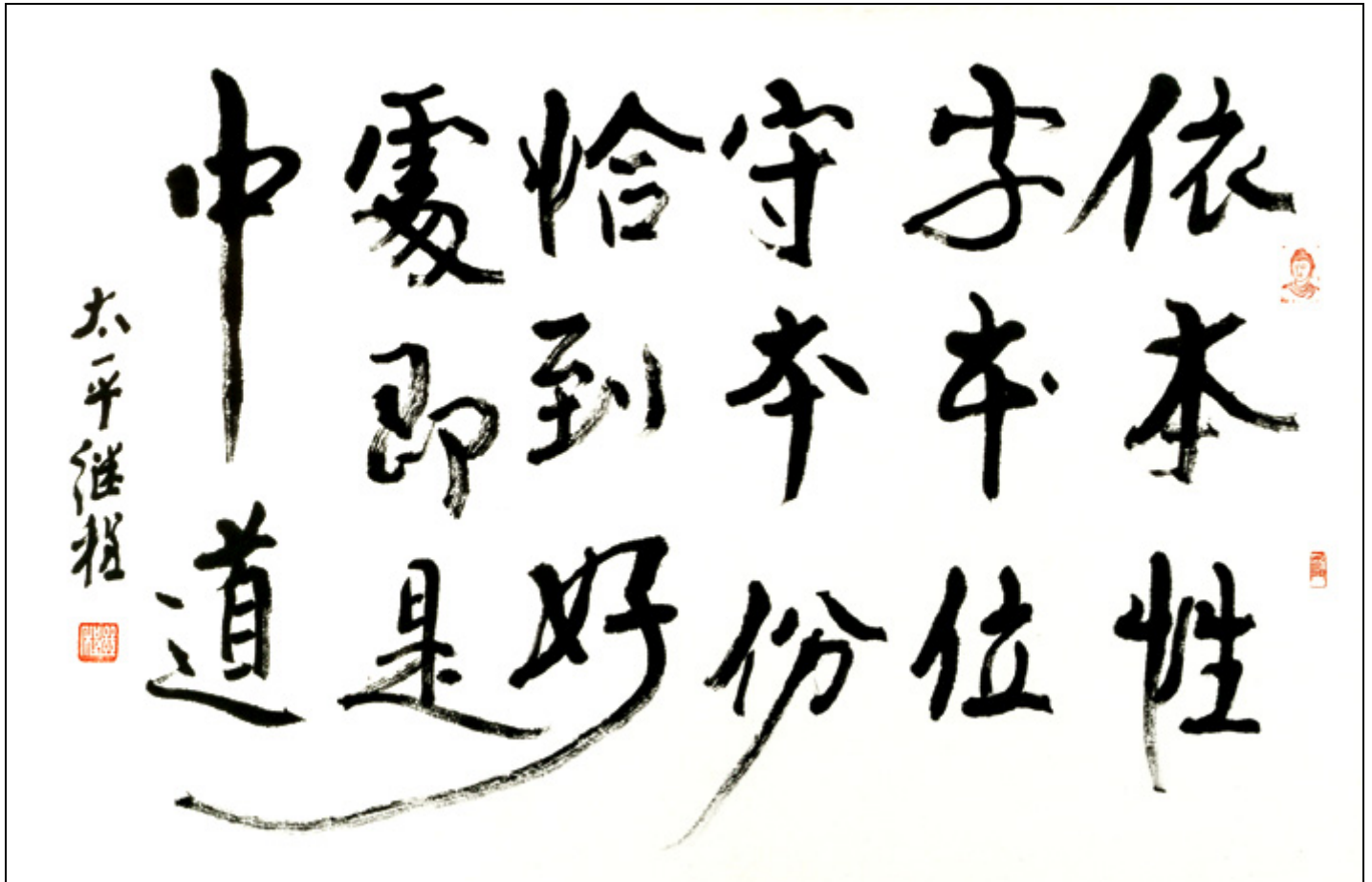


# 中道

## NAKAMICHI



```
0001 // Nakamichi is 100% FREE LZSS SUPERFAST decompressor.
0002
0003 // Nakamichi, revision 1-RSSBO_1GB_wordfetcher_TRIAD_NOMemcpy, written by Kaze, babealicious suggestion by m^2 enforced.
0004 // Change #1: 'memcpy' replaced by GP/XMM/YMM TRIADS.
0005 // Nakamichi, revision 1-RSSBO_1GB_wordfetcher_TRIAD, written by Kaze.
0006 // Change #1: Decompression fetches WORD instead of BYTE+BYTE.
0007 // Change #2: Decompression stores three times 64bit instead of memcpy() for all transfers <=24 bytes.
0008 // Change #3: Fifteenth bit is used and then unused, 16KB -> 32KB -> 16KB.
0009 // 32KB window disappoints speedwise, also sizewise:
0010 /*
0011 D:\_KAZE\_KAZE_GOLD\Nakamichi_projectOLD\Nakamichi_vs_Yappy>Nakamichi_r1-RSSBO_1GB_15bit_wordfetcher.exe enwik8
0012 Nakamichi, revision 1-RSSBO_1GB_15bit_wordfetcher, written by Kaze, based on Nobuo Ito's LZSS source.
0013 Compressing 100000000 bytes ...
0014 -; Each rotation means 128KB are encoded; Done 100%
0015 RAM-to-RAM performance: 130 KB/s.
0016
0017 D:\_KAZE\_KAZE_GOLD\Nakamichi_projectOLD\Nakamichi_vs_Yappy>Nakamichi_r1-RSSBO_1GB_15bit_wordfetcher.exe enwik8.Nakamichi
0018 Nakamichi, revision 1-RSSBO_1GB_15bit_wordfetcher, written by Kaze, based on Nobuo Ito's LZSS source.
0019 Decompressing 65693566 bytes ...
0020 RAM-to-RAM performance: 358 MB/s.
0021
0022 D:\_KAZE\_KAZE_GOLD\Nakamichi_projectOLD\Nakamichi_vs_Yappy>Nakamichi_r1-RSSBO_1GB_15bit_wordfetcher.exe enwik9
0023 Nakamichi, revision 1-RSSBO_1GB_15bit_wordfetcher, written by Kaze, based on Nobuo Ito's LZSS source.
0024 Compressing 1000000000 bytes ...
0025 /; Each rotation means 128KB are encoded; Done 100%
0026 RAM-to-RAM performance: 150 KB/s.
0027
0028 D:\_KAZE\_KAZE_GOLD\Nakamichi_projectOLD\Nakamichi_vs_Yappy>Nakamichi_r1-RSSBO_1GB_15bit_wordfetcher.exe enwik9.Nakamichi
0029 Nakamichi, revision 1-RSSBO_1GB_15bit_wordfetcher, written by Kaze, based on Nobuo Ito's LZSS source.
0030 Decompressing 609319736 bytes ...
0031 RAM-to-RAM performance: 379 MB/s.
0032 */
```

```

0033 // 1-RSSBO_1GB vs 1-RSSBO_1GB_15bit_wordfetcher (16KB/32KB respectively):
0034 // 069,443,065 vs 065,693,566
0035 // 641,441,055 vs 609,319,736
0036
0037 // Nakamichi, revision 1-RSSBO_1GB, written by Kaze.
0038 // Based on Nobuo Ito's source, thanks Ito.
0039 // The main goal of Nakamichi is to allow supersimple and superfast decoding for English x-grams (mainly) in pure C, or not, heh-heh.
0040 // Natively Nakamichi is targeted as 64bit tool with 16 threads, helping Kazahana to traverse faster when I/O is not superior.
0041 // In short, Nakamichi is intended as x-gram decompressor.
0042
0043 // Eightfold Path ~ the Buddhist path to nirvana, comprising eight aspects in which an aspirant must become practised;
0044 // eightfold way ~ (Physics), the grouping of hadrons into supermultiplets by means of SU(3)); (b) adverb to eight times the number or quantity: OE.
0045
0046 // Note1: Fifteenth bit is not used, making the window wider by 1bit i.e. 32KB is not tempting, rather I think to use it as a flag: 8bytes/16bytes.
0047 // Note2: English x-grams are as English texts but more redundant, in other words they are phraselists in most cases, sometimes wordlists.
0048 // Note3: On OSHO.TXT, being a typical English text, Nakamichi's compression ratio is among the worsts:
0049 // 206,908,949 OSHO.TXT
0050 // 125,022,859 OSHO.TXT.Nakamichi
0051 // It struggles with English texts but decompression speed is quite sweet (Core 2 T7500 2200MHz, 32bit code):
0052 // Nakamichi, revision 1-, written by Kaze.
0053 // Decompressing 125022859 bytes ...
0054 // RAM-to-RAM performance: 477681 KB/s.
0055 // Note4: Also I wanted to see how my 'Railgun_Swampshine_BailOut', being a HEAVYGUN i.e. with big overhead and latency, hits in a real-world application.
0056
0057 // Quick notes on PAGODAs (the padded x-gram lists):
0058 // Every single word in English has its own PAGODA, in example below 'on' PAGODA is given (Kazahana_on.PAGODA-order-5.txt):
0059 // PAGODA order 5 (i.e. with 5 tiers) has 5*(5+1)/2=15 subtiers, they are concatenated and space-padded in order to form the pillar 'on':
0060 /*
0061 D:\_KAZE\Nakamichi_r1-RSSBO>dir \_Gw\ka*
0062
0063 04/12/2014 05:07 AM          14 Kazahana_on.1-1.txt
0064 04/12/2014 05:07 AM       1,635,389 Kazahana_on.2-1.txt
0065 04/12/2014 05:07 AM       1,906,734 Kazahana_on.2-2.txt
0066 04/12/2014 05:07 AM      10,891,415 Kazahana_on.3-1.txt
0067 04/12/2014 05:07 AM      15,797,703 Kazahana_on.3-2.txt
0068 04/12/2014 05:07 AM      20,419,280 Kazahana_on.3-3.txt
0069 04/12/2014 05:07 AM      22,141,823 Kazahana_on.4-1.txt
0070 04/12/2014 05:07 AM      36,002,113 Kazahana_on.4-2.txt
0071 04/12/2014 05:07 AM      33,236,772 Kazahana_on.4-3.txt
0072 04/12/2014 05:07 AM      33,902,425 Kazahana_on.4-4.txt
0073 04/12/2014 05:07 AM      24,795,989 Kazahana_on.5-1.txt
0074 04/12/2014 05:07 AM      30,766,220 Kazahana_on.5-2.txt
0075 04/12/2014 05:07 AM      38,982,816 Kazahana_on.5-3.txt
0076 04/12/2014 05:07 AM      38,089,575 Kazahana_on.5-4.txt
0077 04/12/2014 05:07 AM      34,309,057 Kazahana_on.5-5.txt
0078 04/12/2014 05:07 AM      846,351,894 Kazahana_on.PAGODA-order-5.txt
0079
0080 D:\_KAZE\Nakamichi_r1-RSSBO>type \_Gw\Kazahana_on.1-1.txt
0081 9,999,999      on
0082
0083 D:\_KAZE\Nakamichi_r1-RSSBO>type \_Gw\Kazahana_on.2-1.txt
0084 9,999,999      on_the
0085 1,148,054      on_his
0086 0,559,694      on_her
0087 0,487,856      on_this
0088 0,399,485      on_your
0089 0,381,570      on_my
0090 0,367,282      on_their
0091 ...
0092
0093 D:\_KAZE\Nakamichi_r1-RSSBO>type \_Gw\Kazahana_on.2-2.txt
0094 0,545,191      based_on
0095 0,397,408      and_on
0096 0,334,266      go_on
0097 0,329,561      went_on
0098 0,263,035      was_on
0099 0,246,332      it_on
0100 0,229,041      down_on
0101 0,202,151      going_on
0102 ...
0103
0104 D:\_KAZE\Nakamichi_r1-RSSBO>type \_Gw\Kazahana_on.5-5.txt
0105 0,083,564      foundation_oshos_books_on
0106 0,012,404      medium_it_may_be_on
0107 0,012,354      if_you_received_it_on
0108 0,012,152      medium_they_may_be_on
0109 0,012,144      agree_to_also_provide_on
0110 0,012,139      a_united_states_copyright_on
0111 0,008,067      we_are_constantly_working_on
0112 0,008,067      questions_we_have_received_on
0113 0,006,847      file_was_first_posted_on
0114 0,006,441      of_we_are_already_on
0115 0,006,279      you_received_this_ebook_on
0116 0,005,865      you_received_this_etext_on
0117 0,005,833      to_keep_an_eye_on
0118 ...
0119
0120 D:\_KAZE\Nakamichi_r1-RSSBO>dir
0121

```

```
0122 04/12/2014 05:07 AM      846,351,894 Kazahana_on.PAGODA-order-5.txt
0123
0124 D:\_KAZE\Nakamichi_r1-RSSBO>Nakamichi.exe Kazahana_on.PAGODA-order-5.txt
0125 Nakamichi, revision 1-RSSBO, written by Kaze.
0126 Compressing 846351894 bytes ...
0127 /; Each rotation means 128KB are encoded; Done 100%
0128 RAM-to-RAM performance: 512 KB/s.
0129
0130 D:\_KAZE\Nakamichi_r1-RSSBO>dir
0131
0132 04/12/2014 05:07 AM      846,351,894 Kazahana_on.PAGODA-order-5.txt
0133 04/15/2014 06:30 PM      293,049,398 Kazahana_on.PAGODA-order-5.txt.Nakamichi
0134
0135 D:\_KAZE\Nakamichi_r1-RSSBO>Nakamichi.exe Kazahana_on.PAGODA-order-5.txt.Nakamichi
0136 Nakamichi, revision 1-RSSBO, written by Kaze.
0137 Decompressing 293049398 bytes ...
0138 RAM-to-RAM performance: 607 MB/s.
0139
0140 D:\_KAZE\Nakamichi_r1-RSSBO>Yappy.exe Kazahana_on.PAGODA-order-5.txt 4096
0141 YAPPY: [b 4K] bytes 846351894 -> 191149889 22.6% comp 33.8 MB/s uncomp 875.4 MB/s
0142
0143 D:\_KAZE\Nakamichi_r1-RSSBO>Yappy.exe Kazahana_on.PAGODA-order-5.txt 8192
0144 YAPPY: [b 8K] bytes 846351894 -> 184153244 21.8% comp 35.0 MB/s uncomp 898.3 MB/s
0145
0146 D:\_KAZE\Nakamichi_r1-RSSBO>Yappy.exe Kazahana_on.PAGODA-order-5.txt 16384
0147 YAPPY: [b 16K] bytes 846351894 -> 180650931 21.3% comp 28.8 MB/s uncomp 906.4 MB/s
0148
0149 D:\_KAZE\Nakamichi_r1-RSSBO>Yappy.exe Kazahana_on.PAGODA-order-5.txt 32768
0150 YAPPY: [b 32K] bytes 846351894 -> 178902966 21.1% comp 35.0 MB/s uncomp 906.4 MB/s
0151
0152 D:\_KAZE\Nakamichi_r1-RSSBO>Yappy.exe Kazahana_on.PAGODA-order-5.txt 65536
0153 YAPPY: [b 64K] bytes 846351894 -> 178027899 21.0% comp 34.5 MB/s uncomp 914.6 MB/s
0154
0155 D:\_KAZE\Nakamichi_r1-RSSBO>Yappy.exe Kazahana_on.PAGODA-order-5.txt 131072
0156 YAPPY: [b 128K] bytes 846351894 -> 177591807 21.0% comp 34.9 MB/s uncomp 906.4 MB/s
0157
0158 D:\_KAZE\Nakamichi_r1-RSSBO>
0159 */
0160
0161 #include <stdio.h>
0162 #include <stdlib.h>
0163 #include <stdint.h> // uint64_t needed
0164 #include <time.h>
0165 #include <string.h>
0166
0167 #include <emmintrin.h> // SSE2 intrinsics
0168 #include <smmmintrin.h> // SSE4.1 intrinsics
0169 #include <immintrin.h> // AVX intrinsics
0170 // #include <zmmmintrin.h> // AVX2 intrinsics, definitions and declarations for use with 512-bit compiler intrinsics.
0171
0172 void SlowCopy128bit (const char *SOURCE, char *TARGET) { __mm_storeu_si128((__m128i *) (TARGET), __mm_loadu_si128((const __m128i *) (SOURCE))); }
0173 /*
0174  * Move Unaligned Packed Integer Values
0175  * **** VMOVDQU ymm1, m256
0176  * **** VMOVDQU m256, ymm1
0177  * Moves 256 bits of packed integer values from the source operand to the
0178  * destination
0179  */
0180 //extern __m256i __ICL_INTRINCC __mm256_loadu_si256(__m256i const *);
0181 //extern void __ICL_INTRINCC __mm256_storeu_si256(__m256i *, __m256i);
0182 void SlowCopy256bit (const char *SOURCE, char *TARGET) { __mm256_storeu_si256((__m256i *) (TARGET), __mm256_loadu_si256((const __m256i *) (SOURCE))); }
0183 //extern __m512i __ICL_INTRINCC __mm512_loadu_si512(void const*);
0184 //extern void __ICL_INTRINCC __mm512_storeu_si512(void*, __m512i);
0185 //void SlowCopy512bit (const char *SOURCE, char *TARGET) { __mm512_storeu_si512((__m512i *) (TARGET), __mm512_loadu_si512((const __m512i *) (SOURCE))); }
0186
0187 // During compilation use one of these, the granularity of the padded 'memcpy', 4x2x8/2x2x16/1x2x32/1x1x64 respectively as GP/XMM/YMM/ZMM, the maximum literal
length reduced from 127 to 63:
0188 // #define _N_GP
0189 // #define _N_XMM
0190 // #define _N_YMM
0191 // #define _N_ZMM
0192
0193 // icl /o3 Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif.c -D_N_GP /Facs
0194 // ren Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif.cod Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif_GP.cod
0195 // ren Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif.exe Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif_GP.exe
0196 // icl /o3 /QxSSE2 Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif.c -D_N_XMM /Facs
0197 // ren Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif.cod Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif_XMM.cod
0198 // ren Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif.exe Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif_XMM.exe
0199 // icl /o3 /QxAVX Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif.c -D_N_YMM /Facs
0200 // ren Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif.cod Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif_YMM.cod
0201 // ren Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif.exe Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif_YMM.exe
0202 // icl /o3 /QxCORE-AVX2 Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif.c -D_N_ZMM /Facs
0203 // ren Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif.cod Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif_ZMM.cod
0204 // ren Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif.exe Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_NoMemcpy_Noif_ZMM.exe
0205
0206 #ifndef NULL
0207 #define NULL ((void*)0)
0208 #endif
0209
```

```

0210 // Comment it to see how slower 'BruteForce' is, for wikipedia 100MB the ratio is 41KB/s versus 197KB/s.
0211 #define ReplaceBruteForceWithRailgunSwampshineBailOut
0212
0213 void SearchIntoSlidingwindow(unsigned int* retIndex, unsigned int* retMatch, char* refStart, char* refEnd, char* encStart, char* encEnd);
0214 unsigned int SlidingWindowVsLookAheadBuffer(char* refStart, char* refEnd, char* encStart, char* encEnd);
0215 unsigned int Compress(char* ret, char* src, unsigned int srcSize);
0216 unsigned int Decompress(char* ret, char* src, unsigned int srcSize);
0217 char * Railgun_Swampshine_BailOut(char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern);
0218 char * Railgun_Doubllet (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern);
0219
0220 // Min_Match_Length=THRESHOLD=4 means 4 and bigger are to be encoded:
0221 #define Min_Match_BAILOUT_Length (8)
0222 #define Min_Match_Length (8)
0223 #define OffsetBITS (14)
0224 #define LengthBITS (1)
0225
0226 //12bit
0227 // #define REF_SIZE (4095+Min_Match_Length)
0228 #define REF_SIZE ( ((1<<OffsetBITS)-1) + Min_Match_Length )
0229 //3bit
0230 // #define ENC_SIZE (7+Min_Match_Length)
0231 #define ENC_SIZE ( ((1<<LengthBITS)-1) + Min_Match_Length )
0232
0233 int main( int argc, char *argv[] ) {
0234     FILE *fp;
0235     int SourceSize;
0236     int TargetSize;
0237     char* SourceBlock=NULL;
0238     char* TargetBlock=NULL;
0239     char* Nakamichi = ".Nakamichi\0";
0240     char NewFileName[256];
0241     clock_t clocks1, clocks2;
0242
0243     printf("Nakamichi, revision 1-RSSBO_1GB_Wordfetcher_TRIAD_Nomemcpy, written by Kaze, based on Nobuo Ito's LZSS source, babealicious suggestion by m^2
enforced.\n");
0244     if (argc==1) {
0245         printf("Usage: Nakamichi filename\n"); exit(13);
0246     }
0247     if ((fp = fopen(argv[1], "rb")) == NULL) {
0248         printf("Nakamichi: Can't open '%s' file.\n", argv[1]); exit(13);
0249     }
0250     fseek(fp, 0, SEEK_END);
0251     SourceSize = ftell(fp);
0252     fseek(fp, 0, SEEK_SET);
0253     // If filename ends in '.Nakamichi' then mode is decompression otherwise compression.
0254     if (strcmp(argv[1]+(strlen(argv[1])-strlen(Nakamichi)), Nakamichi) == 0) {
0255         SourceBlock = (char*)malloc(SourceSize+512);
0256         //TargetBlock = (char*)malloc(5*SourceSize+512);
0257         TargetBlock = (char*)malloc(1024*1024*1024+512);
0258         fread(SourceBlock, 1, SourceSize, fp);
0259         fclose(fp);
0260         printf("Decompressing %d bytes ...\n", SourceSize );
0261         clocks1 = clock();
0262         TargetSize = Decompress(TargetBlock, SourceBlock, SourceSize);
0263         clocks2 = clock();
0264         printf("RAM-to-RAM performance: %d MB/s.\n", ((TargetSize/(clocks2 - clocks1 + 1))*(long)1000)>>20);
0265         strcpy(NewFileName, argv[1]);
0266         *( NewFileName + strlen(argv[1])-strlen(Nakamichi) ) = '\0';
0267     } else {
0268         SourceBlock = (char*)malloc(SourceSize+512);
0269         TargetBlock = (char*)malloc(SourceSize+512);
0270         fread(SourceBlock, 1, SourceSize, fp);
0271         fclose(fp);
0272         printf("Compressing %d bytes ...\n", SourceSize );
0273         clocks1 = clock();
0274         TargetSize = Compress(TargetBlock, SourceBlock, SourceSize);
0275         clocks2 = clock();
0276         printf("RAM-to-RAM performance: %d KB/s.\n", ((SourceSize/(clocks2 - clocks1 + 1))*(long)1000)>>10);
0277         strcpy(NewFileName, argv[1]);
0278         strcat(NewFileName, Nakamichi);
0279     }
0280     if ((fp = fopen(NewFileName, "wb")) == NULL) {
0281         printf("Nakamichi: Can't write '%s' file.\n", NewFileName); exit(13);
0282     }
0283     fwrite(TargetBlock, 1, TargetSize, fp);
0284     fclose(fp);
0285     free(TargetBlock);
0286     free(SourceBlock);
0287     exit(0);
0288 }
0289
0290 void SearchIntoSlidingwindow(unsigned int* retIndex, unsigned int* retMatch, char* refStart, char* refEnd, char* encStart, char* encEnd){
0291     char* FoundAtPosition;
0292     unsigned int match=0;
0293     *retIndex=0;
0294     *retMatch=0;
0295 #ifdef ReplaceBruteForceWithRailgunSwampshineBailOut
0296     if (refStart < refEnd) {
0297         FoundAtPosition = Railgun_Swampshine_BailOut(refStart, encStart, (uint32_t)(refEnd-refStart), 8);

```

```

0298 //FoundAtPosition = Railgun_Doublet(refStart, encStart, (uint32_t)(refEnd-refStart), 8);
0299 // For bigger windows 'Doublet' is slower:
0300 // Nakamichi, revision 1-RSSBQ_1GB_15bit performance with 'Swampshine':
0301 // Compressing 846351894 bytes ...
0302 // RAM-to-RAM performance: 370 KB/s.
0303 // Nakamichi, revision 1-RSSBQ_1GB_15bit performance with 'Doublet':
0304 // Compressing 846351894 bytes ...
0305 // RAM-to-RAM performance: 213 KB/s.
0306 if (FoundAtPosition!=NULL) {
0307     *retMatch=8;
0308     *retIndex=refEnd-FoundAtPosition;
0309 }
0310 }
0311 #else
0312 while(refStart < refEnd){
0313     match=slidingwindowVsLookAheadBuffer(refStart,refEnd,encStart,encEnd);
0314     if(match > *retMatch){
0315         *retMatch=match;
0316         *retIndex=refEnd-refStart;
0317     }
0318     if(*retMatch >= Min_Match_BAILOUT_Length) break;
0319     refStart++;
0320 }
0321 #endif
0322 }
0323
0324 unsigned int slidingwindowVsLookAheadBuffer( char* refStart, char* refEnd, char* encStart,char* encEnd){
0325     int ret = 0;
0326     while(refStart[ret] == encStart[ret]){
0327         if(&refStart[ret] >= refEnd) break;
0328         if(&encStart[ret] >= encEnd) break;
0329         ret++;
0330         if(ret >= Min_Match_BAILOUT_Length) break;
0331     }
0332     return ret;
0333 }
0334
0335 unsigned int Compress(char* ret, char* src, unsigned int srcSize){
0336     unsigned int srcIndex=0;
0337     unsigned int retIndex=0;
0338     unsigned int index=0;
0339     unsigned int match=0;
0340     unsigned int notMatch=0;
0341     unsigned char* notMatchStart=NULL;
0342     char* refStart=NULL;
0343     char* encEnd=NULL;
0344     int Melnitchka=0;
0345     char *Auberge[4] = {"\\0", "\\0", "-\\0", "\\0\\0"};
0346     int ProgressIndicator;
0347
0348     while(srcIndex < srcSize){
0349         if(srcIndex>=REF_SIZE)
0350             refStart=&src[srcIndex-REF_SIZE];
0351         else
0352             refStart=src;
0353         if(srcIndex>=srcSize-ENC_SIZE)
0354             encEnd=&src[srcSize];
0355         else
0356             encEnd=&src[srcIndex+ENC_SIZE];
0357
0358         SearchIntoSlidingWindow(&index,&match,refStart,&src[srcIndex],&src[srcIndex],encEnd);
0359         //if ( match<Min_Match_Length ) {
0360         //if ( match<Min_Match_Length || match<8 ) {
0361         if ( match==0 ) {
0362             if(notMatch==0){
0363                 notMatchStart=&ret[retIndex];
0364                 retIndex++;
0365             }
0366             else if (notMatch==127) {
0367                 *notMatchStart=(unsigned char)((127)<<1);
0368                 notMatch=0;
0369                 notMatchStart=&ret[retIndex];
0370                 retIndex++;
0371             }
0372             ret[retIndex]=src[srcIndex];
0373             retIndex++;
0374             notMatch++;
0375             srcIndex++;
0376             if ((srcIndex-1) % (1<<17) > srcIndex % (1<<17)) {
0377                 ProgressIndicator = (int)( (srcIndex+1)*(float)100/(srcSize+1) );
0378                 printf("%s; Each rotation means 128KB are encoded; Done %d%%\r", Auberge[Melnitchka++], ProgressIndicator );
0379                 Melnitchka = Melnitchka & 3; // 0 1 2 3: 00 01 10 11
0380             }
0381         } else {
0382             if(notMatch > 0){
0383                 *notMatchStart=(unsigned char)((notMatch)<<1);
0384                 notMatch=0;
0385             }
0386             // -----|

```

```

0387 //          \ /
0388
0389 //ret[retIndex] = 0x80; // Assuming seventh/fifteenth bit is zero i.e. LONG MATCH i.e. Min_Match_BAILOUT_Length*4
0390 //if ( match==Min_Match_BAILOUT_Length ) ret[retIndex] = 0xC0; // 8bit&7bit set, SHORT MATCH if seventh/fifteenth bit is not zero i.e.
Min_Match_BAILOUT_Length
0391 //          / \
0392 // -----|
0393 ret[retIndex] = 0x01; // Assuming seventh/fifteenth bit is zero i.e. LONG MATCH i.e. Min_Match_BAILOUT_Length*4
0394 //if ( match==Min_Match_BAILOUT_Length ) ret[retIndex] = 0x03; // 2bit&1bit set, SHORT MATCH if 2bit is not zero i.e.
Min_Match_BAILOUT_Length
0395 // 1bit+3bits+12bits:
0396 //ret[retIndex] = ret[retIndex] | ((match-Min_Match_Length)<<4);
0397 //ret[retIndex] = ret[retIndex] | (((index-Min_Match_Length) & 0x0F00)>>8);
0398 // 1bit+1bit+14bits:
0399 //ret[retIndex] = ret[retIndex] | ((match-Min_Match_Length)<<(8-(LengthBITS+1))); // No need to set the matchlength
0400 // The fragment below is outrageously ineffective - instead of 8bit&7bit I have to use the lower TWO bits i.e. 2bit&1bit as flags, thus in decompressing one WORD
// can be fetched instead of two BYTE loads followed by SHR by 2.
0401 // -----|
0402 //          \ /
0403 //ret[retIndex] = ret[retIndex] | (((index-Min_Match_Length) & 0x3F00)>>8); // 2+4+8=14
0404 //retIndex++;
0405 //ret[retIndex] = (char)((index-Min_Match_Length) & 0x00FF);
0406 //retIndex++;
0407 //          / \
0408 // -----|
0409 // Now the situation is like LOW:HIGH i.e. FF:3F i.e. 0x3FFF, 16bit&15bit used as flags,
0410 // should become LOW:HIGH i.e. FC:FF i.e. 0xFFFF, 2bit&1bit used as flags.
0411 ret[retIndex] = ret[retIndex] | (((index-Min_Match_Length) & 0x00FF)<<2); // 6+8=14
0412 //ret[retIndex] = ret[retIndex] | (((index-Min_Match_Length) & 0x00FF)<<1); // 7+8=15
0413 retIndex++;
0414 ret[retIndex] = (char)(((index-Min_Match_Length) & 0x3FFF)>>6);
0415 //ret[retIndex] = (char)(((index-Min_Match_Length) & 0x7FFF)>>7);
0416 retIndex++;
0417 //          / \
0418 // -----|
0419 srcIndex+=match;
0420 if ((srcIndex-match) % (1<<17) > srcIndex % (1<<17)) {
0421     ProgressIndicator = (int)( (srcIndex+1)*(float)100/(srcSize+1) );
0422     printf("%s; Each rotation means 128KB are encoded; Done %d%%\r", Auberge[Melnitchka++], ProgressIndicator );
0423     Melnitchka = Melnitchka & 3; // 0 1 2 3: 00 01 10 11
0424 }
0425 }
0426 }
0427 if(notMatch > 0){
0428     *notMatchStart=(unsigned char)((notMatch)<<1);
0429 }
0430 printf("%s; Each rotation means 128KB are encoded; Done %d%%\n", Auberge[Melnitchka], 100 );
0431 return retIndex;
0432 }
0433
0434 unsigned int Decompress(char* ret, char* src, unsigned int srcSize){
0435     unsigned int srcIndex=0;
0436     unsigned int retIndex=0;
0437     unsigned int index=0;
0438     unsigned int match=0;
0439     unsigned int WORDpair;
0440     signed int mm;
0441     unsigned int Fantagirot;
0442
0443     while(srcIndex < srcSize){
0444         //if((unsigned char)src[srcIndex] <= 127){
0445         WORDpair = *(unsigned short int*)&src[srcIndex];
0446         if((WORDpair & 0x01) == 0){
0447
0448             // For enwik9 (as 64bit code) RAM-to-RAM performance: 735 MB/s:
0449             /*
0450                 *(uint64_t*)(ret+retIndex+8*0) = *(uint64_t*)(src+srcIndex+1+8*0);
0451                 SlowCopy128bit((src+srcIndex+1+8*1), (ret+retIndex+8*1));
0452             if ( ((WORDpair & 0xFF)>>1) > 8+16 ) {
0453                 memcpy(&ret[retIndex]+8+16,&src[srcIndex+1]+8+16,((WORDpair & 0xFF)>>1)-(8+16)); // Use padding and replace 'memcpy' with
loop of 4 or 4+4 transfers/stores i.e. *()=DWORD
0454             }
0455             */
0456             // For enwik9 (as 64bit code) RAM-to-RAM performance: 782 MB/s:
0457             /*
0458                 *(uint64_t*)(ret+retIndex+8*0) = *(uint64_t*)(src+srcIndex+1+8*0);
0459                 *(uint64_t*)(ret+retIndex+8*1) = *(uint64_t*)(src+srcIndex+1+8*1);
0460                 *(uint64_t*)(ret+retIndex+8*2) = *(uint64_t*)(src+srcIndex+1+8*2);
0461             if ( ((WORDpair & 0xFF)>>1) > 8*3 ) {
0462                 memcpy(&ret[retIndex]+8*3,&src[srcIndex+1]+8*3,((WORDpair & 0xFF)>>1)-8*3); // Use padding and replace 'memcpy' with loop of
4 or 4+4 transfers/stores i.e. *()=DWORD
0463             }
0464             */
0465             // For enwik9 (as 64bit code) RAM-to-RAM performance: 763 MB/s:
0466             /*
0467             if ( ((WORDpair & 0xFF)>>1) > 8*4 ) {
0468                 memcpy(&ret[retIndex],&src[srcIndex+1],(WORDpair & 0xFF)>>1); // Use padding and replace 'memcpy' with loop of 4 or 4+4
transfers/stores i.e. *()=DWORD
0469             } else {

```

```

0470 //SlowCopy128bit((src+srcIndex+1+16*0), (ret+retIndex+16*0));
0471 //SlowCopy128bit((src+srcIndex+1+16*1), (ret+retIndex+16*1));
0472 *(uint64_t*)(ret+retIndex+8*0) = *(uint64_t*)(src+srcIndex+1+8*0);
0473 *(uint64_t*)(ret+retIndex+8*1) = *(uint64_t*)(src+srcIndex+1+8*1);
0474 *(uint64_t*)(ret+retIndex+8*2) = *(uint64_t*)(src+srcIndex+1+8*2);
0475 *(uint64_t*)(ret+retIndex+8*3) = *(uint64_t*)(src+srcIndex+1+8*3);
0476 // Funny, Nikola Tesla is always right, triads rule, for 'Kazahana_on.PAGODA-order-5.txt.Nakamichi':
0477 // For 8*3 GP : RAM-to-RAM performance: 876 MB/s.
0478 // For 8*4 GP : RAM-to-RAM performance: 833 MB/s.
0479 // For 16*2 XMM: RAM-to-RAM performance: 820 MB/s.
0480 }
0481 */
0482 // For enwik9 (as 64bit code) RAM-to-RAM performance: 793 MB/s:
0483 /*
0484 if ( ((WORDpair & 0xFF)>>1) > 8*3 ) {
0485     memcpy(&ret[retIndex],&src[srcIndex+1],(WORDpair & 0xFF)>>1); // Use padding and replace 'memcpy' with loop of 4 or 4+
0486 transfers/stores i.e. *()=DWORD
0487 } else {
0488     //SlowCopy128bit((src+srcIndex+1+16*0), (ret+retIndex+16*0));
0489     //SlowCopy128bit((src+srcIndex+1+16*1), (ret+retIndex+16*1));
0490     *(uint64_t*)(ret+retIndex+8*0) = *(uint64_t*)(src+srcIndex+1+8*0);
0491     *(uint64_t*)(ret+retIndex+8*1) = *(uint64_t*)(src+srcIndex+1+8*1);
0492     *(uint64_t*)(ret+retIndex+8*2) = *(uint64_t*)(src+srcIndex+1+8*2);
0493     /**(uint64_t*)(ret+retIndex+8*3) = *(uint64_t*)(src+srcIndex+1+8*3);
0494     // Funny, Nikola Tesla is always right, triads rule, for 'Kazahana_on.PAGODA-order-5.txt.Nakamichi':
0495     // For 8*3 GP : RAM-to-RAM performance: 876 MB/s.
0496     // For 8*4 GP : RAM-to-RAM performance: 833 MB/s.
0497     // For 16*2 XMM: RAM-to-RAM performance: 820 MB/s.
0498 }
0499 // For enwik9 (as 64bit code) RAM-to-RAM performance: 727 MB/s:
0500 /*
0501 if ( ((WORDpair & 0xFF)>>1) > 8*2 ) {
0502     memcpy(&ret[retIndex],&src[srcIndex+1],(WORDpair & 0xFF)>>1); // Use padding and replace 'memcpy' with loop of 4 or 4+
0503 transfers/stores i.e. *()=DWORD
0504 } else {
0505     //SlowCopy128bit((src+srcIndex+1+16*0), (ret+retIndex+16*0));
0506     //SlowCopy128bit((src+srcIndex+1+16*1), (ret+retIndex+16*1));
0507     *(uint64_t*)(ret+retIndex+8*0) = *(uint64_t*)(src+srcIndex+1+8*0);
0508     *(uint64_t*)(ret+retIndex+8*1) = *(uint64_t*)(src+srcIndex+1+8*1);
0509     // Funny, Nikola Tesla is always right, triads rule, for 'Kazahana_on.PAGODA-order-5.txt.Nakamichi':
0510     // For 8*3 GP : RAM-to-RAM performance: 876 MB/s.
0511     // For 8*4 GP : RAM-to-RAM performance: 833 MB/s.
0512     // For 16*2 XMM: RAM-to-RAM performance: 820 MB/s.
0513 }
0514 // For enwik9 (as 64bit code) RAM-to-RAM performance: 629 MB/s:
0515 /*
0516 if ( ((WORDpair & 0xFF)>>1) > 8*1 ) {
0517     memcpy(&ret[retIndex],&src[srcIndex+1],(WORDpair & 0xFF)>>1); // Use padding and replace 'memcpy' with loop of 4 or 4+
0518 transfers/stores i.e. *()=DWORD
0519 } else {
0520     //SlowCopy128bit((src+srcIndex+1+16*0), (ret+retIndex+16*0));
0521     //SlowCopy128bit((src+srcIndex+1+16*1), (ret+retIndex+16*1));
0522     *(uint64_t*)(ret+retIndex+8*0) = *(uint64_t*)(src+srcIndex+1+8*0);
0523     // Funny, Nikola Tesla is always right, triads rule, for 'Kazahana_on.PAGODA-order-5.txt.Nakamichi':
0524     // For 8*3 GP : RAM-to-RAM performance: 876 MB/s.
0525     // For 8*4 GP : RAM-to-RAM performance: 833 MB/s.
0526     // For 16*2 XMM: RAM-to-RAM performance: 820 MB/s.
0527 }
0528 */
0529 // The approach below was proposed by m^2, many-many thanks:
0530 // For enwik9 (as 64bit code) RAM-to-RAM performance: 686 MB/s:
0531 mm = ((WORDpair & 0xFF)>>1)/(8*1);
0532 //
0533 // gogogo:
0534 //     *(uint64_t*)(ret+retIndex+8*(0+1*mm)) = *(uint64_t*)(src+srcIndex+1+8*(0+1*mm));
0535 // if (mm--) goto gogogo;
0536 //
0537 // For enwik9 (as 64bit code) RAM-to-RAM performance: 754 MB/s:
0538 mm = ((WORDpair & 0xFF)>>1)/(8*2);
0539 //
0540 // gogogo:
0541 //     *(uint64_t*)(ret+retIndex+8*(0+2*mm)) = *(uint64_t*)(src+srcIndex+1+8*(0+2*mm));
0542 //     *(uint64_t*)(ret+retIndex+8*(1+2*mm)) = *(uint64_t*)(src+srcIndex+1+8*(1+2*mm));
0543 // if (mm--) goto gogogo;
0544 //
0545 // For enwik9 (as 64bit code) RAM-to-RAM performance: 753 MB/s:
0546 mm = ((WORDpair & 0xFF)>>1)/(8*3);
0547 //
0548 // gogogo:
0549 //     *(uint64_t*)(ret+retIndex+8*(0+3*mm)) = *(uint64_t*)(src+srcIndex+1+8*(0+3*mm));
0550 //     *(uint64_t*)(ret+retIndex+8*(1+3*mm)) = *(uint64_t*)(src+srcIndex+1+8*(1+3*mm));
0551 //     *(uint64_t*)(ret+retIndex+8*(2+3*mm)) = *(uint64_t*)(src+srcIndex+1+8*(2+3*mm));
0552 // if (mm--) goto gogogo;
0553 //
0554 // For enwik9 (as 64bit code) RAM-to-RAM performance: 763 MB/s:
0555 for (mm=0; mm <= ((WORDpair & 0xFF)>>1)/(8*3); mm++) {
0556     *(uint64_t*)(ret+retIndex+8*(0+3*mm)) = *(uint64_t*)(src+srcIndex+1+8*(0+3*mm));
0557     *(uint64_t*)(ret+retIndex+8*(1+3*mm)) = *(uint64_t*)(src+srcIndex+1+8*(1+3*mm));

```

```

0556 //      *(uint64_t*)(ret+retIndex+8*(2+3*mm)) = *(uint64_t*)(src+srcIndex+1+8*(2+3*mm));
0557 //  }
0558 //
0559 // For enwik9 (as 64bit code) RAM-to-RAM performance: 773 MB/s:
0560 // mm = ((WORDpair & 0xFF)>>1)/(8*4);
0561 // gogogo:
0562 //      *(uint64_t*)(ret+retIndex+8*(0+4*mm)) = *(uint64_t*)(src+srcIndex+1+8*(0+4*mm));
0563 //      *(uint64_t*)(ret+retIndex+8*(1+4*mm)) = *(uint64_t*)(src+srcIndex+1+8*(1+4*mm));
0564 //      *(uint64_t*)(ret+retIndex+8*(2+4*mm)) = *(uint64_t*)(src+srcIndex+1+8*(2+4*mm));
0565 //      *(uint64_t*)(ret+retIndex+8*(3+4*mm)) = *(uint64_t*)(src+srcIndex+1+8*(3+4*mm));
0566 // if (mm--) goto gogogo;
0567 //
0568 // For enwik9 (as 64bit code) RAM-to-RAM performance: 727 MB/s:
0569 // mm = ((WORDpair & 0xFF)>>1)/(8*5);
0570 // gogogo:
0571 //      *(uint64_t*)(ret+retIndex+8*(0+5*mm)) = *(uint64_t*)(src+srcIndex+1+8*(0+5*mm));
0572 //      *(uint64_t*)(ret+retIndex+8*(1+5*mm)) = *(uint64_t*)(src+srcIndex+1+8*(1+5*mm));
0573 //      *(uint64_t*)(ret+retIndex+8*(2+5*mm)) = *(uint64_t*)(src+srcIndex+1+8*(2+5*mm));
0574 //      *(uint64_t*)(ret+retIndex+8*(3+5*mm)) = *(uint64_t*)(src+srcIndex+1+8*(3+5*mm));
0575 //      *(uint64_t*)(ret+retIndex+8*(4+5*mm)) = *(uint64_t*)(src+srcIndex+1+8*(4+5*mm));
0576 // if (mm--) goto gogogo;
0577 //
0578 // For enwik9 (as 64bit code) RAM-to-RAM performance: 664 MB/s:
0579 // mm = ((WORDpair & 0xFF)>>1)/(8*8);
0580 // gogogo:
0581 //      *(uint64_t*)(ret+retIndex+8*(0+8*mm)) = *(uint64_t*)(src+srcIndex+1+8*(0+8*mm));
0582 //      *(uint64_t*)(ret+retIndex+8*(1+8*mm)) = *(uint64_t*)(src+srcIndex+1+8*(1+8*mm));
0583 //      *(uint64_t*)(ret+retIndex+8*(2+8*mm)) = *(uint64_t*)(src+srcIndex+1+8*(2+8*mm));
0584 //      *(uint64_t*)(ret+retIndex+8*(3+8*mm)) = *(uint64_t*)(src+srcIndex+1+8*(3+8*mm));
0585 //      *(uint64_t*)(ret+retIndex+8*(4+8*mm)) = *(uint64_t*)(src+srcIndex+1+8*(4+8*mm));
0586 //      *(uint64_t*)(ret+retIndex+8*(5+8*mm)) = *(uint64_t*)(src+srcIndex+1+8*(5+8*mm));
0587 //      *(uint64_t*)(ret+retIndex+8*(6+8*mm)) = *(uint64_t*)(src+srcIndex+1+8*(6+8*mm));
0588 //      *(uint64_t*)(ret+retIndex+8*(7+8*mm)) = *(uint64_t*)(src+srcIndex+1+8*(7+8*mm));
0589 // if (mm--) goto gogogo;
0590 //
0591 // For enwik9 (as 64bit code) RAM-to-RAM performance: 702 MB/s:
0592 // mm = ((WORDpair & 0xFF)>>1)/(16*1);
0593 // gogogo:
0594 //      SlowCopy128bit((src+srcIndex+1+16*(0+1*mm)), (ret+retIndex+16*(0+1*mm)));
0595 // if (mm--) goto gogogo;
0596 //
0597 // For enwik9 (as 64bit code) RAM-to-RAM performance: 710 MB/s:
0598 // mm = ((WORDpair & 0xFF)>>1)/(16*2);
0599 // gogogo:
0600 //      SlowCopy128bit((src+srcIndex+1+16*(0+2*mm)), (ret+retIndex+16*(0+2*mm)));
0601 //      SlowCopy128bit((src+srcIndex+1+16*(1+2*mm)), (ret+retIndex+16*(1+2*mm)));
0602 // if (mm--) goto gogogo;
0603 //
0604 // For enwik9 (as 64bit code) RAM-to-RAM performance: 593 MB/s:
0605 // mm = ((WORDpair & 0xFF)>>1)/(16*4);
0606 // gogogo:
0607 //      SlowCopy128bit((src+srcIndex+1+16*(0+4*mm)), (ret+retIndex+16*(0+4*mm)));
0608 //      SlowCopy128bit((src+srcIndex+1+16*(1+4*mm)), (ret+retIndex+16*(1+4*mm)));
0609 //      SlowCopy128bit((src+srcIndex+1+16*(2+4*mm)), (ret+retIndex+16*(2+4*mm)));
0610 //      SlowCopy128bit((src+srcIndex+1+16*(3+4*mm)), (ret+retIndex+16*(3+4*mm)));
0611 // if (mm--) goto gogogo;
0612 //
0613 // For enwik9 (as 64bit code) RAM-to-RAM performance: 793 MB/s:
0614 // mm = ((WORDpair & 0xFF)>>1);
0615 // Fantagi = 0;
0616 // gogogo:
0617 //      *(uint64_t*)(ret+retIndex+8*(0+Fantagi)) = *(uint64_t*)(src+srcIndex+1+8*(0+Fantagi));
0618 //      *(uint64_t*)(ret+retIndex+8*(1+Fantagi)) = *(uint64_t*)(src+srcIndex+1+8*(1+Fantagi));
0619 //      *(uint64_t*)(ret+retIndex+8*(2+Fantagi)) = *(uint64_t*)(src+srcIndex+1+8*(2+Fantagi));
0620 //      // 3 x GP 793MB/s
0621 //      // 2 x GP 773MB/s
0622 //      // 1 x GP 718MB/s
0623 //      // 3 x XMM 664MB/s
0624 //      // 2 x XMM 727MB/s
0625 //      // 1 x XMM 727MB/s
0626 //      Fantagi = Fantagi + 3;
0627 //      mm = mm - 24;
0628 // if (mm>0) goto gogogo;
0629 //
0630 // retIndex+=(WORDpair & 0xFF)>>1;
0631 // srcIndex+=((WORDpair & 0xFF)>>1)+1;
0632 // }
0633 // else{
0634 //      // 1bit+3bits+12bits:
0635 //      //match = ((src[srcIndex] & 0x7F) >> 4)+Min_Match_Length;
0636 //      //index = (src[srcIndex] & 0x0F) << 8;
0637 //      // 1bit+1bit+14bits:
0638 //      //match = ((src[srcIndex] & 0x4F) >> 4)+Min_Match_Length; // In fact, not needed when eightfoldness is commenced, match is 8.
0639 // The fragment below is outrageously ineffective - it can be done in one WORD operation instead of two BYTE operations.
0640 // -----|
0641 // \ /
0642 //      //index = (src[srcIndex] & 0x3F) << 8;
0643 //      //srcIndex++;
0644 //      //index = (index | (unsigned int)(0x00FF & src[srcIndex])) + Min_Match_Length;

```



```

0645 //srcIndex++;
0646 // -----| \
0647 // -----| \
0648 // -----| \
0649 // -----| \
0650 index = (WORDpair>>2) + Min_Match_Length; // 14bit
0651 //index = (WORDpair>>1) + Min_Match_Length; // 15bit
0652 srcIndex=srcIndex+2;
0653 // -----| \
0654 // -----| \
0655 //if (src[srcIndex-1] & 0x40) { // 4 if seventh/fifteenth bit is not zero
0656 //if (WORDpair & 0x02) { // 4 if 2/14 bit is not zero
0657 //    match = Min_Match_BAILOUT_Length;
0658 //    memcpy(&ret[retIndex],&ret[retIndex-index],match);
0659 //    *(uint32_t*)(ret+retIndex) = *(uint32_t*)(ret+retIndex-index);
0660 //} else {
0661 //    match = Min_Match_BAILOUT_Length*4;
0662 //    /*(uint32_t*)(ret+retIndex) = *(uint32_t*)(ret+retIndex-index);
0663 //    /*(uint32_t*)(ret+retIndex+4) = *(uint32_t*)(ret+retIndex-index+4);
0664 //    /*(uint32_t*)(ret+retIndex+8) = *(uint32_t*)(ret+retIndex-index+8);
0665 //    /*(uint32_t*)(ret+retIndex+12) = *(uint32_t*)(ret+retIndex-index+12);
0666 //    SlowCopy128bit((ret+retIndex-index), (ret+retIndex));
0667 //}
0668 match = Min_Match_BAILOUT_Length;
0669 *(uint64_t*)(ret+retIndex) = *(uint64_t*)(ret+retIndex-index);
0670 retIndex+=match;
0671 }
0672 }
0673 return retIndex;
0674 }
0675
0676 // Decompression main loop:
0677 /*
0678 ; mark_description "Intel(R) C++ Intel(R) 64 Compiler XE for applications running on Intel(R) 64, Version 12.1.1.258 Build 20111";
0679 ; mark_description "-O3 -FACS";
0680
0681 .B7.3::
0682 0002e 48 03 c2      add rax, rdx
0683 00031 44 0f b7 18    movzx r11d, WORD PTR [rax]
0684 00035 41 f7 c3 01 00  test r11d, 1
0685 00 00
0686 0003c 0f 85 82 00 00  jne .B7.10 ; Prob 10%
0687 00
0688 .B7.4::
0689 00042 45 0f b6 db      movzx r11d, r11b
0690 00046 45 33 e4        xor r12d, r12d
0691 00049 41 d1 eb        shr r11d, 1
0692 0004c 4a 8d 2c 09      lea rbp, QWORD PTR [rcx+r9]
0693 00050 45 89 dd          mov r13d, r11d
0694 .B7.5::
0695 00053 41 83 c5 e8        add r13d, -24
0696 00057 46 8d 34 e5 00     lea r14d, DWORD PTR [r12*8]
0697 00 00 00
0698 0005f 4e 8b 7c 30 01     mov r15, QWORD PTR [1+rax+r14]
0699 00064 4d 89 3c 2e      mov QWORD PTR [r14+rbp], r15
0700 00068 46 8d 3c e5 08     lea r15d, DWORD PTR [8+r12*8]
0701 00 00 00
0702 00070 4d 8b 74 07 01     mov r14, QWORD PTR [1+r15+rax]
0703 00075 4d 89 34 2f      mov QWORD PTR [r15+rbp], r14
0704 00079 46 8d 34 e5 10     lea r14d, DWORD PTR [16+r12*8]
0705 00 00 00
0706 00081 41 83 c4 03        add r12d, 3
0707 00085 45 85 ed        test r13d, r13d
0708 00088 4d 8b 7c 06 01     mov r15, QWORD PTR [1+r14+rax]
0709 0008d 4d 89 3c 2e      mov QWORD PTR [r14+rbp], r15
0710 00091 7f c0            jg .B7.5 ; Prob 82%
0711 .B7.6::
0712 00093 43 8d 44 1a 01     lea eax, DWORD PTR [1+r10+r11]
0713 00098 45 03 cb        add r9d, r11d
0714 .B7.7::
0715 0009b 41 89 c2          mov r10d, eax
0716 0009e 45 3b d0          cmp r10d, r8d
0717 000a1 72 8b            jb .B7.3 ; Prob 82%
0718 .B7.8::
0719 000a3 4c 8b 64 24 20     mov r12, QWORD PTR [32+rsp]
0720 000a8 4c 8b 6c 24 28     mov r13, QWORD PTR [40+rsp]
0721 000ad 4c 8b 74 24 30     mov r14, QWORD PTR [48+rsp]
0722 000b2 4c 8b 7c 24 38     mov r15, QWORD PTR [56+rsp]
0723 000b7 48 8b 6c 24 40     mov rbp, QWORD PTR [64+rsp]
0724 .B7.9::
0725 000bc 44 89 c8          mov eax, r9d
0726 000bf 48 83 c4 48        add rsp, 72
0727 000c3 c3              ret
0728 .B7.10::
0729 000c4 41 c1 eb 02        shr r11d, 2
0730 000c8 41 83 c2 02        add r10d, 2
0731 000cc 41 83 c3 08        add r11d, 8
0732 000d0 49 f7 db          neg r11
0733 000d3 4c 03 d9          add r11, rcx

```

```

0734 000d6 44 89 d0      mov eax, r10d
0735 000d9 4b 8b 2c 19     mov rbp, QWORD PTR [r9+r11]
0736 000dd 49 89 2c 09     mov QWORD PTR [r9+rcx], rbp
0737 000e1 41 83 c1 08      add r9d, 8
0738 000e5 eb b4          jmp .B7.7 ; Prob 100%
0739 */
0740
0741 // Yappy vs Nakamichi 'Speed Showdown' on Core2 T7500 2200MHz with enwik9:
0742 /*
0743 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Yappy_vs_Nakamichi_SpeedShowdown_on_enwik9.bat
0744 Yappy vs Nakamichi 'Speed Showdown' on enwik9 ...
0745
0746 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Yappy_64bit.exe enwik9 512
0747 YAPPY: [b 0K] bytes 1000000000 -> 779421758 77.9% comp 40.9 MB/s uncomp 702.8 MB/s
0748
0749 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Yappy_64bit.exe enwik9 1024
0750 YAPPY: [b 1K] bytes 1000000000 -> 714236729 71.4% comp 41.8 MB/s uncomp 657.3 MB/s
0751
0752 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Yappy_64bit.exe enwik9 2048
0753 YAPPY: [b 2K] bytes 1000000000 -> 655356839 65.5% comp 40.1 MB/s uncomp 611.3 MB/s
0754
0755 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Yappy_64bit.exe enwik9 4096
0756 YAPPY: [b 4K] bytes 1000000000 -> 593819184 59.4% comp 36.7 MB/s uncomp 566.0 MB/s
0757
0758 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Yappy_64bit.exe enwik9 8192
0759 YAPPY: [b 8K] bytes 1000000000 -> 544342520 54.4% comp 33.6 MB/s uncomp 545.9 MB/s
0760
0761 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Yappy_64bit.exe enwik9 16384
0762 YAPPY: [b 16K] bytes 1000000000 -> 519654588 52.0% comp 32.3 MB/s uncomp 540.9 MB/s
0763
0764 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Yappy_64bit.exe enwik9 32768
0765 YAPPY: [b 32K] bytes 1000000000 -> 507264601 50.7% comp 32.1 MB/s uncomp 541.2 MB/s
0766
0767 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Yappy_64bit.exe enwik9 65536
0768 YAPPY: [b 64K] bytes 1000000000 -> 501106828 50.1% comp 32.2 MB/s uncomp 540.9 MB/s
0769
0770 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Yappy_32bit.exe enwik9 512
0771 YAPPY: [b 0K] bytes 1000000000 -> 779421758 77.9% comp 43.0 MB/s uncomp 710.6 MB/s
0772
0773 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Yappy_32bit.exe enwik9 1024
0774 YAPPY: [b 1K] bytes 1000000000 -> 714236729 71.4% comp 42.2 MB/s uncomp 657.7 MB/s
0775
0776 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Yappy_32bit.exe enwik9 2048
0777 YAPPY: [b 2K] bytes 1000000000 -> 655356839 65.5% comp 39.3 MB/s uncomp 611.3 MB/s
0778
0779 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Yappy_32bit.exe enwik9 4096
0780 YAPPY: [b 4K] bytes 1000000000 -> 593819184 59.4% comp 36.3 MB/s uncomp 571.4 MB/s
0781
0782 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Yappy_32bit.exe enwik9 8192
0783 YAPPY: [b 8K] bytes 1000000000 -> 544342520 54.4% comp 33.5 MB/s uncomp 550.9 MB/s
0784
0785 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Yappy_32bit.exe enwik9 16384
0786 YAPPY: [b 16K] bytes 1000000000 -> 519654588 52.0% comp 32.2 MB/s uncomp 550.6 MB/s
0787
0788 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Yappy_32bit.exe enwik9 32768
0789 YAPPY: [b 32K] bytes 1000000000 -> 507264601 50.7% comp 32.0 MB/s uncomp 545.9 MB/s
0790
0791 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Yappy_32bit.exe enwik9 65536
0792 YAPPY: [b 64K] bytes 1000000000 -> 501106828 50.1% comp 32.2 MB/s uncomp 545.9 MB/s
0793
0794 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_64bit.exe enwik9.Nakamichi
0795 Nakamichi, revision 1-RSSBO_1GB_wordfetcher_TRIAD, written by Kaze, based on Nobuo Ito's LZSS source.
0796 Decompressing 641441055 bytes ...
0797 RAM-to-RAM performance: 772 MB/s.
0798
0799 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_32bit.exe enwik9.Nakamichi
0800 Nakamichi, revision 1-RSSBO_1GB_wordfetcher_TRIAD, written by Kaze, based on Nobuo Ito's LZSS source.
0801 Decompressing 641441055 bytes ...
0802 RAM-to-RAM performance: 678 MB/s.
0803
0804 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy_64bit.exe enwik9.Nakamichi
0805 Nakamichi, revision 1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy, written by Kaze, based on Nobuo Ito's LZSS source, babealicious suggestion by m^2 enforced.
0806 Decompressing 641441055 bytes ...
0807 RAM-to-RAM performance: 783 MB/s.
0808
0809 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy_32bit.exe enwik9.Nakamichi
0810 Nakamichi, revision 1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy, written by Kaze, based on Nobuo Ito's LZSS source, babealicious suggestion by m^2 enforced.
0811 Decompressing 641441055 bytes ...
0812 RAM-to-RAM performance: 678 MB/s.
0813
0814 D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_Nomemcpy>
0815 */
0816
0817 // In my opinion Hamid Buzidi is the best, therefore his lzturbo v1.1 reference results are given below:
0818 /*
0819 D:\_KAZE\Nakamichi_r1-RSSBO>timer32 lzturbo.exe -19 -p0 enwiki-20140304-pages-articles.7z.001 .
0820
0821 Kernel Time = 0.982 = 0%
0822 User Time = 152.537 = 99%

```

```

0823 Process Time = 153.520 = 100% Virtual Memory = 429 MB
0824 Global Time = 153.519 = 100% Physical Memory = 407 MB
0825
0826 D:\_KAZE\Nakamichi_r1-RSSBO>timer32.exe lzturbo.exe -d enwiki-20140304-pages-articles.7z.001.lzt .
0827
0828 Kernel Time = 0.234 = 62%
0829 User Time = 0.187 = 50%
0830 Process Time = 0.421 = 112% Virtual Memory = 98 MB
0831 Global Time = 0.374 = 100% Physical Memory = 70 MB
0832
0833 D:\_KAZE\Nakamichi_r1-RSSBO>dir
0834
0835 04/15/2014 08:05 AM 104,857,600 enwiki-20140304-pages-articles.7z.001
0836 04/15/2014 08:04 AM 41,984,881 enwiki-20140304-pages-articles.7z.001.lzt
0837
0838 D:\_KAZE\Nakamichi_r1-RSSBO>timer32 lzturbo.exe -11 -p0 enwiki-20140304-pages-articles.7z.001 .
0839
0840 Kernel Time = 0.171 = 9%
0841 User Time = 1.622 = 90%
0842 Process Time = 1.794 = 100% Virtual Memory = 58 MB
0843 Global Time = 1.794 = 100% Physical Memory = 39 MB
0844
0845 D:\_KAZE\Nakamichi_r1-RSSBO>timer32.exe lzturbo.exe -d enwiki-20140304-pages-articles.7z.001.lzt .
0846
0847 Kernel Time = 0.249 = 41%
0848 User Time = 0.140 = 23%
0849 Process Time = 0.390 = 64% Virtual Memory = 98 MB
0850 Global Time = 0.608 = 100% Physical Memory = 73 MB
0851
0852 D:\_KAZE\Nakamichi_r1-RSSBO>dir
0853
0854 04/15/2014 08:05 AM 104,857,600 enwiki-20140304-pages-articles.7z.001
0855 04/15/2014 08:05 AM 47,685,453 enwiki-20140304-pages-articles.7z.001.lzt
0856
0857 D:\_KAZE\Nakamichi_r1-RSSBO>
0858 */
0859
0860 // Railgun_Swampshine_BailOut, cpyleft 2014-Jan-31, Kaze.
0861 // Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
0862 #define NeedleThreshold2vs4swampLITE 9+10 // Should be bigger than 9. BMH2 works up to this value (inclusive), if bigger then BMH4 takes over.
0863 char * Railgun_Swampshine_BailOut (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
0864 {
0865     char * pbTargetMax = pbTarget + cbTarget;
0866     register uint32_t ulHashPattern;
0867     signed long count;
0868
0869     unsigned char bm_Horspool_Order2[256*256]; // Bitwise soon...
0870     uint32_t i, Gulliver;
0871
0872     uint32_t PRIMALposition, PRIMALpositionCANDIDATE;
0873     uint32_t PRIMALlength, PRIMALlengthCANDIDATE;
0874     uint32_t j, FoundAtPosition;
0875
0876     if (cbPattern > cbTarget) return(NULL);
0877
0878     if ( cbPattern<4 ) {
0879         // SSE2 i.e. 128bit Assembly rules here:
0880         // ...
0881         pbTarget = pbTarget+cbPattern;
0882         ulHashPattern = ( (*char*)(pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
0883         if ( cbPattern==3 ) {
0884             for ( ;; ) {
0885                 if ( ulHashPattern == ( (*char*)(pbTarget-3))<<8 ) + *(pbTarget-1) ) {
0886                     if ( (*char*)(pbPattern+1) == (*char*)(pbTarget-2) ) return((pbTarget-3));
0887                 }
0888                 if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) {
0889                     pbTarget++;
0890                     if ( (char)(ulHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
0891                 }
0892                 pbTarget++;
0893                 if (pbTarget > pbTargetMax) return(NULL);
0894             }
0895         } else {
0896             for ( ;; ) {
0897                 if ( ulHashPattern == ( (*char*)(pbTarget-2))<<8 ) + *(pbTarget-1) ) return((pbTarget-2));
0898                 if ( (char)(ulHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
0899                 pbTarget++;
0900                 if (pbTarget > pbTargetMax) return(NULL);
0901             }
0902         }
0903     } else { //if ( cbPattern<4 )
0904         if ( cbPattern<=NeedleThreshold2vs4swampLITE ) {
0905             // BMH order 2, needle should be >=4:
0906             ulHashPattern = *(uint32_t*)(pbPattern); // First four bytes
0907             for (i=0; i < 256*256; i++) {bm_Horspool_Order2[i]=0;}
0908             for (i=0; i < cbPattern-1; i++) bm_Horspool_Order2[(unsigned short*)(pbPattern+i)]=1;
0909             i=0;
0910             while (i <= cbTarget-cbPattern) {
0911                 Gulliver = 1; // 'Gulliver' is the skip

```

```

0912         if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1]] != 0 ) {
0913             if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-2]] == 0 ) Gulliver = cbPattern-(2-1)-2; else
{
0914                 if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) { // This fast check ensures not missing a match (for
remainder) when going under 0 in loop below:
0915                     count = cbPattern-4+1;
0916                     while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)&pbTarget[i]+(count-
1)) )
0917                         count = count-4;
0918                     if ( count <= 0 ) return(pbTarget+i);
0919                 }
0920             }
0921         } else Gulliver = cbPattern-(2-1);
0922         i = i + Gulliver;
0923         //GlobalI++; // Comment it, it is only for stats.
0924     }
0925     return(NULL);
0926 } else { // if ( cbPattern<=NeedleThreshold2vs4swampLITE )
0927
0928     // Swampwalker_BAILOUT heuristic order 4 (Needle should be bigger than 4) [
0929     // Needle: 1234567890qwertyuiopasdfghjklzxcv          PRIMALposition=01 PRIMALlength=33 '1234567890qwertyuiopasdfghjklzxcv'
0930     // Needle: vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv          PRIMALposition=29 PRIMALlength=04 'vvvv'
0931     // Needle: vvvvvvvvvvBOOMSHAKALAKAvvvvvvvvv          PRIMALposition=08 PRIMALlength=20 'vvvBOOMSHAKALAKAvvvv'
0932     // Needle: Trollland          PRIMALposition=01 PRIMALlength=09 'Trollland'
0933     // Needle: Swampwalker          PRIMALposition=01 PRIMALlength=11 'Swampwalker'
0934     // Needle: licenselessness          PRIMALposition=01 PRIMALlength=15 'licenselessness'
0935     // Needle: alfalfa          PRIMALposition=02 PRIMALlength=06 'lalfa'
0936     // Needle: Sandokan          PRIMALposition=01 PRIMALlength=08 'Sandokan'
0937     // Needle: shazamish          PRIMALposition=01 PRIMALlength=09 'shazamish'
0938     // Needle: Simplicius Simplicissimus          PRIMALposition=06 PRIMALlength=20 'icius Simplicissimus'
0939     // Needle: domilliaquadringenquattuorquinguintillion          PRIMALposition=01 PRIMALlength=32 'domilliaquadringenquattuorquingu'
0940     // Needle: boom-boom          PRIMALposition=02 PRIMALlength=08 'oom-boom'
0941     // Needle: vvvvv          PRIMALposition=01 PRIMALlength=04 'vvvv'
0942     // Needle: 12345          PRIMALposition=01 PRIMALlength=05 '12345'
0943     // Needle: likey-likey          PRIMALposition=03 PRIMALlength=09 'key-likey'
0944     // Needle: BOOOOOM          PRIMALposition=03 PRIMALlength=05 'OOOOM'
0945     // Needle: aaaaaBOOOOOM          PRIMALposition=02 PRIMALlength=09 'aaaaBOOOO'
0946     // Needle: BOOOOMaaaaa          PRIMALposition=03 PRIMALlength=09 'OOOOMaaaa'
0947     PRIMALlength=0;
0948     for (i=0+(1); i < cbPattern-((4)-1)+(1)-(1); i++) { // -(1) because the last BB order 4 has no counterpart(s)
0949         FoundAtPosition = cbPattern - ((4)-1) + 1;
0950         PRIMALpositionCANDIDATE=i;
0951         while ( PRIMALpositionCANDIDATE <= (FoundAtPosition-1) ) {
0952             j = PRIMALpositionCANDIDATE + 1;
0953             while ( j <= (FoundAtPosition-1) ) {
0954                 if ( *(uint32_t *)(pbPattern+PRIMALpositionCANDIDATE-(1)) == *(uint32_t *)(pbPattern+j-(1)) ) FoundAtPosition = j;
0955                 j++;
0956             }
0957             PRIMALpositionCANDIDATE++;
0958         }
0959         PRIMALlengthCANDIDATE = (FoundAtPosition-1)-i+1+((4)-1);
0960         if (PRIMALlengthCANDIDATE >= PRIMALlength) {PRIMALposition=i; PRIMALlength = PRIMALlengthCANDIDATE;}
0961         if (cbPattern-i+1 <= PRIMALlength) break;
0962         if (PRIMALlength > 128) break; // Bail Out for 129[+]
0963     }
0964     // Swampwalker_BAILOUT heuristic order 4 (Needle should be bigger than 4) ]
0965
0966     // Here we have 4 or bigger NewNeedle, apply order 2 for pbPattern[i+(PRIMALposition-1)] with length 'PRIMALlength' and compare the pbPattern[i] with length
'cbPattern':
0967     PRIMALlengthCANDIDATE = cbPattern;
0968     cbPattern = PRIMALlength;
0969     pbPattern = pbPattern + (PRIMALposition-1);
0970
0971     // Revision 2 commented section [
0972     /*
0973     if (cbPattern-1 <= 255) {
0974         // BMH Order 2 [
0975         ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
0976         for (i=0; i < 256*256; i++) {bm_Horspool_Order2[i]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
0977         for (i=0; i < cbPattern-1; i++) bm_Horspool_Order2[*(unsigned short *)(pbPattern+i)]=i; // Rightmost appearance/position is needed
0978         i=0;
0979         while (i <= cbTarget-cbPattern) {
0980             Gulliver = bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+cbPattern-1-1]];
0981             if ( Gulliver != cbPattern-1 ) { // CASE #2: if equal means the pair (char order 2) is not found i.e. Gulliver remains
intact, skip the whole pattern and fall back (order-1) chars i.e. one char for Order 2
0982                 if ( Gulliver == cbPattern-2 ) { // CASE #1: means the pair (char order 2) is found
0983                     if ( *(uint32_t *)&pbTarget[i] == ulHashPattern ) {
0984                         count = cbPattern-4+1;
0985                         while ( count > 0 && *(uint32_t *)(pbPattern+count-1) == *(uint32_t *)&pbTarget[i]+(count-1)) )
0986                             count = count-4;
0987                     // If we miss to hit then no need to compare the original: Needle
0988                     if ( count <= 0 ) {
0989                         // I have to add out-of-range checks...
0990                         // i-(PRIMALposition-1) >= 0
0991                         // &pbTarget[i-(PRIMALposition-1)] <= pbTargetMax - 4
0992                         // i-(PRIMALposition-1)+(count-1) >= 0
0993                         // &pbTarget[i-(PRIMALposition-1)+(count-1)] <= pbTargetMax - 4
0994                         if ( (i-(PRIMALposition-1)) >= 0 && (&pbTarget[i-(PRIMALposition-1)] <= pbTargetMax - 4) && (&pbTarget[i-(PRIMALposition-1)+(count-1)] <= pbTargetMax - 4) ) {
0995                             if ( *(uint32_t *)&pbTarget[i-(PRIMALposition-1)] == *(uint32_t *)(pbPattern-(PRIMALposition-1))) { // This fast check ensures not missing a match

```

```

(for remainder) when going under 0 in loop below:
0996         count = PRIMALlengthCANDIDATE-4+1;
0997         while ( count > 0 && *(uint32_t *) (pbPattern-(PRIMALposition-1)+count-1) == *(uint32_t *) (&pbTarget[i-(PRIMALposition-1)]+(count-1)) )
0998             count = count-4;
0999         if ( count <= 0 ) return(pbTarget+i-(PRIMALposition-1));
1000     }
1001 }
1002 }
1003     }
1004     Gulliver = 1;
1005 } else
1006     Gulliver = cbPattern - Gulliver - 2; // CASE #3: the pair is found and not as suffix i.e. rightmost position
1007 }
1008 i = i + Gulliver;
1009 //GlobalI++; // Comment it, it is only for stats.
1010 }
1011 return(NULL);
1012 // BMH Order 2 ]
1013 } else {
1014     // BMH order 2, needle should be >=4:
1015     ulHashPattern = *(uint32_t *) (pbPattern); // First four bytes
1016     for (i=0; i < 256*256; i++) {bm_Horspool_Order2[i]=0;}
1017     for (i=0; i < cbPattern-1; i++) bm_Horspool_Order2[(unsigned short *) (pbPattern+i)]=1;
1018     i=0;
1019     while (i <= cbTarget-cbPattern) {
1020         Gulliver = 1; // 'Gulliver' is the skip
1021         if ( bm_Horspool_Order2[(unsigned short *) &pbTarget[i+cbPattern-1-1]] != 0 ) {
1022             if ( bm_Horspool_Order2[(unsigned short *) &pbTarget[i+cbPattern-1-2]] == 0 ) Gulliver = cbPattern-(2-1)-2; else
1023             if ( *(uint32_t *) &pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for
remainder) when going under 0 in loop below:
1024                 count = cbPattern-4+1;
1025                 while ( count > 0 && *(uint32_t *) (pbPattern+count-1) == *(uint32_t *) (&pbTarget[i]+(count-
1))) )
1026                     count = count-4;
1027 // If we miss to hit then no need to compare the original: Needle
1028 if ( count <= 0 ) {
1029 // I have to add out-of-range checks...
1030 // i-(PRIMALposition-1) >= 0
1031 // &pbTarget[i-(PRIMALposition-1)] <= pbTargetMax - 4
1032 // i-(PRIMALposition-1)+(count-1) >= 0
1033 // &pbTarget[i-(PRIMALposition-1)+(count-1)] <= pbTargetMax - 4
1034 if ( (i-(PRIMALposition-1) >= 0) && (&pbTarget[i-(PRIMALposition-1)] <= pbTargetMax - 4) && (&pbTarget[i-(PRIMALposition-1)+(count-1)] <= pbTargetMax - 4) ) {
1035 if ( *(uint32_t *) &pbTarget[i-(PRIMALposition-1)] == *(uint32_t *) (pbPattern-(PRIMALposition-1))) { // This fast check ensures not missing a match
(for remainder) when going under 0 in loop below:
1036     count = PRIMALlengthCANDIDATE-4+1;
1037     while ( count > 0 && *(uint32_t *) (pbPattern-(PRIMALposition-1)+count-1) == *(uint32_t *) (&pbTarget[i-(PRIMALposition-1)]+(count-1)) )
1038         count = count-4;
1039     if ( count <= 0 ) return(pbTarget+i-(PRIMALposition-1));
1040 }
1041 }
1042 }
1043     }
1044     }
1045     } else Gulliver = cbPattern-(2-1);
1046     i = i + Gulliver;
1047     //GlobalI++; // Comment it, it is only for stats.
1048 }
1049 return(NULL);
1050 }
1051 */
1052 // Revision 2 commented section ]
1053
1054 if ( cbPattern<=NeedleThreshold2vs4swampLITE ) {
1055
1056     // BMH order 2, needle should be >=4:
1057     ulHashPattern = *(uint32_t *) (pbPattern); // First four bytes
1058     for (i=0; i < 256*256; i++) {bm_Horspool_Order2[i]=0;}
1059     for (i=0; i < cbPattern-1; i++) bm_Horspool_Order2[(unsigned short *) (pbPattern+i)]=1;
1060     i=0;
1061     while (i <= cbTarget-cbPattern) {
1062         Gulliver = 1; // 'Gulliver' is the skip
1063         if ( bm_Horspool_Order2[(unsigned short *) &pbTarget[i+cbPattern-1-1]] != 0 ) {
1064             if ( bm_Horspool_Order2[(unsigned short *) &pbTarget[i+cbPattern-1-2]] == 0 ) Gulliver = cbPattern-(2-1)-2; else
1065             if ( *(uint32_t *) &pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for
remainder) when going under 0 in loop below:
1066                 count = cbPattern-4+1;
1067                 while ( count > 0 && *(uint32_t *) (pbPattern+count-1) == *(uint32_t *) (&pbTarget[i]+(count-
1))) )
1068                     count = count-4;
1069 // If we miss to hit then no need to compare the original: Needle
1070 if ( count <= 0 ) {
1071 // I have to add out-of-range checks...
1072 // i-(PRIMALposition-1) >= 0
1073 // &pbTarget[i-(PRIMALposition-1)] <= pbTargetMax - 4
1074 // i-(PRIMALposition-1)+(count-1) >= 0
1075 // &pbTarget[i-(PRIMALposition-1)+(count-1)] <= pbTargetMax - 4
1076 if ( (i-(PRIMALposition-1) >= 0) && (&pbTarget[i-(PRIMALposition-1)] <= pbTargetMax - 4) && (&pbTarget[i-(PRIMALposition-1)+(count-1)] <= pbTargetMax - 4) ) {

```

```

1077         if ( *(uint32_t *)&pbTarget[i-(PRIMALposition-1)] == *(uint32_t *)(pbPattern-(PRIMALposition-1))) { // This fast check ensures not missing a match
(for remainder) when going under 0 in loop below:
1078             count = PRIMALlengthCANDIDATE-4+1;
1079             while ( count > 0 && *(uint32_t *)(pbPattern-(PRIMALposition-1)+count-1) == *(uint32_t *)&pbTarget[i-(PRIMALposition-1)]+(count-1) )
1080                 count = count-4;
1081             if ( count <= 0 ) return(pbTarget+i-(PRIMALposition-1));
1082         }
1083     }
1084 }
1085
1086     }
1087     } else Gulliver = cbPattern-(2-1);
1088     i = i + Gulliver;
1089     //GlobalI++; // Comment it, it is only for stats.
1090 }
1091 return(NULL);
1092
1093 } else { // if ( cbPattern<=NeedleThreshold2vs4swampLITE )
1094
1095     // BMH pseudo-order 4, needle should be >=8+2:
1096     ulHashPattern = *(uint32_t *)(pbPattern); // First four bytes
1097     for (i=0; i < 256*256; i++) {bm_Horspool_Order2[i]=0;}
1098     // 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:
1099     // "fast"
1100     // "aste"
1101     // "stes"
1102     // "test"
1103     // "est "
1104     // "st f"
1105     // "t fo"
1106     // " fox"
1107     // for (i=0; i < cbPattern-4+1; i++) bm_Horspool_Order2[( *(unsigned short *)(pbPattern+i+0) + *(unsigned short *)(pbPattern+i+2) ) & (
(1<<16)-1 )]=1;
1108     // for (i=0; i < cbPattern-4+1; i++) bm_Horspool_Order2[( *(uint32_t *)(pbPattern+i+0)>>16)+*(uint32_t *)(pbPattern+i+0)&0xFFFF) & (
(1<<16)-1 )]=1;
1109     // Above line is replaced by next one with better hashing:
1110     for (i=0; i < cbPattern-4+1; i++) bm_Horspool_Order2[( *(uint32_t *)(pbPattern+i+0)>>(16-1))+*(uint32_t *)(pbPattern+i+0)&0xFFFF) & (
(1<<16)-1 )]=1;
1111     i=0;
1112     while (i <= cbTarget-cbPattern) {
1113         Gulliver = 1;
1114         // 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 ) { // DWORD #1
1115             // Above line is replaced by next one with better hashing:
1116             if ( bm_Horspool_Order2[( *(uint32_t *)&pbTarget[i+cbPattern-1-1-2]>>(16-1))+*(uint32_t *)&pbTarget[i+cbPattern-1-1-
2]&0xFFFF) & ( (1<<16)-1 ) ] != 0 ) { // DWORD #1
1117                 // 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 {
1118                 // Above line is replaced in order to strengthen the skip by checking the middle DWORD, if the two DWORDs are 'ab'
and 'cd' i.e. [2x][2a][2b][2c][2d] then the middle DWORD is 'bc'.
1119                 // The respective offsets (backwards) are: -10/-8/-6/-4 for 'xa'/'ab'/'bc'/'cd'.
1120                 // if ( ( bm_Horspool_Order2[( *(uint32_t *)&pbTarget[i+cbPattern-1-1-2-6]>>16)+*(uint32_t
*)&pbTarget[i+cbPattern-1-1-2-6]&0xFFFF) & ( (1<<16)-1 ) ] + ( 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 ) ] + ( bm_Horspool_Order2[( *(uint32_t *)&pbTarget[i+cbPattern-1-1-2-2]>>16)+*(uint32_t
*)&pbTarget[i+cbPattern-1-1-2-2]&0xFFFF) & ( (1<<16)-1 ) ] < 3 ) Gulliver = cbPattern-(2-1)-2-4-2; else {
1121                 // Above line is replaced by next one with better hashing:
1122                 // when using (16-1) right shifting instead of 16 we will have two different pairs (if they are equal), the
highest bit being lost do the job especialy for ASCII texts with no symbols in range 128-255.
1123                 // Example for genomesque pair TT+TT being shifted by (16-1):
1124                 // T          = 01010100
1125                 // TT         = 01010100 01010100
1126                 // TTTT        = 01010100 01010100 01010100 01010100
1127                 // TTTT>>16    = 00000000 00000000 01010100 01010100
1128                 // TTTT>>(16-1) = 00000000 00000000 10101000 10101000 <--- Due to the left shift by 1, the 8th bits of 1st and 2nd
bytes are populated - usually they are 0 for English texts & 'ACGT' data.
1129                 // if ( ( bm_Horspool_Order2[( *(uint32_t *)&pbTarget[i+cbPattern-1-1-2-6]>>(16-1))+*(uint32_t
*)&pbTarget[i+cbPattern-1-1-2-6]&0xFFFF) & ( (1<<16)-1 ) ] + ( bm_Horspool_Order2[( *(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>(16-1))+*(uint32_t
*)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) & ( (1<<16)-1 ) ] + ( bm_Horspool_Order2[( *(uint32_t *)&pbTarget[i+cbPattern-1-1-2-2]>>(16-1))+*(uint32_t
*)&pbTarget[i+cbPattern-1-1-2-2]&0xFFFF) & ( (1<<16)-1 ) ] < 3 ) Gulliver = cbPattern-(2-1)-2-4-2; else {
1130                 // 'Maximus' uses branched 'if', again.
1131                 if ( \
1132                     ( bm_Horspool_Order2[( *(uint32_t *)&pbTarget[i+cbPattern-1-1-2-6 +1]>>(16-1))+*(uint32_t
*)&pbTarget[i+cbPattern-1-1-2-6 +1]&0xFFFF) & ( (1<<16)-1 ) ] == 0 \
1133                     || ( bm_Horspool_Order2[( *(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4 +1]>>(16-1))+*(uint32_t
*)&pbTarget[i+cbPattern-1-1-2-4 +1]&0xFFFF) & ( (1<<16)-1 ) ] == 0 \
1134                     ) Gulliver = cbPattern-(2-1)-2-4-2 +1; else {
1135                 // Above line is not optimized (several a SHR are used), we have 5 non-overlapping WORDS, or 3 overlapping WORDS,
within 4 overlapping DWORDS so:
1136 // [2x][2a][2b][2c][2d]
1137 // DWORD #4
1138 // [2a] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-6]>>16) = !SHR to be avoided! <--
1139 // [2x] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-6]&0xFFFF) = -----|
1140 // DWORD #3
1141 // [2b] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>16) = !SHR to be avoided! |<--
1142 // [2a] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) = -----|
1143 // DWORD #2
1144 // [2c] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-2]>>16) = !SHR to be avoided! |<--
1145 // [2b] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-2]&0xFFFF) = -----|

```

```

1146 //          DWORD #1
1147 // [2d] (* (uint32_t *)&pbTarget[i+cbPattern-1-1-2-0]>>16) =
1148 // [2c] (* (uint32_t *)&pbTarget[i+cbPattern-1-1-2-0]&0xFFFF) = -----
1149 //
1150 // So in order to remove 3 SHR instructions the equal extractions are:
1151 // DWORD #4
1152 // [2a] (* (uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) = !SHR to be avoided! <--
1153 // [2x] (* (uint32_t *)&pbTarget[i+cbPattern-1-1-2-6]&0xFFFF) =
1154 //          DWORD #3
1155 // [2b] (* (uint32_t *)&pbTarget[i+cbPattern-1-1-2-2]&0xFFFF) = !SHR to be avoided! <--
1156 // [2a] (* (uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) = -----
1157 //          DWORD #2
1158 // [2c] (* (uint32_t *)&pbTarget[i+cbPattern-1-1-2-0]&0xFFFF) = !SHR to be avoided! <--
1159 // [2b] (* (uint32_t *)&pbTarget[i+cbPattern-1-1-2-2]&0xFFFF) = -----
1160 //          DWORD #1
1161 // [2d] (* (uint32_t *)&pbTarget[i+cbPattern-1-1-2-0]>>16) =
1162 // [2c] (* (uint32_t *)&pbTarget[i+cbPattern-1-1-2-0]&0xFFFF) = -----
1163 //if ( ( bm_Horspool_Order2[ ( (* (uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF)+(* (uint32_t
*)&pbTarget[i+cbPattern-1-1-2-6]&0xFFFF) ) & ( (1<<16)-1 ) ] ) + ( bm_Horspool_Order2[ ( (* (uint32_t *)&pbTarget[i+cbPattern-1-1-2-2]&0xFFFF)+(* (uint32_t
*)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) ) & ( (1<<16)-1 ) ] ) + ( bm_Horspool_Order2[ ( (* (uint32_t *)&pbTarget[i+cbPattern-1-1-2-0]&0xFFFF)+(* (uint32_t
*)&pbTarget[i+cbPattern-1-1-2-2]&0xFFFF) ) & ( (1<<16)-1 ) ] ) < 3 ) Gulliver = cbPattern-(2-1)-2-6; else {
1164 // Since the above Decumanus mumbo-jumbo (3 overlapping lookups vs 2 non-overlapping lookups) is not fast enough we go DuoDecumanus or 3x4:
1165 // [2y][2x][2a][2b][2c][2d]
1166 //          DWORD #3
1167 //          DWORD #2
1168 //          DWORD #1
1169 //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 ) ] ) + ( bm_Horspool_Order2[ ( (* (uint32_t *)&pbTarget[i+cbPattern-1-1-2-8]>>16)+(* (uint32_t
*)&pbTarget[i+cbPattern-1-1-2-8]&0xFFFF) ) & ( (1<<16)-1 ) ] ) < 2 ) Gulliver = cbPattern-(2-1)-2-8; else {
1170 //if ( (* (uint32_t *)&pbTarget[i] == ulHashPattern) {
1171 //    // Order 4 [
1172 //    // Let's try something "outrageous" like comparing with[out] overlap BBS 4bytes long instead of 1 byte
back-to-back:
1173 //    // 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:
1174 //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
1175 //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
1176 //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
1177 //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
1178 //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
1179 //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
1180 //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
1181 //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
1182 //count = cbPattern-4+1;
1183 // Below comparison is UNIdirectional:
1184 //while ( count > 0 && (* (uint32_t *) (pbPattern+count-1)) == (* (uint32_t *) (&pbTarget[i]+(count-
1185 //)) )
count = count-4;
1186 // count = cbPattern-4+1 = 23-4+1 = 20
1187 // boomshakalakazzzzzz[Zzzz] 20
1188 // boomshakalakazz[Zzzz]Zzzz 20-4
1189 // boomshakala[kazz]Zzzzzzzz 20-8 = 12
1190 // boomsha[kala]kazzzzzzzzzz 20-12 = 8
1191 // boo[msha]kalakazzzzzzzzzz 20-16 = 4
1192
1193 // If we miss to hit then no need to compare the original: Needle
1194 if ( count <= 0 ) {
1195 // I have to add out-of-range checks...
1196 // i-(PRIMALposition-1) >= 0
1197 // &pbTarget[i-(PRIMALposition-1)] <= pbTargetMax - 4
1198 // i-(PRIMALposition-1)+(count-1) >= 0
1199 // &pbTarget[i-(PRIMALposition-1)+(count-1)] <= pbTargetMax - 4
1200 // if ( (i-(PRIMALposition-1) >= 0) && (&pbTarget[i-(PRIMALposition-1)] <= pbTargetMax - 4) && (&pbTarget[i-(PRIMALposition-1)+(count-1)] <= pbTargetMax - 4) ) {
1201 //     if ( (* (uint32_t *)&pbTarget[i-(PRIMALposition-1)]) == (* (uint32_t *) (pbPattern-(PRIMALposition-1))) ) { // This fast check ensures not missing a match
(for remainder) when going under 0 in loop below:
1202 //         count = PRIMALlengthCANDIDATE-4+1;
1203 //         while ( count > 0 && (* (uint32_t *) (pbPattern-(PRIMALposition-1)+count-1)) == (* (uint32_t *) (&pbTarget[i-(PRIMALposition-1)]+(count-1))) )
1204 //             count = count-4;
1205 //         if ( count <= 0 ) return(pbTarget+i-(PRIMALposition-1));
1206 //     }
1207 // }
1208 }
1209
1210 // In order to avoid only-left or only-right WCS the memcmp should be done as left-to-right
and right-to-left AT THE SAME TIME.
1211 // Below comparison is BiDirectional. It pays off when needle is 8+++ long:
1212 // for (count = cbPattern-4+1; count > 0; count = count-4) {
1213 //     if ( (* (uint32_t *) (pbPattern+count-1)) != (* (uint32_t *) (&pbTarget[i]+(count-1))) )
1214 //         if ( (* (uint32_t *) (pbPattern+(cbPattern-4+1)-count)) != (* (uint32_t
*) (&pbTarget[i]+(cbPattern-4+1)-count)) ) {count = (cbPattern-4+1)-count +1; break;} // +1) because two lookups are implemented as one, also no danger of 'count' being
0 because of the fast check outwith the 'while': if ( (* (uint32_t *)&pbTarget[i] == ulHashPattern)
1215 //         }
1216 //         if ( count <= 0 ) return(pbTarget+i);
1217 //         // checking the order 2 pairs in mismatched DWORD, all the 3:
1218 //         //if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+count-1]] == 0 )
1219 //         //if ( bm_Horspool_Order2[*(unsigned short *)&pbTarget[i+count-1+1]] == 0 )
Gulliver = count; // 1 or bigger, as it should
1219
Gulliver = count+1; // 1 or bigger, as it should

```

```

1220 //if ( bm_Horspool_Order2[(unsigned short *)&pbTarget[i+count-1+1]] == 0 )
Gulliver = count+1+1; // 1 or bigger, as it should
1221 // if ( bm_Horspool_Order2[(unsigned short *)&pbTarget[i+count-1]] +
bm_Horspool_Order2[(unsigned short *)&pbTarget[i+count-1+1]] + bm_Horspool_Order2[(unsigned short *)&pbTarget[i+count-1+1]] < 3 ) Gulliver = count; // 1 or bigger,
as it should, THE MIN(count,count+1,count+1+1)
1222 // Above compound 'if' guarantees not that Gulliver > 1, an example:
1223 // Needle: fastest tax
1224 // Window: ...fastest tax...
1225 // After matching 'tax' vs 'tax' and 'fast' vs 'fast' the mismatched DWORD is
'test' vs 'tast':
1226 // 'tast' when factorized down to order 2 yields: 'ta','as','st' - all the three
when summed give 1+1+1=3 i.e. Gulliver remains 1.
1227 // Roughly speaking, this attempt maybe has its place in worst-case scenarios but
not in English text and even not in ACGT data, that's why I commented it in original 'Shockeroo'.
1228 //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
1229 // Above line is replaced by next one with better hashing:
1230 // if ( bm_Horspool_Order2[( (uint32_t *)&pbTarget[i+count-1]>>(16-1))+(uint32_t
*)&pbTarget[i+count-1]&0xFFFF) ] & ( (1<<16)-1) ) == 0 ) Gulliver = count; // 1 or bigger, as it should
1231 // Order 4 ]
1232 }
1233 }
1234 } else Gulliver = cbPattern-(2-1)-2; // -2 because we check the 4 rightmost bytes not 2.
1235 i = i + Gulliver;
1236 //GlobalI++; // Comment it, it is only for stats.
1237 }
1238 return(NULL);
1239
1240 } // if ( cbPattern<=NeedleThreshold2vs4swampLITE )
1241 } // if ( cbPattern<=NeedleThreshold2vs4swampLITE )
1242 } //if ( cbPattern<4 )
1243 }
1244
1245 // Fixed version from 2012-Feb-27.
1246 // Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
1247 char * Railgun_Doublet (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
1248 {
1249 char * pbTargetMax = pbTarget + cbTarget;
1250 register uint32_t ulHashPattern;
1251 uint32_t ulHashTarget, count, countSTATIC;
1252
1253 if (cbPattern > cbTarget) return(NULL);
1254
1255 countSTATIC = cbPattern-2;
1256
1257 pbTarget = pbTarget+cbPattern;
1258 ulHashPattern = (*(uint16_t *) (pbPattern));
1259
1260 for ( ;; ) {
1261 if ( ulHashPattern == (*(uint16_t *) (pbTarget-cbPattern)) ) {
1262 count = countSTATIC;
1263 while ( count && *(char *) (pbPattern+2+(countSTATIC-count)) == *(char *) (pbTarget-cbPattern+2+(countSTATIC-count)) ) {
1264 count--;
1265 }
1266 if ( count == 0 ) return((pbTarget-cbPattern));
1267 }
1268 pbTarget++;
1269 if (pbTarget > pbTargetMax) return(NULL);
1270 }
1271 }
1272
1273 // Last change: 2014-Apr-25
1274 // If you want to help me to improve it, email me at: sanmayce@sanmayce.com
1275 // Enfun!

```



The extraordinary eight-fold path.

This unique way avoids the two extremes: self-mortification that weakens one's body and self-indulgence that retards one's mind.

It consists of the following eight factors:

- 1) Harmonious perspective (Sammā Diññhi)
- 2) Harmonious feeling (Sammā Saṅkappa)
- 3) Harmonious speech (Sammā Vācā)
- 4) Harmonious action (Sammā Kammanta)
- 5) Harmonious living (Sammā Ajāva)
- 6) Harmonious practice (Sammā Vāyāma)
- 7) Harmonious introspection (Sammā Sati)
- 8) Harmonious equilibrium (Sammā Samādhi)

*/'Treasury of Truth', Illustrated Dhammapada, Author: Ven. Weragoda Sarada Maha Thero/*

1. The best of paths is the Eightfold Path. The best of truths are the four Sayings. Non-attachment is the best of states. The best of bipeds is the Seeing One.
2. This is the only Way. There is none other for the purity of vision. Do you follow this path. This is the bewilderment of Māra.
3. Entering upon that path, you will make an end of pain. Having learnt the removal of thorns, have I taught you the path.
4. Striving should be done by yourselves; the Tathāgatas are only teachers. The meditative ones, who enter the way, are delivered from the bonds of Māra.
5. "All conditions are impermanent:" when one sees this with wisdom, one is disenchanted with suffering; this is the path to purity.
6. "All conditions are unsatisfactory:" when one sees this with wisdom, one is disenchanted with suffering; this is the path to purity.
7. "All phenomena are not-self:" when one sees this with wisdom, one is disenchanted with suffering; this is the path to purity.
8. The inactive idler who strives not when he should strive, who, though young and strong, is slothful, with (good) thoughts depressed, does not by wisdom realise the Path.
9. Watchful of speech, well restrained in mind, let him do nought unskillful through his body. Let him purify these three ways of action and win the path realised by the sages.
10. From meditation arises wisdom. Without meditation wisdom wanes. Knowing this twofold path of gain and loss, let one so conduct oneself so that wisdom increases.
11. Cut down the entire forest, not just a single tree. From the forest springs fear. Cutting down both forest and brushwood, be passionless, O monks.
12. For as long as the slightest passion of man towards women is not cut down, so long is his mind in bondage, like the calf to its mother.
13. Cut off your affection, as though it were an autumn lily, with the hand. Cultivate this path of peace. Nibbāna has been expounded by the Auspicious One.
14. Here will I live in the rainy season, here in the autumn and in the summer: thus muses the fool. He realises not the danger (of death).
15. The doting man with mind set on children and herds, death seizes and carries away, as a great flood (sweeps away) a slumbering village.
16. There are no sons for one's protection, neither father nor even kinsmen; for one who is overcome by death no protection is to be found among kinsmen.
17. Realising this fact, let the virtuous and wise person swiftly clear the way that leads to nibbāna.

*/'The Dhammapada', 20 — Magga Vagga, edited by Bhikkhu Pesala/*

*Are you listening very carefully  
 Close your eyes and come with me  
 To go where the evening comes undone, yeah  
 Late at night in the city streets  
 I feel the need to see the streets  
 I go where you'll always find someone, yeah  
 Is it good night, is it good morning  
 Is this real life, are you performing  
 You're like a vision I can['t] control  
 We're in a movie I'm playing a role  
 I can be by myself sometimes  
 But I won't be by myself tonight, a-a-a  
 The sun comes out and suddenly  
 Something's happening inside of me  
 I wanna know where the silence has come from  
 Is it good night, is it good morning...  
 /Beth Ditto/*

*Nakamichi* (32bit/64bit) performs good well on my 'Bonboniera' laptop (Core 2 T7500 2200MHz, DDR2 dual-channel):

```
D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD>timer32.exe Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_32bit.exe enwik9.Nakamichi
Nakamichi, revision 1-RSSBO_1GB_wordfetcher_TRIAD, written by Kaze, based on Nobuo Ito's LZSS source.
Decompressing 641441055 bytes ...
RAM-to-RAM performance: 678 MB/s.
```

```
Kernel Time = 2.308 = 17%
User Time = 1.107 = 8%
Process Time = 3.416 = 25% Virtual Memory = 1640 MB
Global Time = 13.182 = 100% Physical Memory = 1571 MB
```

```
D:\_KAZE\_KAZE_GOLD\Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD>timer32.exe Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_64bit.exe enwik9.Nakamichi
Nakamichi, revision 1-RSSBO_1GB_wordfetcher_TRIAD, written by Kaze, based on Nobuo Ito's LZSS source.
Decompressing 641441055 bytes ...
RAM-to-RAM performance: 793 MB/s.
```

```
Kernel Time = 2.714 = 21%
User Time = 0.998 = 8%
Process Time = 3.712 = 29% Virtual Memory = 1640 MB
Global Time = 12.402 = 100% Physical Memory = 1570 MB
```

```
D:\_KAZE\_KAZE_GOLD\Nakamichi_vs_Yappy>timer32 Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_32bit.exe Kazahana_on.PAGODA-order-5.txt.Nakamichi
Nakamichi, revision 1-RSSBO_1GB_wordfetcher_TRIAD, written by Kaze, based on Nobuo Ito's LZSS source.
Decompressing 293049398 bytes ...
RAM-to-RAM performance: 782 MB/s.
```

```
Kernel Time = 1.809 = 17%
User Time = 0.842 = 8%
Process Time = 2.652 = 26% Virtual Memory = 1307 MB
Global Time = 10.140 = 100% Physical Memory = 1091 MB
```

```
D:\_KAZE\_KAZE_GOLD\Nakamichi_vs_Yappy>timer32 Nakamichi_r1-RSSBO_1GB_wordfetcher_TRIAD_64bit.exe Kazahana_on.PAGODA-order-5.txt.Nakamichi
Nakamichi, revision 1-RSSBO_1GB_wordfetcher_TRIAD, written by Kaze, based on Nobuo Ito's LZSS source.
Decompressing 293049398 bytes ...
RAM-to-RAM performance: 890 MB/s.
```

```
Kernel Time = 1.981 = 19%
User Time = 0.717 = 6%
Process Time = 2.698 = 26% Virtual Memory = 1307 MB
Global Time = 10.311 = 100% Physical Memory = 1091 MB
```

293,049,398 Kazahana\_on.PAGODA-order-5.txt.Nakamichi  
 846,351,894 Kazahana\_on.PAGODA-order-5.txt