdlen ] = workbyte;
                                  }
                        if (workbyte == 10) {
                                             TotalLinesd++;
// Wildcard search [
                        if ( 0 < wrdlen && wrdlen < MAXboth)
                                  {
                                             wrd[ wrdlen ] = 0;
                                             wrdLOW[ wrdlen ] = 0;
                                             if ( wrd[ wrdlen-1 ] == 13 ) //CR
                                                        {--wrdlen; wrd[ wrdlen ] = 0; wrdLOW[ wrdlen ] = 0;}

if (WILDCARD_IP_flag) {
// WILDCARD IP [
                                             WordsCheckedd++;
                                             if (Exact_flag) {
                                                        //if ((long)( Railgun_Quadruplet_7Gulliver_d(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                        #ifdef RG7Gulliver
                                                                   //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_d(wrdLOW, wrdARG, wrdlen, n);
                                                                   FOUNDinPTR = Railgun_Sekireigan_Wolfram_d(wrdLOW, wrdARG, wrdlen, n);
```

```
                                                        #else
                                                                FOUNDinPTR = Railgun_Quadruplet_7_d(wrdLOW, wrdARG, wrdlen, n);
                                                        #endif
                                                        // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                        // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                        //if ((long)( FOUNDinPTR - wrdLOW )>=0)
                                                        if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(wrdLOW))
                                                        //if ((long)( Railgun_Quadruplet_7_d(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                                {
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                DumpedLinesd++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                                }
                                                } else {
                                                        if (WILDCARD_FAST_flag) {
#if defined(_WildFastKaze_)
                                                        if ( WildcardMatch_Iterative_Kazed(wrdARG, wrdLOW) ) {
#else
                                                        if ( IterativeWildcardsd(wrdARG, wrdLOW) ) {
#endif
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                DumpedLinesd++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                        }
                                                        } else {
                                                        maskGLOBALlen = n;
                                                        nameGLOBALlend = wrdlen;
                                                        if ( EnhancedMaskTest_OrEmpty_AndNotEmpty_d(wrdARG, 0, wrdLOW, 0) ) { // Caution: Not lowercased as it should!
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                DumpedLinesd++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                        }
                                                        }
                                                }
// WILDCARD IP ]
} else {

// A simple heuristic #1: Don't enter the nasty loops unless MaximumLevenshteinDistance >= ABS(m-n).
m = wrdlen; // strlen(wrd);
//if (m>MaxLineLength)
//{ printf( "\nKazahana: Incoming xgram exceeding the limit.\n" ); exit( 2 ); }
// Above two commented lines are too severe, changed with next line allowing to search into lines bigger than our needs:
if (m)
if (EXHAUSTIVE_flag == 1) {

// Here we'll walk through the whole length of 'm', ay-yaa.
// EXHAUSTIVE [[[[[[[[[[[[[[[[[[[[[[[[[[[[
// Here the old fuzzy is replaced with exhaustive one, using BBs of the incoming string (here 'm').
// We need to factorize 'm' down to all 'n+AtMostLevenshteinDistance' long strings/BBs and to search into them ONE-BY-ONE - a gruelling task indeed!
//// Example:
// One of those 33/5 lines is a 46 bytes long string, i.e. m = 46:
// |Edelweiß || [[edelweiss]] || edelweiss flower
// Let's search fuzzily for "edelvais", pretending we don't know the right spelling:
// To find "edelweiss" we need at least Levenshtein distance 3, i.e. n = 8, AtMostLevenshteinDistance = 3:
// 'edelvais' vs 'edelweiss':
// - the mispelled one has one character less;
// - the mispelled one has 2 wrong characters: 'v' & 'a' instead of 'w' & 'e'.
```

```
// From above we need Building-Blocks of 46 bytes order 8+3.
// Inhere we are using order 11, 'm - Order + 1' is the number of total BBs for text 'm' bytes long: 46-11+1 = 36
// The 36 BBs:
// |Edelweiß |
// Edelweiß ||
// ...
// weiss flowe
// eiss flower

// Slow! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
/*
                                        for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                        for (l=0; l < m-BB+1; l++) {
                    WordsCheckedd++;
// LD [
                    SkipHeuristic=0;
                    for(i=1;i<=BB;i++) {
                                for (j=1;j<=n;j++) {
                                        if(wrdLOW[l+i-1] == wrdARG[j-1])
                                                LevenshteinTd[i][j] = LevenshteinTd[i-1][j-1];
                                        else
#if defined(_WIN32ASM_)
                                                LevenshteinTd[i][j] = min_AF(LevenshteinTd[i-1][j]+1, LevenshteinTd[i][j-1]+1, LevenshteinTd[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                                LevenshteinTd[i][j] = MIN(MIN((LevenshteinTd[i-1][j]+1),(LevenshteinTd[i][j-1]+1)),(LevenshteinTd[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                                //{LevenshteinTd[i][j] = MIN(MIN(LevenshteinTd[i-1][j],LevenshteinTd[i][j-1]),LevenshteinTd[i-1][j-1]); --LevenshteinTd[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                                }

                                // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. BB) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                                if ( LevenshteinTd[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinTd[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
                    }
                                                if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinTd[BB][n]) {
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                        fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLinesd++;
                                                        }
                                                        goto EXHAUSTIVEd;
                                                        //break; // No need of further checking down the line, one dump only is needed.
                                                        //if ((DumpedLines & 0xff) == 0xff)
                                                        //        //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                        //        fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                }
// LD ]
                                        }
                                        }
EXHAUSTIVEd:;
*/
// Slow! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Fast! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

                                        for (l=0; l < m-(n-AtMostLevenshteinDistance)+1; l++) { // Here BB = n-AtMostLevenshteinDistance the smallest BB
                                        for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                        // From here on 'm' will be replaced by BB
                                        if (l < m-BB+1) {

if (AtMostLevenshteinDistance >= MAX(BB,n)-MIN(BB,n)) // This is the only add-on for r.1+
{
                    WordsCheckedd++;
// LD [
StartingPosition = 1;
while (     wrdCACHEDTd[StartingPosition-1] && wrdLOW[l+StartingPosition-1] && wrdCACHEDTd[StartingPosition-1]==wrdLOW[l+StartingPosition-1] )
                    StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
```

```
StartingPosition = MIN(StartingPosition, i);

	SkipHeuristic=0;
	for(i=StartingPosition;i<=BB;i++) { // StartingPosition is in range 1..
		for (j=1;j<=n;j++) {
			if(wrdLOW[l+i-1] == wrdARG[j-1])
				LevenshteinTd[i][j] = LevenshteinTd[i-1][j-1];
			else
#if defined(_WIN32ASM_)
				LevenshteinTd[i][j] = min_AF(LevenshteinTd[i-1][j]+1, LevenshteinTd[i][j-1]+1, LevenshteinTd[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinTd[i][j] = MIN(MIN((LevenshteinTd[i-1][j]+1),(LevenshteinTd[i][j-1]+1)),(LevenshteinTd[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinTd[i][j] = MIN(MIN(LevenshteinTd[i-1][j],LevenshteinTd[i][j-1]),LevenshteinTd[i-1][j-1]); --LevenshteinTd[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! // 237,270 xgrams/s
#endif
		}

		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinTd[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinTd[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinTd[BB][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLinesd++;
					}
					// Once dumping the line we need double 'break' from BB and l 'for's:
					goto EXHAUSTIVEd;
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
					//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
				}
	// The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
	//memcpy(wrdCACHEDTd, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
	memcpy(wrdCACHEDTd, &wrdLOW[l], BB); wrdCACHEDTd[BB]=0;
// LD ]
}

			} // if (l < m-BB+1)
			}
			}
EXHAUSTIVEd:;

// Fast! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// EXHAUSTIVE ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]

} else {
if (m<=MaxLineLength)
if (AtMostLevenshteinDistance >= MAX(m,n)-MIN(m,n)) // This is the only add-on for r.1+
{
	WordsCheckedd++;
// LD [
StartingPosition = 1;
while (	wrdCACHEDTd[StartingPosition-1] && wrdCACHEDTd[StartingPosition-1]==wrdLOW[StartingPosition-1] ) // No need of && wrd[StartingPosition-1]
	StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

	SkipHeuristic=0;
	for(i=StartingPosition;i<=m;i++) { // StartingPosition is in range 1..
		for (j=1;j<=n;j++) {
			if(wrdLOW[i-1] == wrdARG[j-1])
				LevenshteinTd[i][j] = LevenshteinTd[i-1][j-1];
			else
#if defined(_WIN32ASM_)
				LevenshteinTd[i][j] = min_AF(LevenshteinTd[i-1][j]+1, LevenshteinTd[i][j-1]+1, LevenshteinTd[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
```

```
#else
                                             LevenshteinTd[i][j] = MIN(MIN((LevenshteinTd[i-1][j]+1),(LevenshteinTd[i][j-1]+1)),(LevenshteinTd[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                             //{LevenshteinTd[i][j] = MIN(MIN(LevenshteinTd[i-1][j],LevenshteinTd[i][j-1]),LevenshteinTd[i-1][j-1]); --LevenshteinTd[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                       }

                       // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                       if ( LevenshteinTd[i][n] - (m-i) >= 0 && AtMostLevenshteinDistance < LevenshteinTd[i][n] - (m-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
           }
                                             if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinTd[m][n]) {
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                        fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLinesd++;
                                                        }
                                                        //if ((DumpedLines & 0xff) == 0xff)
                                                        //          //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                        //          fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                             }
           // The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
           memcpy(wrdCACHEDTd, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
// LD ]
}
} //if (EXHAUSTIVE_flag == 1)

} //if (WILDCARD_IP_flag)
                                  }
// Wildcard search ]
                                             wrdlen = 0;
                        }
                        else wrdlen++;
        } // k 'for'
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                             } // if (Exact_flag) {
           }

// Eth thread
#ifdef Commence_OpenMP
    #pragma omp section
#endif
           {

                                             if (Exact_flag) {
// WHOLE buffer at once not line-by-line [[[[[[[ Since r.1-++
                                                        k = WorkAreaLedgeTe;
                                                        while ( k < WorkAreaRedgeTe ) {
                                                        #ifdef RG7Gulliver
                                                                    //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_e(&xgamsCACHE[k], wrdARG, WorkAreaRedgeTe-k+1, n);
                                                                    FOUNDinPTR = Railgun_Sekireigan_Wolfram_e(&xgamsCACHE[k], wrdARG, WorkAreaRedgeTe-k+1, n);
                                                        #else
                                                                    FOUNDinPTR = Railgun_Quadruplet_7_e(&xgamsCACHE[k], wrdARG, WorkAreaRedgeTe-k+1, n);
                                                        #endif
                                                        // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                        // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                        //if ((long)( FOUNDinPTR - &xgamsCACHE[k] )>=0) {
                                                        if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(&xgamsCACHE[k])) {
                                                                    i = k + (long)( FOUNDinPTR - &xgamsCACHE[k] ); j = i;
                                                                    while (xgamsCACHE[j] != 10) {++j;} // Works both on UNIX(LF) and Windows(CRLF)
                                                                    while (i > k && xgamsCACHE[i-1] != 10) {--i;}
                                                                    k = j+1; // Should "point" to first symbol after the dumped fragment.

                                                                    //fwrite( &xgamsCACHE[i], j - i + 1, 1, fp_outLINE ); DumpedLines1++;
                                                                    if (j - i + 1 <= 168*MaxLineLength) { // fix for 1-++
                                                                    memcpy(wrd, &xgamsCACHE[i], j - i + 1); wrd[j - i + 1]=0; DumpedLinese++;
                                                                    j--; if (wrd[j - i + 1]==10) wrd[j - i + 1]=0;
                                                                    j--; if (wrd[j - i + 1]==13) wrd[j - i + 1]=0;
```

```
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                    if (Dump_flag)
                                                                    fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]);
                                                                    else
                                                                    fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                                    }
                                                        } else k = WorkAreaRedgeTe;
                                                        } // while
// WHOLE buffer at once not line-by-line ]]]]]]]] Since r.1-++
                                             } else { // if (Exact_flag) {

           //memset (wrdCACHEDTe, 0, MaxLineLength+1+1);
           i = 1;
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
           wrdlen = 0;
        for( k = WorkAreaLedgeTe; k <= WorkAreaRedgeTe; k++ )
           {
                      workbyte = xgamsCACHE[k];

                        if( wrdlen < MAXboth) {
                                                        if (CaseSensitiveWildcardMatching_flag == 0)
                                   wrdLOW[ wrdlen ] = KAZE_tolower( workbyte );
                                                        else
                                   wrdLOW[ wrdlen ] = ( workbyte );
                                             wrd[ wrdlen ] = workbyte;
                                  }
                        if (workbyte == 10) {
                                             TotalLinese++;
// Wildcard search [
                        if ( 0 < wrdlen && wrdlen < MAXboth)
                                  {
                                             wrd[ wrdlen ] = 0;
                                             wrdLOW[ wrdlen ] = 0;
                                             if ( wrd[ wrdlen-1 ] == 13 ) //CR
                                                        {--wrdlen; wrd[ wrdlen ] = 0; wrdLOW[ wrdlen ] = 0;}

if (WILDCARD_IP_flag) {
// WILDCARD IP [
                                             WordsCheckede++;
                                             if (Exact_flag) {
                                                        //if ((long)( Railgun_Quadruplet_7Gulliver_e(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                        #ifdef RG7Gulliver
                                                                    //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_e(wrdLOW, wrdARG, wrdlen, n);
                                                                    FOUNDinPTR = Railgun_Sekireigan_Wolfram_e(wrdLOW, wrdARG, wrdlen, n);
                                                        #else
                                                                    FOUNDinPTR = Railgun_Quadruplet_7_e(wrdLOW, wrdARG, wrdlen, n);
                                                        #endif
                                                        // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                        // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                        //if ((long)( FOUNDinPTR - wrdLOW )>=0)
                                                        if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(wrdLOW))
                                                        //if ((long)( Railgun_Quadruplet_7_e(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                                    {
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                    DumpedLinese++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                                    }
                                             } else {
                                                        if (WILDCARD_FAST_flag) {
#if defined(_WildFastKaze_)
```

```
                                                        if ( WildcardMatch_Iterative_Kazee(wrdARG, wrdLOW) ) {
#else
                                                        if ( IterativeWildcardse(wrdARG, wrdLOW) ) {
#endif
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                DumpedLinese++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                        }
                                                        } else {
                                                        maskGLOBALlen = n;
                                                        nameGLOBALlene = wrdlen;
                                                        if ( EnhancedMaskTest_OrEmpty_AndNotEmpty_e(wrdARG, 0, wrdLOW, 0) ) { // Caution: Not lowercased as it should!
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                                DumpedLinese++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                        }
                                                        }
                                                }
// WILDCARD IP ]
} else {

// A simple heuristic #1: Don't enter the nasty loops unless MaximumLevenshteinDistance >= ABS(m-n).
m = wrdlen; // strlen(wrd);
//if (m>MaxLineLength)
//{ printf( "\nKazahana: Incoming xgram exceeding the limit.\n" ); exit( 2 ); }
// Above two commented lines are too severe, changed with next line allowing to search into lines bigger than our needs:
if (m)
if (EXHAUSTIVE_flag == 1) {

// Here we'll walk through the whole length of 'm', ay-yaa.
// EXHAUSTIVE [[[[[[[[[[[[[[[[[[[[[[[[[[[[[[
// Here the old fuzzy is replaced with exhaustive one, using BBs of the incoming string (here 'm').
// We need to factorize 'm' down to all 'n+AtMostLevenshteinDistance' long strings/BBs and to search into them ONE-BY-ONE - a gruelling task indeed!
//// Example:
// One of those 33/5 lines is a 46 bytes long string, i.e. m = 46:
// |Edelweiß || [[edelweiss]] || edelweiss flower
// Let's search fuzzily for "edelvais", pretending we don't know the right spelling:
// To find "edelweiss" we need at least Levenshtein distance 3, i.e. n = 8, AtMostLevenshteinDistance = 3:
// 'edelvais' vs 'edelweiss':
// - the mispelled one has one character less;
// - the mispelled one has 2 wrong characters: 'v' & 'a' instead of 'w' & 'e'.
// From above we need Building-Blocks of 46 bytes order 8+3.
// Inhere we are using order 11, 'm - Order + 1' is the number of total BBs for text 'm' bytes long: 46-11+1 = 36
// The 36 BBs:
// |Edelweiß |
// Edelweiß ||
// ...
// weiss flowe
// eiss flower

// Slow! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
/*
                                for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                for (l=0; l < m-BB+1; l++) {
            WordsCheckede++;
// LD [
            SkipHeuristic=0;
            for(i=1;i<=BB;i++) {
                        for (j=1;j<=n;j++) {
                                    if(wrdLOW[l+i-1] == wrdARG[j-1])
                                                LevenshteinTe[i][j] = LevenshteinTe[i-1][j-1];
```

```
                                        else
#if defined(_WIN32ASM_)
                                                LevenshteinTe[i][j] = min_AF(LevenshteinTe[i-1][j]+1, LevenshteinTe[i][j-1]+1, LevenshteinTe[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                                LevenshteinTe[i][j] = MIN(MIN((LevenshteinTe[i-1][j]+1),(LevenshteinTe[i][j-1]+1)),(LevenshteinTe[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                                //{LevenshteinTe[i][j] = MIN(MIN(LevenshteinTe[i-1][j],LevenshteinTe[i][j-1]),LevenshteinTe[i-1][j-1]); --LevenshteinTe[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. BB) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinTe[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinTe[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                                if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinTe[BB][n]) {
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical
#endif
                                                        {
                                                        fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLinese++;
                                                        }
                                                        goto EXHAUSTIVEe;
                                                        //break; // No need of further checking down the line, one dump only is needed.
                                                        //if ((DumpedLines & 0xff) == 0xff)
                                                        //        //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                        //        fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                }
// LD ]
                                        }
                                        }
EXHAUSTIVEe:;
*/
// Slow! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Fast! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

                                        for (l=0; l < m-(n-AtMostLevenshteinDistance)+1; l++) { // Here BB = n-AtMostLevenshteinDistance the smallest BB
                                        for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                        // From here on 'm' will be replaced by BB
                                        if (l < m-BB+1) {

if (AtMostLevenshteinDistance >= MAX(BB,n)-MIN(BB,n)) // This is the only add-on for r.1+
{
            WordsCheckede++;
// LD [
StartingPosition = 1;
while (     wrdCACHEDTe[StartingPosition-1] && wrdLOW[l+StartingPosition-1] && wrdCACHEDTe[StartingPosition-1]==wrdLOW[l+StartingPosition-1] )
            StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

            SkipHeuristic=0;
            for(i=StartingPosition;i<=BB;i++) { // StartingPosition is in range 1..
                        for (j=1;j<=n;j++) {
                                        if(wrdLOW[l+i-1] == wrdARG[j-1])
                                                LevenshteinTe[i][j] = LevenshteinTe[i-1][j-1];
                                        else
#if defined(_WIN32ASM_)
                                                LevenshteinTe[i][j] = min_AF(LevenshteinTe[i-1][j]+1, LevenshteinTe[i][j-1]+1, LevenshteinTe[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                                LevenshteinTe[i][j] = MIN(MIN((LevenshteinTe[i-1][j]+1),(LevenshteinTe[i][j-1]+1)),(LevenshteinTe[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                                //{LevenshteinTe[i][j] = MIN(MIN(LevenshteinTe[i-1][j],LevenshteinTe[i][j-1]),LevenshteinTe[i-1][j-1]); --LevenshteinTe[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinTe[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinTe[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
```

```
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinTe[BB][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLinese++;
					}
					// Once dumping the line we need double 'break' from BB and l 'for's:
					goto EXHAUSTIVEe;
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
					//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
				}
	// The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
	//memcpy(wrdCACHEDTe, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
	memcpy(wrdCACHEDTe, &wrdLOW[l], BB); wrdCACHEDTe[BB]=0;
// LD ]
}

			} // if (l < m-BB+1)
			}
			}
EXHAUSTIVEe:;

// Fast! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// EXHAUSTIVE ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]

} else {
if (m<=MaxLineLength)
if (AtMostLevenshteinDistance >= MAX(m,n)-MIN(m,n)) // This is the only add-on for r.1+
{
	WordsCheckede++;
// LD [
StartingPosition = 1;
while (	wrdCACHEDTe[StartingPosition-1] && wrdCACHEDTe[StartingPosition-1]==wrdLOW[StartingPosition-1] ) // No need of && wrd[StartingPosition-1]
	StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

	SkipHeuristic=0;
	for(i=StartingPosition;i<=m;i++) { // StartingPosition is in range 1..
		for (j=1;j<=n;j++) {
			if(wrdLOW[i-1] == wrdARG[j-1])
				LevenshteinTe[i][j] = LevenshteinTe[i-1][j-1];
			else
#if defined(_WIN32ASM_)
				LevenshteinTe[i][j] = min_AF(LevenshteinTe[i-1][j]+1, LevenshteinTe[i][j-1]+1, LevenshteinTe[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinTe[i][j] = MIN(MIN((LevenshteinTe[i-1][j]+1),(LevenshteinTe[i][j-1]+1)),(LevenshteinTe[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinTe[i][j] = MIN(MIN(LevenshteinTe[i-1][j],LevenshteinTe[i][j-1]),LevenshteinTe[i-1][j-1]); --LevenshteinTe[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
		}

		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinTe[i][n] - (m-i) >= 0 && AtMostLevenshteinDistance < LevenshteinTe[i][n] - (m-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinTe[m][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLinese++;
					}
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
```

```
                                                        //         fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                }
            // The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
            memcpy(wrdCACHEDTe, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
// LD ]
}
} //if (EXHAUSTIVE_flag == 1)

} //if (WILDCARD_IP_flag)
                                        }
// Wildcard search ]
                                                wrdlen = 0;
                                        }
                                        else wrdlen++;
        } // k 'for'
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                                } // if (Exact_flag) {
            }

// Fth thread
#ifdef Commence_OpenMP
    #pragma omp section
#endif
            {

                                                if (Exact_flag) {
// WHOLE buffer at once not line-by-line [[[[[[[[ Since r.1-++
                                                        k = WorkAreaLedgeTf;
                                                        while ( k < WorkAreaRedgeTf ) {
                                                        #ifdef RG7Gulliver
                                                                //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_f(&xgamsCACHE[k], wrdARG, WorkAreaRedgeTf-k+1, n);
                                                                FOUNDinPTR = Railgun_Sekireigan_Wolfram_f(&xgamsCACHE[k], wrdARG, WorkAreaRedgeTf-k+1, n);
                                                        #else
                                                                FOUNDinPTR = Railgun_Quadruplet_7_f(&xgamsCACHE[k], wrdARG, WorkAreaRedgeTf-k+1, n);
                                                        #endif
                                                        // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                        // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                        //if ((long)( FOUNDinPTR - &xgamsCACHE[k] )>=0) {
                                                        if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(&xgamsCACHE[k])) {
                                                                i = k + (long)( FOUNDinPTR - &xgamsCACHE[k] ); j = i;
                                                                while (xgamsCACHE[j] != 10) {++j;} // Works both on UNIX(LF) and Windows(CRLF)
                                                                while (i > k && xgamsCACHE[i-1] != 10) {--i;}
                                                                k = j+1; // Should "point" to first symbol after the dumped fragment.

                                                                //fwrite( &xgamsCACHE[i], j - i + 1, 1, fp_outLINE ); DumpedLines1++;
                                                                if (j - i + 1 <= 168*MaxLineLength) { // fix for 1-++
                                                                memcpy(wrd, &xgamsCACHE[i], j - i + 1); wrd[j - i + 1]=0; DumpedLinesf++;
                                                                j--; if (wrd[j - i + 1]==10) wrd[j - i + 1]=0;
                                                                j--; if (wrd[j - i + 1]==13) wrd[j - i + 1]=0;
                                                        // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                        #pragma omp critical

#endif
                                                        {
                                                                if (Dump_flag)
                                                                fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]);
                                                                else
                                                                fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                                }
                                                        } else k = WorkAreaRedgeTf;
                                                        } // while
// WHOLE buffer at once not line-by-line ]]]]]]]] Since r.1-++
                                                } else { // if (Exact_flag) {

            //memset (wrdCACHEDTf, 0, MaxLineLength+1+1);
            i = 1;
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            wrdlen = 0;
```

```
        for( k = WorkAreaLedgeTf; k <= WorkAreaRedgeTf; k++ )
           {
                        workbyte = xgamsCACHE[k];

                         if( wrdlen < MAXboth) {
                                                  if (CaseSensitiveWildcardMatching_flag == 0)
                                   wrdLOW[ wrdlen ] = KAZE_tolower( workbyte );
                                                  else
                                   wrdLOW[ wrdlen ] = ( workbyte );
                                            wrd[ wrdlen ] = workbyte;
                                        }
                         if (workbyte == 10) {
                                            TotalLinesf++;
// Wildcard search [
                         if ( 0 < wrdlen && wrdlen < MAXboth)
                                        {
                                            wrd[ wrdlen ] = 0;
                                            wrdLOW[ wrdlen ] = 0;
                                            if ( wrd[ wrdlen-1 ] == 13 ) //CR
                                                      {--wrdlen; wrd[ wrdlen ] = 0; wrdLOW[ wrdlen ] = 0;}

if (WILDCARD_IP_flag) {
// WILDCARD IP [
                                            WordsCheckedf++;
                                            if (Exact_flag) {
                                                      //if ((long)( Railgun_Quadruplet_7Gulliver_f(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                      #ifdef RG7Gulliver
                                                                //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_f(wrdLOW, wrdARG, wrdlen, n);
                                                                FOUNDinPTR = Railgun_Sekireigan_Wolfram_f(wrdLOW, wrdARG, wrdlen, n);
                                                      #else
                                                                FOUNDinPTR = Railgun_Quadruplet_7_f(wrdLOW, wrdARG, wrdlen, n);
                                                      #endif
                                                      // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                      // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                      //if ((long)( FOUNDinPTR - wrdLOW )>=0)
                                                      if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(wrdLOW))
                                                      //if ((long)( Railgun_Quadruplet_7_f(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                                {
                                                      // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                      #pragma omp critical
#endif
                                                      {
                                                                DumpedLinesf++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                      }
                                                                }
                                            } else {
                                                      if (WILDCARD_FAST_flag) {
#if defined(_WildFastKaze_)
                                                      if ( WildcardMatch_Iterative_Kazef(wrdARG, wrdLOW) ) {
#else
                                                      if ( IterativeWildcardsf(wrdARG, wrdLOW) ) {
#endif
                                                      // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                      #pragma omp critical
#endif
                                                      {
                                                                DumpedLinesf++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                      }
                                                      }
                                                      } else {
                                                      maskGLOBALlen = n;
                                                      nameGLOBALlenf = wrdlen;
                                                      if ( EnhancedMaskTest_OrEmpty_AndNotEmpty_f(wrdARG, 0, wrdLOW, 0) ) { // Caution: Not lowercased as it should!
                                                      // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                      #pragma omp critical
#endif
```

```
                                                        {
                                                                DumpedLinesf++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                        }
                                                        }
                                                        }
                                                }
// WILDCARD IP ]
} else {

// A simple heuristic #1: Don't enter the nasty loops unless MaximumLevenshteinDistance >= ABS(m-n).
m = wrdlen; // strlen(wrd);
//if (m>MaxLineLength)
//{ printf( "\nKazahana: Incoming xgram exceeding the limit.\n" ); exit( 2 ); }
// Above two commented lines are too severe, changed with next line allowing to search into lines bigger than our needs:
if (m)
if (EXHAUSTIVE_flag == 1) {

// Here we'll walk through the whole length of 'm', ay-yaa.
// EXHAUSTIVE [[[[[[[[[[[[[[[[[[[[[[[[[[[[[
// Here the old fuzzy is replaced with exhaustive one, using BBs of the incoming string (here 'm').
// We need to factorize 'm' down to all 'n+AtMostLevenshteinDistance' long strings/BBs and to search into them ONE-BY-ONE - a gruelling task indeed!
//// Example:
// One of those 33/5 lines is a 46 bytes long string, i.e. m = 46:
// |Edelweiß || [[edelweiss]] || edelweiss flower
// Let's search fuzzily for "edelvais", pretending we don't know the right spelling:
// To find "edelweiss" we need at least Levenshtein distance 3, i.e. n = 8, AtMostLevenshteinDistance = 3:
// 'edelvais' vs 'edelweiss':
// - the mispelled one has one character less;
// - the mispelled one has 2 wrong characters: 'v' & 'a' instead of 'w' & 'e'.
// From above we need Building-Blocks of 46 bytes order 8+3.
// Inhere we are using order 11, 'm - Order + 1' is the number of total BBs for text 'm' bytes long: 46-11+1 = 36
// The 36 BBs:
// |Edelweiß |
// Edelweiß ||
// ...
// weiss flowe
// eiss flower

// Slow! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
/*
                                for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                for (l=0; l < m-BB+1; l++) {
            WordsCheckedf++;
// LD [
            SkipHeuristic=0;
            for(i=1;i<=BB;i++) {
                        for (j=1;j<=n;j++) {
                                if(wrdLOW[l+i-1] == wrdARG[j-1])
                                        LevenshteinTf[i][j] = LevenshteinTf[i-1][j-1];
                                else
#if defined(_WIN32ASM_)
                                        LevenshteinTf[i][j] = min_AF(LevenshteinTf[i-1][j]+1, LevenshteinTf[i][j-1]+1, LevenshteinTf[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                        LevenshteinTf[i][j] = MIN(MIN((LevenshteinTf[i-1][j]+1),(LevenshteinTf[i][j-1]+1)),(LevenshteinTf[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                        //{LevenshteinTf[i][j] = MIN(MIN(LevenshteinTf[i-1][j],LevenshteinTf[i][j-1]),LevenshteinTf[i-1][j-1]); --LevenshteinTf[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. BB) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinTf[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinTf[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                        if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinTf[BB][n]) {
                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                #pragma omp critical
#endif
                                                {
                                                fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLinesf++;
```

```
                                                        }
                                                        goto EXHAUSTIVEf;
                                                        //break; // No need of further checking down the line, one dump only is needed.
                                                        //if ((DumpedLines & 0xff) == 0xff)
                                                        //        //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                        //        fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                }
// LD ]
                                        }
                                        }
EXHAUSTIVEf:;
*/
// Slow! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Fast! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

                                        for (l=0; l < m-(n-AtMostLevenshteinDistance)+1; l++) { // Here BB = n-AtMostLevenshteinDistance the smallest BB
                                        for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                        // From here on 'm' will be replaced by BB
                                        if (l < m-BB+1) {

if (AtMostLevenshteinDistance >= MAX(BB,n)-MIN(BB,n)) // This is the only add-on for r.1+
{
                WordsCheckedf++;
// LD [
StartingPosition = 1;
while (     wrdCACHEDTf[StartingPosition-1] && wrdLOW[l+StartingPosition-1] && wrdCACHEDTf[StartingPosition-1]==wrdLOW[l+StartingPosition-1] )
                StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

                SkipHeuristic=0;
                for(i=StartingPosition;i<=BB;i++) { // StartingPosition is in range 1..
                                for (j=1;j<=n;j++) {
                                                if(wrdLOW[l+i-1] == wrdARG[j-1])
                                                                LevenshteinTf[i][j] = LevenshteinTf[i-1][j-1];
                                                else
#if defined(_WIN32ASM_)
                                                                LevenshteinTf[i][j] = min_AF(LevenshteinTf[i-1][j]+1, LevenshteinTf[i][j-1]+1, LevenshteinTf[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                                                LevenshteinTf[i][j] = MIN(MIN((LevenshteinTf[i-1][j]+1),(LevenshteinTf[i][j-1]+1)),(LevenshteinTf[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                                                //{LevenshteinTf[i][j] = MIN(MIN(LevenshteinTf[i-1][j],LevenshteinTf[i][j-1]),LevenshteinTf[i-1][j-1]); --LevenshteinTf[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                                }

                                // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                                if ( LevenshteinTf[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinTf[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
                }
                                                                LevenshteinTf[i][j] = MIN(MIN(LevenshteinTf[i-1][j],LevenshteinTf[i][j-1]),LevenshteinTf[i-1][j-1]); --LevenshteinTf[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                                if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinTf[BB][n]) {
                                                // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                #pragma omp critical
#endif
                                                {
                                                fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLinesf++;
                                                }
                                                // Once dumping the line we need double 'break' from BB and l 'for's:
                                                goto EXHAUSTIVEf;
                                                //if ((DumpedLines & 0xff) == 0xff)
                                                //        //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                //        fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                }
                // The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
                //memcpy(wrdCACHEDTf, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
                memcpy(wrdCACHEDTf, &wrdLOW[l], BB); wrdCACHEDTf[BB]=0;
// LD ]
}
```

```
                                    } // if (l < m-BB+1)
                                    }
                                    }
EXHAUSTIVEf:;

// Fast! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// EXHAUSTIVE ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]

} else {
if (m<=MaxLineLength)
if (AtMostLevenshteinDistance >= MAX(m,n)-MIN(m,n)) // This is the only add-on for r.1+
{
            WordsCheckedf++;
// LD [
StartingPosition = 1;
while (     wrdCACHEDTf[StartingPosition-1] && wrdCACHEDTf[StartingPosition-1]==wrdLOW[StartingPosition-1] ) // No need of && wrd[StartingPosition-1]
            StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

            SkipHeuristic=0;
            for(i=StartingPosition;i<=m;i++) { // StartingPosition is in range 1..
                        for (j=1;j<=n;j++) {
                                    if(wrdLOW[i-1] == wrdARG[j-1])
                                                LevenshteinTf[i][j] = LevenshteinTf[i-1][j-1];
                                    else
#if defined(_WIN32ASM_)
                                                LevenshteinTf[i][j] = min_AF(LevenshteinTf[i-1][j]+1, LevenshteinTf[i][j-1]+1, LevenshteinTf[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                                LevenshteinTf[i][j] = MIN(MIN((LevenshteinTf[i-1][j]+1),(LevenshteinTf[i][j-1]+1)),(LevenshteinTf[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                                //{LevenshteinTf[i][j] = MIN(MIN(LevenshteinTf[i-1][j],LevenshteinTf[i][j-1]),LevenshteinTf[i-1][j-1]); --LevenshteinTf[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }

                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinTf[i][n] - (m-i) >= 0 && AtMostLevenshteinDistance < LevenshteinTf[i][n] - (m-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                                if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinTf[m][n]) {
                                                            // Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
                                                            #pragma omp critical
#endif
                                                            {
                                                            fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLinesf++;
                                                            }
                                                            //if ((DumpedLines & 0xff) == 0xff)
                                                            //          //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                            //          fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                }
            // The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
            memcpy(wrdCACHEDTf, wrdLOW, m+1); // +1 because we need the ASCII 000 termination.
// LD ]
}
} //if (EXHAUSTIVE_flag == 1)

} //if (WILDCARD_IP_flag)
                                    }
// Wildcard search ]
                                                wrdlen = 0;
                         }
                         else wrdlen++;
        } // k 'for'
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                                } // if (Exact_flag) {
            }

    }
```

```
}// pragma

          memcpy(xgamsCACHE, &xgamsCACHE[ (CACHEsize-1-CACHEremainder)+1 ], CACHEremainder);
} //while (size_inLINESIXFOURleftforparsing >= CACHEsize-CACHEremainder) {


// We have not finished the 'size_inLINESIXFOURleftforparsing' loop yet! Must check for remainder! [
if (size_inLINESIXFOURleftforparsing != 0) {
          fread( xgamsCACHE+CACHEremainder, 1, size_inLINESIXFOURleftforparsing, fp_inLINE );
          // Working area: xgamsCACHE..xgamsCACHE +(CACHEremainder+size_inLINESIXFOURleftforparsing) -1
          //fwrite( xgamsCACHE, 1, xgamsCACHE +(CACHEremainder+size_inLINESIXFOURleftforparsing) -1 - xgamsCACHE +1, fp_outLINE ); //DELDEL
          //...

                                        if (Exact_flag) {
// WHOLE buffer at once not line-by-line [[[[[[[[ Since r.1-++
                                                k = 0;
                                                while ( k < (CACHEremainder+size_inLINESIXFOURleftforparsing) -1 ) {
                                                #ifdef RG7Gulliver
                                                          //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_1(&xgamsCACHE[k], wrdARG, (CACHEremainder+size_inLINESIXFOURleftforparsing) -1-k+1, n);
                                                          FOUNDinPTR = Railgun_Sekireigan_Wolfram_1(&xgamsCACHE[k], wrdARG, (CACHEremainder+size_inLINESIXFOURleftforparsing) -1-k+1, n);
                                                #else
                                                          FOUNDinPTR = Railgun_Quadruplet_7_1(&xgamsCACHE[k], wrdARG, (CACHEremainder+size_inLINESIXFOURleftforparsing) -1-k+1, n);
                                                #endif
                                                // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                //if ((long)( FOUNDinPTR - &xgamsCACHE[k] )>=0) {
                                                if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(&xgamsCACHE[k])) {
                                                          i = k + (long)( FOUNDinPTR - &xgamsCACHE[k] ); j = i;
                                                          while (xgamsCACHE[j] != 10) {++j;} // Works both on UNIX(LF) and Windows(CRLF)
                                                          while (i > k && xgamsCACHE[i-1] != 10) {--i;}
                                                          k = j+1; // Should "point" to first symbol after the dumped fragment.

                                                          //fwrite( &xgamsCACHE[i], j - i + 1, 1, fp_outLINE ); DumpedLines1++;
                                                          if (j - i + 1 <= 168*MaxLineLength) { // fix for 1-++
                                                          memcpy(wrd, &xgamsCACHE[i], j - i + 1); wrd[j - i + 1]=0; DumpedLines++;
                                                          j--; if (wrd[j - i + 1]==10) wrd[j - i + 1]=0;
                                                          j--; if (wrd[j - i + 1]==13) wrd[j - i + 1]=0;
                                                          if (Dump_flag)
                                                          fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]);
                                                          else
                                                          fprintf( fp_outLINE, "%s\r\n", wrd);
                                                          }
                                                } else k = (CACHEremainder+size_inLINESIXFOURleftforparsing) -1;
                                                } // while
// WHOLE buffer at once not line-by-line ]]]]]]]] Since r.1-++
                                        } else { // if (Exact_flag) {

          //memset (wrdCACHEDT1, 0, MaxLineLength+1+1);
          i = 1;
       //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
          wrdlen = 0;
       for( k = 0; k <= (CACHEremainder+size_inLINESIXFOURleftforparsing) -1; k++ )
          {
                    workbyte = xgamsCACHE[k];

                     if( wrdlen < MAXboth) {
                                                if (CaseSensitiveWildcardMatching_flag == 0)
                               wrdLOW[ wrdlen ] = KAZE_tolower( workbyte );
                                                else
                               wrdLOW[ wrdlen ] = ( workbyte );
                                        wrd[ wrdlen ] = workbyte;
                                    }
                     if (workbyte == 10) {
                                        TotalLines++;
// Wildcard search [
                     if ( 0 < wrdlen && wrdlen < MAXboth)
                                    {
                                        wrd[ wrdlen ] = 0;
                                        wrdLOW[ wrdlen ] = 0;
```

```
                                        if ( wrd[ wrdlen-1 ] == 13 ) //CR
                                                {--wrdlen; wrd[ wrdlen ] = 0; wrdLOW[ wrdlen ] = 0;}

if (WILDCARD_IP_flag) {
// WILDCARD IP [
                                        WordsChecked++;
                                        if (Exact_flag) {
                                                //if ((long)( Railgun_Quadruplet_7Gulliver_1(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                #ifdef RG7Gulliver
                                                        //FOUNDinPTR = Railgun_Quadruplet_7Gulliver_1(wrdLOW, wrdARG, wrdlen, n);
                                                        FOUNDinPTR = Railgun_Sekireigan_Wolfram_1(wrdLOW, wrdARG, wrdlen, n);
                                                #else
                                                        FOUNDinPTR = Railgun_Quadruplet_7_1(wrdLOW, wrdARG, wrdlen, n);
                                                #endif
                                                // Commented line below works under MinGW & Intel 12.1 for Windows but fails under Linux:
                                                // Linux thinks that 0 - ~3,000,000,000 = ~1,000,000,000
                                                //if ((long)( FOUNDinPTR - wrdLOW )>=0)
                                                if ((unsigned long long)( FOUNDinPTR) >=(unsigned long long)(wrdLOW))
                                                //if ((long)( Railgun_Quadruplet_7_1(wrdLOW, wrdARG, wrdlen, n) - wrdLOW )>=0)
                                                        {DumpedLines++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);}
                                        } else {
                                                if (WILDCARD_FAST_flag) {
#if defined(_WildFastKaze_)
                                                if ( WildcardMatch_Iterative_Kaze1(wrdARG, wrdLOW) ) {
#else
                                                if ( IterativeWildcards1(wrdARG, wrdLOW) ) {
#endif
                                                        DumpedLines++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                }
                                                } else {
                                                maskGLOBALlen = n;
                                                nameGLOBALlen1 = wrdlen;
                                                if ( EnhancedMaskTest_OrEmpty_AndNotEmpty_1(wrdARG, 0, wrdLOW, 0) ) { // Caution: Not lowercased as it should!
                                                        DumpedLines++; if (Dump_flag) fprintf( fp_outLINE, "[%s] %s /%s/\r\n", wrdARG, wrd, argv[3-WILDCARD_IP_flag]); else fprintf( fp_outLINE, "%s\r\n", wrd);
                                                }
                                                }
                                        }
// WILDCARD IP ]
} else {

// A simple heuristic #1: Don't enter the nasty loops unless MaximumLevenshteinDistance >= ABS(m-n).
m = wrdlen; // strlen(wrd);
//if (m>MaxLineLength)
//{ printf( "\nKazahana: Incoming xgram exceeding the limit.\n" ); exit( 2 ); }
// Above two commented lines are too severe, changed with next line allowing to search into lines bigger than our needs:
if (m)
if (EXHAUSTIVE_flag == 1) {

// Here we'll walk through the whole length of 'm', ay-yaa.
// EXHAUSTIVE [[[[[[[[[[[[[[[[[[[[[[[[[[[[
// Here the old fuzzy is replaced with exhaustive one, using BBs of the incoming string (here 'm').
// We need to factorize 'm' down to all 'n+AtMostLevenshteinDistance' long strings/BBs and to search into them ONE-BY-ONE - a gruelling task indeed!
//// Example:
// One of those 33/5 lines is a 46 bytes long string, i.e. m = 46:
// |Edelweiß || [[edelweiss]] || edelweiss flower
// Let's search fuzzily for "edelvais", pretending we don't know the right spelling:
// To find "edelweiss" we need at least Levenshtein distance 3, i.e. n = 8, AtMostLevenshteinDistance = 3:
// 'edelvais' vs 'edelweiss':
// - the mispelled one has one character less;
// - the mispelled one has 2 wrong characters: 'v' & 'a' instead of 'w' & 'e'.
// From above we need Building-Blocks of 46 bytes order 8+3.
// Inhere we are using order 11, 'm - Order + 1' is the number of total BBs for text 'm' bytes long: 46-11+1 = 36
// The 36 BBs:
// |Edelweiß |
// Edelweiß ||
// ...
// weiss flowe
// eiss flower
```

```
// Slow! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
/*
                                for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                for (l=0; l < m-BB+1; l++) {
            WordsChecked++;
// LD [
            SkipHeuristic=0;
            for(i=1;i<=BB;i++) {
                        for (j=1;j<=n;j++) {
                                    if(wrdLOW[l+i-1] == wrdARG[j-1])
                                                LevenshteinT1[i][j] = LevenshteinT1[i-1][j-1];
                                    else
#if defined(_WIN32ASM_)
                                                LevenshteinT1[i][j] = min_AF(LevenshteinT1[i-1][j]+1, LevenshteinT1[i][j-1]+1, LevenshteinT1[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                                LevenshteinT1[i][j] = MIN(MIN((LevenshteinT1[i-1][j]+1),(LevenshteinT1[i][j-1]+1)),(LevenshteinT1[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                                //{LevenshteinT1[i][j] = MIN(MIN(LevenshteinT1[i-1][j],LevenshteinT1[i][j-1]),LevenshteinT1[i-1][j-1]); --LevenshteinT1[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
                        }
                        // A simple heuristic #2: Discontinue the nasty vertical loop (i.e. BB) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
                        if ( LevenshteinT1[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT1[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
            }
                                                if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT1[BB][n]) {
                                                            {
                                                            fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines++;
                                                            }
                                                            goto EXHAUSTIVE;
                                                            //break; // No need of further checking down the line, one dump only is needed.
                                                            //if ((DumpedLines & 0xff) == 0xff)
                                                            //         //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                            //         fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                                }
// LD ]
                                }
                                }
EXHAUSTIVE:;
*/
// Slow! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// Fast! [<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

                                for (l=0; l < m-(n-AtMostLevenshteinDistance)+1; l++) { // Here BB = n-AtMostLevenshteinDistance the smallest BB
                                for (BB=n-AtMostLevenshteinDistance; BB <=n+AtMostLevenshteinDistance; BB++) {
                                // From here on 'm' will be replaced by BB
                                if (l < m-BB+1) {

if (AtMostLevenshteinDistance >= MAX(BB,n)-MIN(BB,n)) // This is the only add-on for r.1+
{
            WordsChecked++;
// LD [
StartingPosition = 1;
while (     wrdCACHEDT1[StartingPosition-1] && wrdLOW[l+StartingPosition-1] && wrdCACHEDT1[StartingPosition-1]==wrdLOW[l+StartingPosition-1] )
            StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

            SkipHeuristic=0;
            for(i=StartingPosition;i<=BB;i++) { // StartingPosition is in range 1..
                        for (j=1;j<=n;j++) {
                                    if(wrdLOW[l+i-1] == wrdARG[j-1])
                                                LevenshteinT1[i][j] = LevenshteinT1[i-1][j-1];
                                    else
#if defined(_WIN32ASM_)
                                                LevenshteinT1[i][j] = min_AF(LevenshteinT1[i-1][j]+1, LevenshteinT1[i][j-1]+1, LevenshteinT1[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
                                                LevenshteinT1[i][j] = MIN(MIN((LevenshteinT1[i-1][j]+1),(LevenshteinT1[i][j-1]+1)),(LevenshteinT1[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
                                                //{LevenshteinT1[i][j] = MIN(MIN(LevenshteinT1[i-1][j],LevenshteinT1[i][j-1]),LevenshteinT1[i-1][j-1]); --LevenshteinT1[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
```

```
#endif
		}

		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinT1[i][n] - (BB-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT1[i][n] - (BB-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT1[BB][n]) {
					// Below pragma is needed explicitly only for MinGW, grrr...
#ifdef Commence_OpenMP
					#pragma omp critical
#endif
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines++;
					}
					// Once dumping the line we need double 'break' from BB and l 'for's:
					goto EXHAUSTIVE;
					//if ((DumpedLines & 0xff) == 0xff)
					//	//printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
					//	fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
				}
	// The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
	//memcpy(wrdCACHEDT1, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
	memcpy(wrdCACHEDT1, &wrdLOW[l], BB); wrdCACHEDT1[BB]=0;
// LD ]
}

			} // if (l < m-BB+1)
			}
			}
EXHAUSTIVE:;

// Fast! ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// EXHAUSTIVE ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]

} else {
if (m<=MaxLineLength)
if (AtMostLevenshteinDistance >= MAX(m,n)-MIN(m,n)) // This is the only add-on for r.1+
{
	WordsChecked++;
// LD [
StartingPosition = 1;
while (	wrdCACHEDT1[StartingPosition-1] && wrdCACHEDT1[StartingPosition-1]==wrdLOW[StartingPosition-1] ) // No need of && wrd[StartingPosition-1]
	StartingPosition++;
// The bail out 'i' value (heuristic #2) affects our cached value here, 'StartingPosition' cannot be greater than 'i':
StartingPosition = MIN(StartingPosition, i);

	SkipHeuristic=0;
	for(i=StartingPosition;i<=m;i++) { // StartingPosition is in range 1..
		for (j=1;j<=n;j++) {
			if(wrdLOW[i-1] == wrdARG[j-1])
				LevenshteinT1[i][j] = LevenshteinT1[i-1][j-1];
			else
#if defined(_WIN32ASM_)
				LevenshteinT1[i][j] = min_AF(LevenshteinT1[i-1][j]+1, LevenshteinT1[i][j-1]+1, LevenshteinT1[i-1][j-1]+1); // Variant 1: 237,270 xgrams/s
#else
				LevenshteinT1[i][j] = MIN(MIN((LevenshteinT1[i-1][j]+1),(LevenshteinT1[i][j-1]+1)),(LevenshteinT1[i-1][j-1]+1)); // Variant 2: This MS code is faster than above jumpless code! // 358,327 xgrams/s
				//{LevenshteinT1[i][j] = MIN(MIN(LevenshteinT1[i-1][j],LevenshteinT1[i][j-1]),LevenshteinT1[i-1][j-1]); --LevenshteinT1[i][j];} // Variant 3: This compound line is much slower than above inc-inc-inc code! //
237,270 xgrams/s
#endif
		}

		// A simple heuristic #2: Discontinue the nasty vertical loop (i.e. m) when the LD in cell in the last column minus the remaining vertical cycles is greater than our MAX LD:
		if ( LevenshteinT1[i][n] - (m-i) >= 0 && AtMostLevenshteinDistance < LevenshteinT1[i][n] - (m-i) ) {SkipHeuristic=1; break;} // Caution: Levenshtein[i][n] can be less than (m-i), this changes nothing the logic is the same.
	}
				if (SkipHeuristic==0 && AtMostLevenshteinDistance >= LevenshteinT1[m][n]) {
					{
					fprintf( fp_outLINE, "%s\r\n", wrd); DumpedLines++;
					}
```

```
                                                          //if ((DumpedLines & 0xff) == 0xff)
                                                          //          //printf( "Dumped lines i.e. hits so far: %s\r", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                                          //          fflush(fp_outLINE); // Not sure: CTRL+C doesn't flush?!
                                              }
            // The bail out 'i' value (heuristic #2) affects our cached value here, 'i' is the needed one:
            memcpy(wrdCACHEDT1, wrdLOW, m+1); // +1 because we need the ASCII 000 termination;
// LD ]
}
} //if (EXHAUSTIVE_flag == 1)

} //if (WILDCARD_IP_flag)
                                  }
// Wildcard search ]
                                              wrdlen = 0;
                        }
                        else wrdlen++;
        } // k 'for'
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                              } // if (Exact_flag) {

}
// We have not finished the 'size_inLINESIXFOURleftforparsing' loop yet! Must check for remainder! ]
// MT PARSING MT PARSING MT PARSING MT PARSING MT PARSING MT PARSING MT PARSING MT PARSING MT PARSING MT PARSING MT PARSING MT PARSING ]

            fclose(fp_inLINE);
            fclose(fp_outLINE);
            //free(xgamsCACHE); // Cannot free it like that because it is not the same as when malloc() was used!
            (void) time(&t3);
            if (t3 <= t1) {t3 = t1; t3++;}
            clocks2 = clock();
            printf( "\n" );
            DumpedLines=DumpedLines+DumpedLines1+DumpedLines2+DumpedLines3+DumpedLines4+DumpedLines5+DumpedLines6+DumpedLines7+DumpedLines8+DumpedLines9+DumpedLines0+DumpedLinesa+DumpedLinesb+DumpedLinesc+DumpedLinesd+DumpedLinese+DumpedLinesf;
            TotalLines=TotalLines+TotalLines1+TotalLines2+TotalLines3+TotalLines4+TotalLines5+TotalLines6+TotalLines7+TotalLines8+TotalLines9+TotalLines0+TotalLinesa+TotalLinesb+TotalLinesc+TotalLinesd+TotalLinese+TotalLinesf;
            WordsChecked=WordsChecked+WordsChecked1+WordsChecked2+WordsChecked3+WordsChecked4+WordsChecked5+WordsChecked6+WordsChecked7+WordsChecked8+WordsChecked9+WordsChecked0+WordsCheckeda+WordsCheckedb+WordsCheckedc+WordsCheckedd+WordsCheckede+WordsC
heckedf;
                                              if (Exact_flag)
                       printf( "Kazahana: Dumped xgrams: %s\n", _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                                              else
                       printf( "Kazahana: Total/Checked/Dumped xgrams: %s/%s/%s\n", _ui64toaKAZEcomma(TotalLines, llTOaDigits3, 10), _ui64toaKAZEcomma(WordsChecked, llTOaDigits2, 10), _ui64toaKAZEcomma(DumpedLines, llTOaDigits, 10) );
                       printf( "Kazahana: Performance: %s KB/clock\n", _ui64toaKAZEcomma((size_inLINESIXFOUR>>10)/((long)(clocks2 - clocks1 + 1)), llTOaDigits, 10) );
                                              if (Exact_flag) {} else
                       printf( "Kazahana: Performance: %s xgrams/clock\n", _ui64toaKAZEcomma((TotalLines)/((long)(clocks2 - clocks1 + 1)), llTOaDigits, 10) ); // CLOCKS_PER_SEC
                       printf( "Kazahana: Performance: Total/fread() clocks: %s/%s\n", _ui64toaKAZEcomma((long)(clocks2 - clocks1 + 1), llTOaDigits, 10), _ui64toaKAZEcomma((long)FREADclocks, llTOaDigits2, 10) );
                       printf( "Kazahana: Performance: I/O time, i.e. fread() time, is %s percents\n", _ui64toaKAZEcomma(FREADclocks*100/(long)(clocks2 - clocks1 + 1), llTOaDigits, 10) );
#if defined(_icl_mumbo_jumbo_)
                       printf( "Kazahana: Performance: RDTSC I/O time, i.e. fread() time, is %s ticks\n", _ui64toaKAZEcomma(ticksTOTAL, llTOaDigits, 10) );
#endif
                       printf( "Kazahana: Done.\n" );
            }
            exit (0);
}


// Test on laptop with Q9550s 2833MHz, 4/4 cores/threads, Windows 7 64bit:
/*
D:\_KAZE\GameraWikipediaWiktionary>type Kazahana_2014-Dec-04\Kazahana_compile_GCC.bat
gcc -O3 -funroll-loops -static -o Kazahana_Hexadecad_GCC_472 Kazahana_r1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER+EX+CS_fix_DEFINE.c -fopenmp -DCommence_OpenMP -D_FILE_OFFSET_BITS=64 -D_gcc_mumbo_jumbo_
gcc -O3 -funroll-loops -static -o Kazahana_Monad_GCC_472 Kazahana_r1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER+EX+CS_fix_DEFINE.c -fopenmp -D_FILE_OFFSET_BITS=64 -D_gcc_mumbo_jumbo_

D:\_KAZE\GameraWikipediaWiktionary>type Kazahana_2014-Dec-04\Kazahana_compile_Intel12_64bit.bat
icl /O3 /arch:SSE2 /QxSSE2 /Qunroll /MT Kazahana_r1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER+EX+CS_fix_DEFINE.c /FAcs /FeKazahana_r1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER+EX+CS_fix_DEFINE_HEXADECAD-Threads_IntelV12_SSE2_64bit /Qopenmp
/Qopenmp-link:static -DCommence_OpenMP -D_icl_mumbo_jumbo_
icl /O3 /arch:SSE2 /QxSSE2 /Qunroll /MT Kazahana_r1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER+EX+CS_fix_DEFINE.c /FeKazahana_r1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER+EX+CS_fix_DEFINE_MONAD-Thread_IntelV12_SSE2_64bit -D_icl_mumbo_jumbo_

D:\_KAZE\GameraWikipediaWiktionary>timer32.exe Kazahana_Hexadecad_GCC_472.exe 4e "Silvestor Staloune" enwiki-20141008-pages-articles.xml 11263
Kazahana, a superfast exact & wildcards & Levenshtein Distance (Wagner-Fischer) searcher, r. 1-++fix+nowait_critical_nixFIX_Wolfram+fixITER+EX+CS_fix, copyleft Kaze 2014-Nov-19.
Pattern: Silvestor Staloune
omp_get_num_procs( ) = 4
omp_get_max_threads( ) = 4
```

```
Enforcing HEXADECAD i.e. hexadecuple-threads ...
Allocating Master-Buffer 11263KB ... OK
\; 00,000,001,376 bytes/clock
Kazahana: Total/Checked/Dumped xgrams: 800,855,553/342,059,464,575/2,106
Kazahana: Performance: 1 KB/clock
Kazahana: Performance: 21 xgrams/clock
Kazahana: Performance: Total/fread() clocks: 36,459,222/1,379,563
Kazahana: Performance: I/O time, i.e. fread() time, is 3 percents
Kazahana: Done.

Kernel  Time =    38.345 =    0%
User    Time =136250.493 =  373%
Process Time =136288.838 =  373%    Virtual  Memory =     14 MB
Global  Time = 36460.185 =  100%    Physical Memory =     16 MB

D:\_KAZE\GameraWikipediaWiktionary>dir Kazahana.txt
 Volume in drive D is S640_Vol5
 Volume Serial Number is 5861-9E6C

 Directory of D:\_KAZE\GameraWikipediaWiktionary

12/03/2014  01:10 PM         1,064,420 Kazahana.txt
               1 File(s)      1,064,420 bytes
               0 Dir(s)  63,694,749,696 bytes free

D:\_KAZE\GameraWikipediaWiktionary>timer32.exe Kazahana_r1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER+EX+CS_fix_DEFINE_HEXADECAD-Threads_IntelV12_SSE2_64bit 4e "Silvestor Staloune" enwiki-20141008-pages-articles.xml 11263
Kazahana, a superfast exact & wildcards & Levenshtein Distance (Wagner-Fischer) searcher, r. 1-++fix+nowait_critical_nixFIX_Wolfram+fixITER+EX+CS_fix_DEFINE, copyleft Kaze 2014-Dec-03.
Pattern: Silvestor Staloune
omp_get_num_procs( ) = 4
omp_get_max_threads( ) = 4
Enforcing HEXADECAD i.e. hexadecuple-threads ...
Allocating Master-Buffer 11263KB ... OK
\; Speed: 00,000,002,001 bytes/clock; Traversed: 50,144,448,379 bytes
Kazahana: Total/Checked/Dumped xgrams: 800,855,553/342,059,464,575/2,106
Kazahana: Performance: 1 KB/clock
Kazahana: Performance: 31 xgrams/clock
Kazahana: Performance: Total/fread() clocks: 25,073,428/602,292
Kazahana: Performance: I/O time, i.e. fread() time, is 2 percents
Kazahana: Performance: RDTSC I/O time, i.e. fread() time, is 1,704,219,997,078 ticks
Kazahana: Done.

Kernel  Time =   284.670 =    1%
User    Time = 92233.204 =  367%
Process Time = 92517.875 =  368%    Virtual  Memory =     17 MB
Global  Time = 25073.682 =  100%    Physical Memory =     16 MB

D:\_KAZE\GameraWikipediaWiktionary>dir Kazahana.txt
 Volume in drive D is S640_Vol5
 Volume Serial Number is 5861-9E6C

 Directory of D:\_KAZE\GameraWikipediaWiktionary

12/04/2014  08:51 AM         1,064,420 Kazahana.txt
               1 File(s)      1,064,420 bytes
               0 Dir(s)  67,609,645,056 bytes free

D:\_KAZE\GameraWikipediaWiktionary>
*/

D:\_KAZE\GameraWikipediaWiktionary\Kazahana_2014-Dec-04>dir

12/04/2014  10:47 AM               394 Kazahana_compile_GCC.bat
12/04/2014  10:47 AM               553 Kazahana_compile_Intel12_32bit.bat
12/04/2014  10:47 AM               553 Kazahana_compile_Intel12_64bit.bat
12/04/2014  10:47 AM           278,041 Kazahana_Hexadecad_GCC_472.exe
12/04/2014  10:47 AM           260,419 Kazahana_Monad_GCC_472.exe
12/04/2014  10:47 AM        10,197,478 Kazahana_r1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER+EX+CS_fix_DEFINE.32bit.cod
12/04/2014  10:47 AM        11,050,900 Kazahana_r1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER+EX+CS_fix_DEFINE.64bit.cod
12/04/2014  10:47 AM         1,005,288 Kazahana_r1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER+EX+CS_fix_DEFINE.c
```

```
12/04/2014  10:47 AM           489,984 Kazahana_r1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER+EX+CS_fix_DEFINE_HEXADECAD-Threads_IntelV12_SSE2_32bit.exe
12/04/2014  10:47 AM           581,120 Kazahana_r1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER+EX+CS_fix_DEFINE_HEXADECAD-Threads_IntelV12_SSE2_64bit.exe
12/04/2014  10:47 AM           174,080 Kazahana_r1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER+EX+CS_fix_DEFINE_MONAD-Thread_IntelV12_SSE2_32bit.exe
12/04/2014  10:47 AM           203,776 Kazahana_r1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER+EX+CS_fix_DEFINE_MONAD-Thread_IntelV12_SSE2_64bit.exe
12/04/2014  10:47 AM             1,604 MokujIN prompt.lnk

D:\_KAZE\GameraWikipediaWiktionary\Kazahana_2014-Dec-04>"Kazahana_r1-++fix+nowait_critical_nixFIX_WolfRAM+fixITER+EX+CS_fix_DEFINE_HEXADECAD-Threads_IntelV12_SSE2_64bit.exe"
Kazahana, a superfast exact & wildcards & Levenshtein Distance (Wagner-Fischer) searcher, r. 1-++fix+nowait_critical_nixFIX_Wolfram+fixITER+EX+CS_fix_DEFINE, copyleft Kaze 2014-Dec-04.
Usage: Kazahana [AtMostLevenshteinDistance][e] string textualfile MasterBufferSize
Note0: MasterBufferSize is in KB, consider 1024, 3072, 7168 or bigger. Two additional flags were mapped on this value: all dump
       lines (except fuzzy's) will have/lack pattern-source info when the number is even/odd respectively, see Examples #5 and #6.
Note1: There are three regimes: exact, wildcards and fuzzy searches. First two kick in when 3 parameters are given, fuzzy when 4.
Note2: What decides whether exact or wildcards? Of course presence of at least one wildcard. To see exact search see Example #4.
Note3: Exact search hits with 'Railgun_Sekireigan_Wolfram'.
Note4a: Incoming string is automatically lowercased for fuzzy searches i.e. they are case insensitive.
Note4b: Incoming string is NOT automatically lowercased for wildcards searches when MasterBufferSize ends in 0..4 i.e. they are case sensitive.
Note4c: Incoming string is automatically lowercased for wildcards searches when MasterBufferSize ends in 5..9 i.e. they are case insensitive.
Note5: Incoming string could be up to 26208/156 chars for Exact&Wildcard&ExhaustiveFuzzy/Fuzzy respectively.
Note5a: Since 2013-Nov-21 Levenshtein search exits not when the incoming line is bigger than 156 chars, now it just skips longer lines.
Note5b: Since 2013-Dec-05 Levenshtein search can be EXHAUSTIVE if LD is postfixed with 'e'.
Note6: Incoming textualfile could be bigger than 4GB.
Note7: Each line should end with [CR]LF, that is Windows or/and UNIX style.
Note8: The dump goes to Kazahana.txt file.
Note9a: Nine SLOW wildcards are available:
        wildcard '*' any character(s) or empty,
        wildcard '.' any ALPHA character(s) or empty,
        wildcard '`' any NON-ALPHA character(s) or empty,
        wildcard '@'/'#' any character {or empty}/{and not empty},
        wildcard '^'/'$' any ALPHA character {or empty}/{and not empty},
        wildcard '|'/'~' any NON-ALPHA character {or empty}/{and not empty}.
Note9b: Two FAST wildcards are available:
        wildcard '&' any character(s) or empty,
        wildcard '+' any character and not empty.
Note9c: Don't mix SLOW and FAST, the SLOW overrides the FAST, i.e. presence of at least one of the 9 wildcards cancels FAST mode.
Example1: E:\>Kazahana 0 ramjet MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd 1536
Example2: E:\>Kazahana 3 psychedlicize MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd 1536
Example3: E:\>Kazahana "psyched^^^^^^ize^" MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd 1536
Example4: E:\>Kazahana "metal fatigue" enwiki-20121201-pages-articles.xml 7168
Example5: E:\>Kazahana "out^^^^^^^^^^^^ize*" MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd 1536
          E:\>type Kazahana.txt
          [out^^^^^^^^^^^^ize*] outhyperbolize /MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd/
          [out^^^^^^^^^^^^ize*] outsize /MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd/
          [out^^^^^^^^^^^^ize*] outsized /MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd/
          [out^^^^^^^^^^^^ize*] outstrategize /MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd/
          [out^^^^^^^^^^^^ize*] outtyrannize /MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd/
Example6: E:\>Kazahana "out^^^^^^^^^^^^ize*" MASAKARI_General-Purpose_Grade_English_Wordlist_r3_316423_words.wrd 1537
          E:\>type Kazahana.txt
          outhyperbolize
          outsize
          outsized
          outstrategize
          outtyrannize
Example7: E:\>Kazahana 2e edelvais MASAKARI_General-Purpose_Grade_English_Wordlist.wrd 1024
          E:\>type Kazahana.txt
          bordelais
          bordelaise
          edelweiss
          edelweisses
          foredevised
          predellas
          psychedelicism
Info1: One second seems to have 1,000 clocks.
Info2: This CPU seems to be working at 2,829 MHz.

D:\_KAZE\GameraWikipediaWiktionary\Kazahana_2014-Dec-04>
```