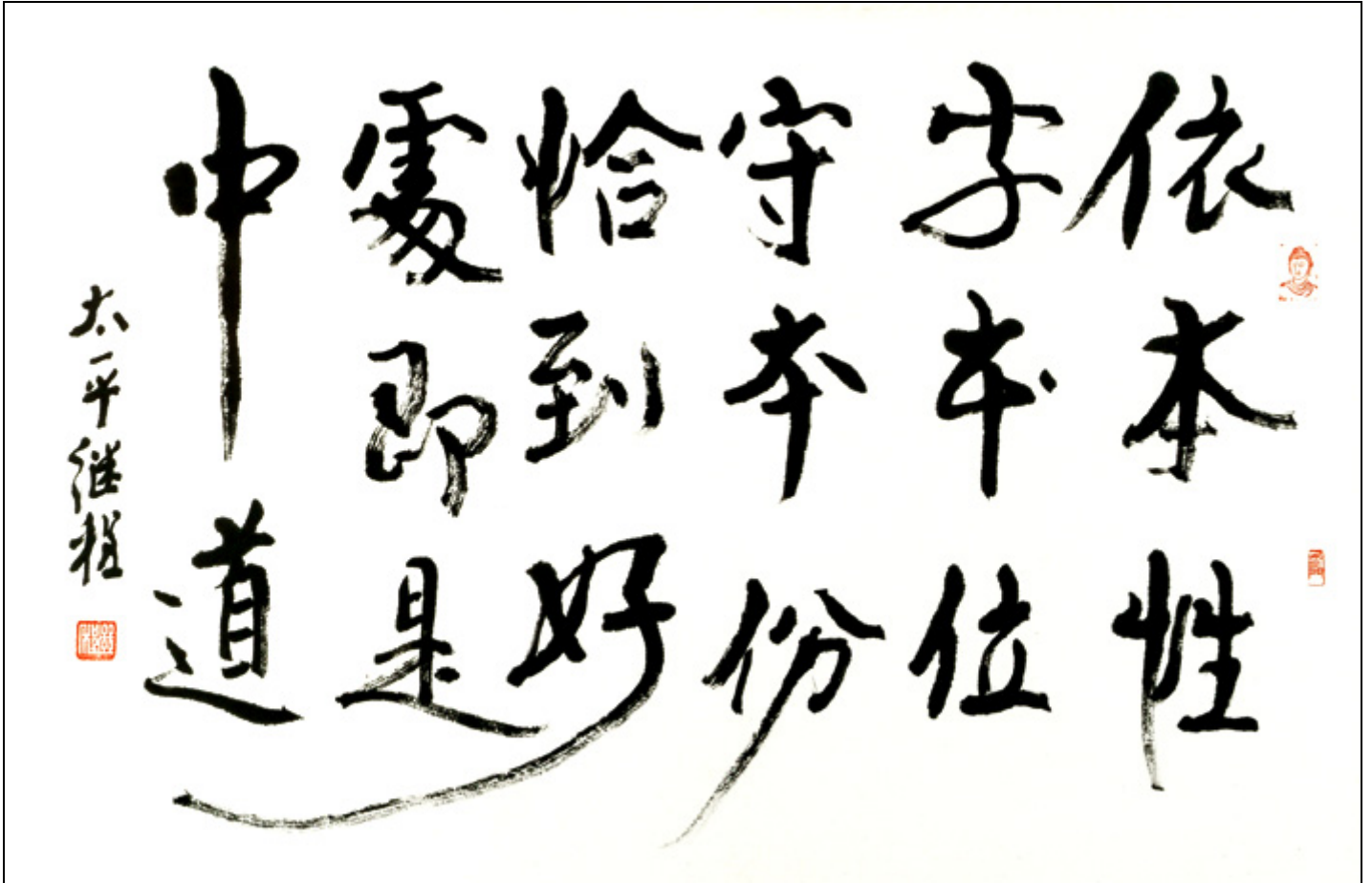


# 中道

## NAKAMICHI



```
001 // Nakamichi is 100% FREE LZSS SUPERFAST decompressor.
002
003 // Nakamichi, revision 1-RSSBO_1GB_wordfetcher_TRIAD, written by Kaze.
004 // Change #1: Decompression fetches WORD instead of BYTE+BYTE.
005 // Change #2: Decompression stores three times 64bit instead of memcpy() for all transfers <=24 bytes.
006 // Change #3: Fifteenth bit is used and then unused, 16KB -> 32KB -> 16KB.
007 // 32KB window disappoints speedwise, also sizewise:
008 /*
009 D:\_KAZE\_KAZE_GOLD\Nakamichi_projectOLD\Nakamichi_vs_Yappy>Nakamichi_r1-RSSBO_1GB_15bit_wordfetcher.exe enwik8
010 Nakamichi, revision 1-RSSBO_1GB_15bit_wordfetcher, written by Kaze, based on Nobuo Ito's LZSS source.
011 Compressing 100000000 bytes ...
012 -; Each rotation means 128KB are encoded; Done 100%
013 RAM-to-RAM performance: 130 KB/s.
014
015 D:\_KAZE\_KAZE_GOLD\Nakamichi_projectOLD\Nakamichi_vs_Yappy>Nakamichi_r1-RSSBO_1GB_15bit_wordfetcher.exe enwik8.Nakamichi
016 Nakamichi, revision 1-RSSBO_1GB_15bit_wordfetcher, written by Kaze, based on Nobuo Ito's LZSS source.
017 Decompressing 65693566 bytes ...
018 RAM-to-RAM performance: 358 MB/s.
019
020 D:\_KAZE\_KAZE_GOLD\Nakamichi_projectOLD\Nakamichi_vs_Yappy>Nakamichi_r1-RSSBO_1GB_15bit_wordfetcher.exe enwik9
021 Nakamichi, revision 1-RSSBO_1GB_15bit_wordfetcher, written by Kaze, based on Nobuo Ito's LZSS source.
022 Compressing 1000000000 bytes ...
023 /; Each rotation means 128KB are encoded; Done 100%
024 RAM-to-RAM performance: 150 KB/s.
025
026 D:\_KAZE\_KAZE_GOLD\Nakamichi_projectOLD\Nakamichi_vs_Yappy>Nakamichi_r1-RSSBO_1GB_15bit_wordfetcher.exe enwik9.Nakamichi
027 Nakamichi, revision 1-RSSBO_1GB_15bit_wordfetcher, written by Kaze, based on Nobuo Ito's LZSS source.
028 Decompressing 609319736 bytes ...
029 RAM-to-RAM performance: 379 MB/s.
030 */
031 // 1-RSSBO_1GB vs 1-RSSBO_1GB_15bit_wordfetcher (16KB/32KB respectively):
032 // 069,443,065 vs 065,693,566
```

```

033 // 641,441,055 vs 609,319,736
034
035 // Nakamichi, revision 1-RSSBO_1GB, written by Kaze.
036 // Based on Nobuo Ito's source, thanks Ito.
037 // The main goal of Nakamichi is to allow supersimple and superfast decoding for English x-grams (mainly) in pure C, or not, heh-heh.
038 // Natively Nakamichi is targeted as 64bit tool with 16 threads, helping Kazahana to traverse faster when I/O is not superior.
039 // In short, Nakamichi is intended as x-gram decompressor.
040
041 // Eightfold Path ~ the Buddhist path to nirvana, comprising eight aspects in which an aspirant must become practised;
042 // eightfold way ~ (Physics), the grouping of hadrons into supermultiplets by means of SU(3)); (b) adverb to eight times the number or quantity: OE.
043
044 // 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.
045 // Note2: English x-grams are as English texts but more redundant, in other words they are phraselists in most cases, sometimes wordlists.
046 // Note3: On OSHO.TXT, being a typical English text, Nakamichi's compression ratio is among the worsts:
047 //      206,908,949 OSHO.TXT
048 //      125,022,859 OSHO.TXT.Nakamichi
049 //      It struggles with English texts but decompression speed is quite sweet (Core 2 T7500 2200MHZ, 32bit code):
050 //      Nakamichi, revision 1-, written by Kaze.
051 //      Decompressing 125022859 bytes ...
052 //      RAM-to-RAM performance: 477681 KB/s.
053 // 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.
054
055 // Quick notes on PAGODAS (the padded x-gram lists):
056 // Every single word in English has its own PAGODA, in example below 'on' PAGODA is given (Kazahana_on.PAGODA-order-5.txt):
057 // 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':
058 /*
059 D:\_KAZE\Nakamichi_r1-RSSBO>dir \_GW\ka*
060
061 04/12/2014  05:07 AM             14 Kazahana_on.1-1.txt
062 04/12/2014  05:07 AM          1,635,389 Kazahana_on.2-1.txt
063 04/12/2014  05:07 AM          1,906,734 Kazahana_on.2-2.txt
064 04/12/2014  05:07 AM          10,891,415 Kazahana_on.3-1.txt
065 04/12/2014  05:07 AM          15,797,703 Kazahana_on.3-2.txt
066 04/12/2014  05:07 AM          20,419,280 Kazahana_on.3-3.txt
067 04/12/2014  05:07 AM          22,141,823 Kazahana_on.4-1.txt
068 04/12/2014  05:07 AM          36,002,113 Kazahana_on.4-2.txt
069 04/12/2014  05:07 AM          33,236,772 Kazahana_on.4-3.txt
070 04/12/2014  05:07 AM          33,902,425 Kazahana_on.4-4.txt
071 04/12/2014  05:07 AM          24,795,989 Kazahana_on.5-1.txt
072 04/12/2014  05:07 AM          30,766,220 Kazahana_on.5-2.txt
073 04/12/2014  05:07 AM          38,982,816 Kazahana_on.5-3.txt
074 04/12/2014  05:07 AM          38,089,575 Kazahana_on.5-4.txt
075 04/12/2014  05:07 AM          34,309,057 Kazahana_on.5-5.txt
076 04/12/2014  05:07 AM          846,351,894 Kazahana_on.PAGODA-order-5.txt
077
078 D:\_KAZE\Nakamichi_r1-RSSBO>type \_GW\Kazahana_on.1-1.txt
079 9,999,999      on
080
081 D:\_KAZE\Nakamichi_r1-RSSBO>type \_GW\Kazahana_on.2-1.txt
082 9,999,999      on_the
083 1,148,054      on_his
084 0,559,694      on_her
085 0,487,856      on_this
086 0,399,485      on_your
087 0,381,570      on_my
088 0,367,282      on_their
089 ...
090
091 D:\_KAZE\Nakamichi_r1-RSSBO>type \_GW\Kazahana_on.2-2.txt
092 0,545,191      based_on
093 0,397,408      and_on
094 0,334,266      go_on
095 0,329,561      went_on
096 0,263,035      was_on
097 0,246,332      it_on
098 0,229,041      down_on
099 0,202,151      going_on
100 ...
101
102 D:\_KAZE\Nakamichi_r1-RSSBO>type \_GW\Kazahana_on.5-5.txt
103 0,083,564      foundation_osho_s_books_on
104 0,012,404      medium_it_may_be_on
105 0,012,354      if_you_received_it_on
106 0,012,152      medium_they_may_be_on
107 0,012,144      agree_to_also_provide_on
108 0,012,139      a_united_states_copyright_on
109 0,008,067      we_are_constantly_working_on
110 0,008,067      questions_we_have_received_on
111 0,006,847      file_was_first_posted_on
112 0,006,441      of_we_are_already_on
113 0,006,279      you_received_this_ebook_on
114 0,005,865      you_received_this_etext_on
115 0,005,833      to_keep_an_eye_on
116 ...
117
118 D:\_KAZE\Nakamichi_r1-RSSBO>dir
119
120 04/12/2014  05:07 AM          846,351,894 Kazahana_on.PAGODA-order-5.txt
121

```

```

122 D:\_KAZE\Nakamichi_r1-RSSBO>Nakamichi.exe Kazahana_on.PAGODA-order-5.txt
123 Nakamichi, revision 1-RSSBO, written by Kaze.
124 Compressing 846351894 bytes ...
125 /; Each rotation means 128KB are encoded; Done 100%
126 RAM-to-RAM performance: 512 KB/s.
127
128 D:\_KAZE\Nakamichi_r1-RSSBO>dir
129
130 04/12/2014  05:07 AM      846,351,894 Kazahana_on.PAGODA-order-5.txt
131 04/15/2014  06:30 PM      293,049,398 Kazahana_on.PAGODA-order-5.txt.Nakamichi
132
133 D:\_KAZE\Nakamichi_r1-RSSBO>Nakamichi.exe Kazahana_on.PAGODA-order-5.txt.Nakamichi
134 Nakamichi, revision 1-RSSBO, written by Kaze.
135 Decompressing 293049398 bytes ...
136 RAM-to-RAM performance: 607 MB/s.
137
138 D:\_KAZE\Nakamichi_r1-RSSBO>Yappy.exe Kazahana_on.PAGODA-order-5.txt 4096
139 YAPPY: [b 4K] bytes 846351894 -> 191149889 22.6% comp 33.8 MB/s uncomp 875.4 MB/s
140
141 D:\_KAZE\Nakamichi_r1-RSSBO>Yappy.exe Kazahana_on.PAGODA-order-5.txt 8192
142 YAPPY: [b 8K] bytes 846351894 -> 184153244 21.8% comp 35.0 MB/s uncomp 898.3 MB/s
143
144 D:\_KAZE\Nakamichi_r1-RSSBO>Yappy.exe Kazahana_on.PAGODA-order-5.txt 16384
145 YAPPY: [b 16K] bytes 846351894 -> 180650931 21.3% comp 28.8 MB/s uncomp 906.4 MB/s
146
147 D:\_KAZE\Nakamichi_r1-RSSBO>Yappy.exe Kazahana_on.PAGODA-order-5.txt 32768
148 YAPPY: [b 32K] bytes 846351894 -> 178902966 21.1% comp 35.0 MB/s uncomp 906.4 MB/s
149
150 D:\_KAZE\Nakamichi_r1-RSSBO>Yappy.exe Kazahana_on.PAGODA-order-5.txt 65536
151 YAPPY: [b 64K] bytes 846351894 -> 178027899 21.0% comp 34.5 MB/s uncomp 914.6 MB/s
152
153 D:\_KAZE\Nakamichi_r1-RSSBO>Yappy.exe Kazahana_on.PAGODA-order-5.txt 131072
154 YAPPY: [b 128K] bytes 846351894 -> 177591807 21.0% comp 34.9 MB/s uncomp 906.4 MB/s
155
156 D:\_KAZE\Nakamichi_r1-RSSBO>
157 */
158
159 #include <stdio.h>
160 #include <stdlib.h>
161 #include <stdint.h> // uint64_t needed
162 #include <time.h>
163 #include <string.h>
164
165 #include <emmintrin.h> // SSE2 intrinsics
166 // #include <smmintrin.h> // SSE4.1 intrinsics
167 // #include <immintrin.h> // AVX intrinsics
168
169 void SlowCopy128bit (const char *SOURCE, char *TARGET) { _mm_storeu_si128((__m128i *) (TARGET), _mm_loadu_si128((const __m128i *) (SOURCE))); }
170
171 #ifndef NULL
172 #define NULL ((void*)0)
173 #endif
174
175 // Comment it to see how slower 'BruteForce' is, for wikipedia 100MB the ratio is 41KB/s versus 197KB/s.
176 #define ReplaceBruteForceWithRailgunSwampshineBailOut
177
178 void SearchIntoSlidingWindow(unsigned int* retIndex, unsigned int* retMatch, char* refStart, char* refEnd, char* encStart, char* encEnd);
179 unsigned int SlidingWindowVsLookAheadBuffer(char* refStart, char* refEnd, char* encStart, char* encEnd);
180 unsigned int Compress(char* ret, char* src, unsigned int srcSize);
181 unsigned int Decompress(char* ret, char* src, unsigned int srcSize);
182 char * Railgun_Swampshine_BailOut(char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern);
183 char * Railgun_Doublet (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern);
184
185 // Min_Match_Length=THRESHOLD=4 means 4 and bigger are to be encoded:
186 #define Min_Match_BAILOUT_Length (8)
187 #define Min_Match_Length (8)
188 #define OffsetBITS (14)
189 #define LengthBITS (1)
190
191 //12bit
192 // #define REF_SIZE (4095+Min_Match_Length)
193 #define REF_SIZE ( ((1<<OffsetBITS)-1) + Min_Match_Length )
194 //3bit
195 // #define ENC_SIZE (7+Min_Match_Length)
196 #define ENC_SIZE ( ((1<<LengthBITS)-1) + Min_Match_Length )
197
198 int main( int argc, char *argv[] ) {
199     FILE *fp;
200     int SourceSize;
201     int TargetSize;
202     char* SourceBlock=NULL;
203     char* TargetBlock=NULL;
204     char* Nakamichi = ".Nakamichi\0";
205     char NewFileName[256];
206     clock_t clocks1, clocks2;
207
208     printf("Nakamichi, revision 1-RSSBO_1GB_Wordfetcher_TRIAD, written by Kaze, based on Nobuo Ito's LZSS source.\n");
209     if (argc==1) {
210         printf("Usage: Nakamichi filename\n"); exit(13);

```

```

211     }
212     if ((fp = fopen(argv[1], "rb")) == NULL) {
213         printf("Nakamichi: Can't open '%s' file.\n", argv[1]); exit(13);
214     }
215     fseek(fp, 0, SEEK_END);
216     SourceSize = ftell(fp);
217     fseek(fp, 0, SEEK_SET);
218     // If filename ends in '.Nakamichi' then mode is decompression otherwise compression.
219     if (strcmp(argv[1]+(strlen(argv[1])-strlen(Nakamichi)), Nakamichi) == 0) {
220         SourceBlock = (char*)malloc(SourceSize+512);
221         //TargetBlock = (char*)malloc(5*SourceSize+512);
222         TargetBlock = (char*)malloc(1024*1024*1024+512);
223         fread(SourceBlock, 1, SourceSize, fp);
224         fclose(fp);
225         printf("Decompressing %d bytes ...\n", SourceSize);
226         clocks1 = clock();
227         TargetSize = Decompress(TargetBlock, SourceBlock, SourceSize);
228         clocks2 = clock();
229         printf("RAM-to-RAM performance: %d MB/s.\n", ((TargetSize/(clocks2 - clocks1 + 1))*(long)1000)>>20);
230         strcpy(NewFileName, argv[1]);
231         *(NewFileName + strlen(argv[1])-strlen(Nakamichi)) = '\0';
232     } else {
233         SourceBlock = (char*)malloc(SourceSize+512);
234         TargetBlock = (char*)malloc(SourceSize+512);
235         fread(SourceBlock, 1, SourceSize, fp);
236         fclose(fp);
237         printf("Compressing %d bytes ...\n", SourceSize);
238         clocks1 = clock();
239         TargetSize = Compress(TargetBlock, SourceBlock, SourceSize);
240         clocks2 = clock();
241         printf("RAM-to-RAM performance: %d KB/s.\n", ((SourceSize/(clocks2 - clocks1 + 1))*(long)1000)>>10);
242         strcpy(NewFileName, argv[1]);
243         strcat(NewFileName, Nakamichi);
244     }
245     if ((fp = fopen(NewFileName, "wb")) == NULL) {
246         printf("Nakamichi: Can't write '%s' file.\n", NewFileName); exit(13);
247     }
248     fwrite(TargetBlock, 1, TargetSize, fp);
249     fclose(fp);
250     free(TargetBlock);
251     free(SourceBlock);
252     exit(0);
253 }
254
255 void SearchIntoSlidingWindow(unsigned int* retIndex, unsigned int* retMatch, char* refStart, char* refEnd, char* encStart, char* encEnd){
256     char* FoundAtPosition;
257     unsigned int match=0;
258     *retIndex=0;
259     *retMatch=0;
260 #ifdef ReplaceBruteForceWithRailgunSwampshineBailOut
261     if (refStart < refEnd) {
262         FoundAtPosition = Railgun_Swampshine_BailOut(refStart, encStart, (uint32_t)(refEnd-refStart), 8);
263         //FoundAtPosition = Railgun_Doublet(refStart, encStart, (uint32_t)(refEnd-refStart), 8);
264         // For bigger windows 'Doublet' is slower:
265         // Nakamichi, revision 1-RSSBO_1GB_15bit performance with 'Swampshine':
266         // Compressing 846351894 bytes ...
267         // RAM-to-RAM performance: 370 KB/s.
268         // Nakamichi, revision 1-RSSBO_1GB_15bit performance with 'Doublet':
269         // Compressing 846351894 bytes ...
270         // RAM-to-RAM performance: 213 KB/s.
271         if (FoundAtPosition!=NULL) {
272             *retMatch=8;
273             *retIndex=refEnd-FoundAtPosition;
274         }
275     }
276 #else
277     while(refStart < refEnd){
278         match=SlidingWindowsLookAheadBuffer(refStart,refEnd,encStart,encEnd);
279         if(match > *retMatch){
280             *retMatch=match;
281             *retIndex=refEnd-refStart;
282         }
283         if(*retMatch >= Min_Match_BAILOUT_Length) break;
284         refStart++;
285     }
286 #endif
287 }
288
289 unsigned int SlidingWindowsLookAheadBuffer( char* refStart, char* refEnd, char* encStart, char* encEnd){
290     int ret = 0;
291     while(refStart[ret] == encStart[ret]){
292         if(&refStart[ret] >= refEnd) break;
293         if(&encStart[ret] >= encEnd) break;
294         ret++;
295         if(ret >= Min_Match_BAILOUT_Length) break;
296     }
297     return ret;
298 }
299

```

```

300 unsigned int Compress(char* ret, char* src, unsigned int srcSize){
301     unsigned int srcIndex=0;
302     unsigned int retIndex=0;
303     unsigned int index=0;
304     unsigned int match=0;
305     unsigned int notMatch=0;
306     unsigned char* notMatchStart=NULL;
307     char* refStart=NULL;
308     char* encEnd=NULL;
309     int Melnitchka=0;
310     char *Auberge[4] = {"\\0", "\\0", "-\\0", "\\0\\0"};
311     int ProgressIndicator;
312
313     while(srcIndex < srcSize){
314         if(srcIndex>=REF_SIZE)
315             refStart=&src[srcIndex-REF_SIZE];
316         else
317             refStart=src;
318         if(srcIndex>=srcSize-ENC_SIZE)
319             encEnd=&src[srcSize];
320         else
321             encEnd=&src[srcIndex+ENC_SIZE];
322
323         SearchIntoSlidingwindow(&index,&match,refStart,&src[srcIndex],&src[srcIndex],encEnd);
324         //if ( match<Min_Match_Length ) {
325         //if ( match<Min_Match_Length || match<8 ) {
326         if ( match==0 ) {
327             if(notMatch==0){
328                 notMatchStart=&ret[retIndex];
329                 retIndex++;
330             }
331             else if (notMatch==127) {
332                 *notMatchStart=(unsigned char)((127)<<1);
333                 notMatch=0;
334                 notMatchStart=&ret[retIndex];
335                 retIndex++;
336             }
337             ret[retIndex]=src[srcIndex];
338             retIndex++;
339             notMatch++;
340             srcIndex++;
341             if ((srcIndex-1) % (1<<17) > srcIndex % (1<<17)) {
342                 ProgressIndicator = (int)( (srcIndex+1)*(float)100/(srcSize+1) );
343                 printf("%s; Each rotation means 128KB are encoded; Done %d%%\r", Auberge[Melnitchka++], ProgressIndicator );
344                 Melnitchka = Melnitchka & 3; // 0 1 2 3: 00 01 10 11
345             }
346         } else {
347             if(notMatch > 0){
348                 *notMatchStart=(unsigned char)((notMatch)<<1);
349                 notMatch=0;
350             }
351             // -----|
352             //          \ /
353
354             //ret[retIndex] = 0x80; // Assuming seventh/fifteenth bit is zero i.e. LONG MATCH i.e. Min_Match_BAILOUT_Length*4
355             //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
356             // -----| \
357             //          \ /
358             ret[retIndex] = 0x01; // Assuming seventh/fifteenth bit is zero i.e. LONG MATCH i.e. Min_Match_BAILOUT_Length*4
359             //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
360             // 1bit+3bits+12bits:
361             //ret[retIndex] = ret[retIndex] | ((match-Min_Match_Length)<<4);
362             //ret[retIndex] = ret[retIndex] | (((index-Min_Match_Length) & 0x0F00)>>8);
363             // 1bit+1bit+14bits:
364             //ret[retIndex] = ret[retIndex] | ((match-Min_Match_Length)<<(8-(Length&BITS+1))); // No need to set the matchlength
365             // 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
366             // -----|
367             //          \ /
368             //ret[retIndex] = ret[retIndex] | (((index-Min_Match_Length) & 0x3F00)>>8); // 2+4+8=14
369             //retIndex++;
370             //ret[retIndex] = (char)((index-Min_Match_Length) & 0x00FF);
371             //retIndex++;
372             // -----| \
373             //          \ /
374             // Now the situation is like LOW:HIGH i.e. FF:3F i.e. 0x3FFF, 16bit&15bit used as flags,
375             // should become LOW:HIGH i.e. FC:FF i.e. 0xFFFF, 2bit&1bit used as flags.
376             ret[retIndex] = ret[retIndex] | (((index-Min_Match_Length) & 0x00FF)<<2); // 6+8=14
377             //ret[retIndex] = ret[retIndex] | (((index-Min_Match_Length) & 0x00FF)<<1); // 7+8=15
378             retIndex++;
379             ret[retIndex] = (char)(((index-Min_Match_Length) & 0x3FFF)>>6);
380             //ret[retIndex] = (char)(((index-Min_Match_Length) & 0x7FFF)>>7);
381             retIndex++;
382             // -----| \
383             //          \ /
384             srcIndex+=match;
385             if ((srcIndex-match) % (1<<17) > srcIndex % (1<<17)) {

```

```

386 ProgressIndicator = (int)( (srcIndex+1)*(float)100/(srcSize+1) );
387 printf("%s; Each rotation means 128KB are encoded; Done %d%%\r", Auberge[MeInitchka++], ProgressIndicator );
388 MeInitchka = MeInitchka & 3; // 0 1 2 3: 00 01 10 11
389     }
390 }
391 }
392 if(notMatch > 0){
393     *notMatchStart=(unsigned char)((notMatch)<<1);
394 }
395 printf("%s; Each rotation means 128KB are encoded; Done %d%%\n", Auberge[MeInitchka], 100 );
396 return retIndex;
397 }
398
399 unsigned int Decompress(char* ret, char* src, unsigned int srcSize){
400     unsigned int srcIndex=0;
401     unsigned int retIndex=0;
402     unsigned int index=0;
403     unsigned int match=0;
404     unsigned int WORDpair;
405
406     while(srcIndex < srcSize){
407         //if((unsigned char)src[srcIndex] <= 127){
408             WORDpair = *(unsigned short int*)&src[srcIndex];
409             if((WORDpair & 0x01) == 0){
410                 if ( ((WORDpair & 0xFF)>>1) > 8*3 ) {
411                     memcpy(&ret[retIndex],&src[srcIndex+1],(WORDpair & 0xFF)>>1); // Use padding and replace 'memcpy' with loop of 4 or 4+4
412 transfers/stores i.e. *()=DWORD
413                 } else {
414                     //SlowCopy128bit((src+srcIndex+1+16*0), (ret+retIndex+16*0));
415                     //SlowCopy128bit((src+srcIndex+1+16*1), (ret+retIndex+16*1));
416                     *(uint64_t*)(ret+retIndex+8*0) = *(uint64_t*)(src+srcIndex+1+8*0);
417                     *(uint64_t*)(ret+retIndex+8*1) = *(uint64_t*)(src+srcIndex+1+8*1);
418                     *(uint64_t*)(ret+retIndex+8*2) = *(uint64_t*)(src+srcIndex+1+8*2);
419                     *(uint64_t*)(ret+retIndex+8*3) = *(uint64_t*)(src+srcIndex+1+8*3);
420                     // Funny, Nikola Tesla is always right, triads rule, for 'Kazahana_on.PAGODA-order-5.txt.Nakamichi':
421                     // For 8*3 GP : RAM-to-RAM performance: 876 MB/s.
422                     // For 8*4 GP : RAM-to-RAM performance: 833 MB/s.
423                     // For 16*2 XMM: RAM-to-RAM performance: 820 MB/s.
424                 }
425                 retIndex+=(WORDpair & 0xFF)>>1;
426                 srcIndex+=(((WORDpair & 0xFF)>>1)+1);
427             }
428         } else{
429             // 1bit+3bits+12bits:
430             //match = ((src[srcIndex] & 0x7F) >> 4)+Min_Match_Length;
431             //index = (src[srcIndex] & 0x0F) << 8;
432             // 1bit+1bit+14bits:
433             //match = ((src[srcIndex] & 0x4F) >> 4)+Min_Match_Length; // In fact, not needed when eightfoldness is commenced, match is 8.
434 // The fragment below is outrageously ineffective - it can be done in one WORD operation instead of two BYTE operations.
435 // -----|
436 //          \
437             //index = (src[srcIndex] & 0x3F) << 8;
438             //srcIndex++;
439             //index = (index | (unsigned int)(0x00FF & src[srcIndex])) + Min_Match_Length;
440             //srcIndex++;
441 // -----|
442 //          \
443 //          \
444             index = (WORDpair>>2) + Min_Match_Length; // 14bit
445             //index = (WORDpair>>1) + Min_Match_Length; // 15bit
446             srcIndex=srcIndex+2;
447 // -----|
448 //          \
449             //if (src[srcIndex-1] & 0x40) { // 4 if seventh/fifteenth bit is not zero
450             //if (WORDpair & 0x02) { // 4 if 2/14 bit is not zero
451                 match = Min_Match_BAILOUT_Length;
452                 //memcpy(&ret[retIndex],&ret[retIndex-index],match);
453                 *(uint32_t*)(ret+retIndex) = *(uint32_t*)(ret+retIndex-index);
454             } else {
455                 match = Min_Match_BAILOUT_Length*4;
456                 /*(uint32_t*)(ret+retIndex) = *(uint32_t*)(ret+retIndex-index);
457                 /*(uint32_t*)(ret+retIndex+4) = *(uint32_t*)(ret+retIndex-index+4);
458                 /*(uint32_t*)(ret+retIndex+8) = *(uint32_t*)(ret+retIndex-index+8);
459                 /*(uint32_t*)(ret+retIndex+12) = *(uint32_t*)(ret+retIndex-index+12);
460                 SlowCopy128bit((ret+retIndex-index), (ret+retIndex));
461             }
462             match = Min_Match_BAILOUT_Length;
463             *(uint64_t*)(ret+retIndex) = *(uint64_t*)(ret+retIndex-index);
464             retIndex+=match;
465         }
466     }
467     return retIndex;
468 }
469
470 // Decompression main loop:
471 /*
472 ; mark_description "Intel(R) C++ Intel(R) 64 Compiler XE for applications running on Intel(R) 64, Version 12.1.1.258 Build 20111";
473 ; mark_description "-O3 -FACS";

```

```

474
475 .B6.3::
476 00045 42 0f b7 34 20 movzx esi, WORD PTR [rax+r12]
477 0004a f7 c6 01 00 00
478 00 test esi, 1
479 00050 75 42 jne .B6.8
480 .B6.4::
481 00052 40 0f b6 f6 movzx esi, si
482 00056 4b 8d 0c 2e lea rcx, QWORD PTR [r14+r13]
483 0005a d1 ee shr esi, 1
484 0005c 83 fe 18 cmp esi, 24
485 0005f 77 1c ja .B6.6
486 .B6.5::
487 00061 4e 8b 5c 20 01 mov r11, QWORD PTR [1+rax+r12]
488 00066 4c 89 19 mov QWORD PTR [rcx], r11
489 00069 4e 8b 5c 20 09 mov r11, QWORD PTR [9+rax+r12]
490 0006e 4a 8b 44 20 11 mov rax, QWORD PTR [17+rax+r12]
491 00073 4c 89 59 08 mov QWORD PTR [8+rcx], r11
492 00077 48 89 41 10 mov QWORD PTR [16+rcx], rax
493 0007b eb 0e jmp .B6.7
494 .B6.6::
495 0007d 8d 53 01 lea edx, DWORD PTR [1+rbx]
496 00080 49 89 f0 mov r8, rsi
497 00083 49 03 d4 add rdx, r12
498 00086 e8 fc ff ff call _intel_fast_memcpy
499 .B6.7::
500 0008b 8d 44 33 01 lea eax, DWORD PTR [1+rbx+rsi]
501 0008f 44 03 ee add r13d, esi
502 00092 eb 1f jmp .B6.9
503 .B6.8::
504 00094 c1 ee 02 shr esi, 2
505 00097 83 c3 02 add ebx, 2
506 0009a 83 c6 08 add esi, 8
507 0009d 48 f7 de neg rsi
508 000a0 49 03 f6 add rsi, r14
509 000a3 89 d8 mov eax, ebx
510 000a5 49 8b 54 35 00 mov rdx, QWORD PTR [r13+rsi]
511 000aa 4b 89 54 35 00 mov QWORD PTR [r13+r14], rdx
512 000af 41 83 c5 08 add r13d, 8
513 .B6.9::
514 000b3 89 c3 mov ebx, eax
515 000b5 3b df cmp ebx, edi
516 000b7 72 8c jb .B6.3
517 */
518
519 // In my opinion Hamid Buzidi is the best, therefore his lzturbo v1.1 reference results are given below:
520 /*
521 D:\_KAZE\Nakamichi_r1-RSSBO>timer32 lzturbo.exe -19 -p0 enwiki-20140304-pages-articles.7z.001 .
522
523 Kernel Time = 0.982 = 0%
524 User Time = 152.537 = 99%
525 Process Time = 153.520 = 100% Virtual Memory = 429 MB
526 Global Time = 153.519 = 100% Physical Memory = 407 MB
527
528 D:\_KAZE\Nakamichi_r1-RSSBO>timer32.exe lzturbo.exe -d enwiki-20140304-pages-articles.7z.001.lzt .
529
530 Kernel Time = 0.234 = 62%
531 User Time = 0.187 = 50%
532 Process Time = 0.421 = 112% Virtual Memory = 98 MB
533 Global Time = 0.374 = 100% Physical Memory = 70 MB
534
535 D:\_KAZE\Nakamichi_r1-RSSBO>dir
536
537 04/15/2014 08:05 AM 104,857,600 enwiki-20140304-pages-articles.7z.001
538 04/15/2014 08:04 AM 41,984,881 enwiki-20140304-pages-articles.7z.001.lzt
539
540 D:\_KAZE\Nakamichi_r1-RSSBO>timer32 lzturbo.exe -11 -p0 enwiki-20140304-pages-articles.7z.001 .
541
542 Kernel Time = 0.171 = 9%
543 User Time = 1.622 = 90%
544 Process Time = 1.794 = 100% Virtual Memory = 58 MB
545 Global Time = 1.794 = 100% Physical Memory = 39 MB
546
547 D:\_KAZE\Nakamichi_r1-RSSBO>timer32.exe lzturbo.exe -d enwiki-20140304-pages-articles.7z.001.lzt .
548
549 Kernel Time = 0.249 = 41%
550 User Time = 0.140 = 23%
551 Process Time = 0.390 = 64% Virtual Memory = 98 MB
552 Global Time = 0.608 = 100% Physical Memory = 73 MB
553
554 D:\_KAZE\Nakamichi_r1-RSSBO>dir
555
556 04/15/2014 08:05 AM 104,857,600 enwiki-20140304-pages-articles.7z.001
557 04/15/2014 08:05 AM 47,685,453 enwiki-20140304-pages-articles.7z.001.lzt
558
559 D:\_KAZE\Nakamichi_r1-RSSBO>
560 */
561
562 // Railgun_Swampshine_BailOut, copleyleft 2014-Jan-31, Kaze.

```

```
// Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
#define NeedleThreshold2vs4swampLITE 9+10 // Should be bigger than 9. BMH2 works up to this value (inclusive), if bigger then BMH4 takes over.
char * Railgun_Swamphshine_BaiOut (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
{
    char * pbTargetMax = pbTarget + cbTarget;
    register uint32_t uHashPattern;
    signed long count;

    unsigned char bm_Horspool_Order2[256*256]; // Bitwise soon...
    uint32_t i, Gulliver;

    uint32_t PRIMALposition, PRIMALpositionCANDIDATE;
    uint32_t PRIMAlength, PRIMAlengthCANDIDATE;
    uint32_t j, FoundAtPosition;

    if (cbPattern > cbTarget) return(NULL);

    if ( cbPattern<4 ) {
        // SSE2 i.e. 128bit Assembly rules here:
        // ...
        pbTarget = pbTarget+cbPattern;
        uHashPattern = ( (*char *)pbPattern)<<8 ) + *(pbPattern+(cbPattern-1));
        if ( cbPattern==3 ) {
            for ( ;; ) {
                if ( uHashPattern == ( (*char *)pbTarget-3)<<8 ) + *(pbTarget-1) ) {
                    if ( (*char *)pbPattern+1 == (*char *)pbTarget-2 ) return((pbTarget-3));
                }
                if ( (char)(uHashPattern>>8) != *(pbTarget-2) ) {
                    pbTarget++;
                    if ( (char)(uHashPattern>>8) != *(pbTarget-2) ) pbTarget++;
                }
                pbTarget++;
                if (pbTarget > pbTargetMax) return(NULL);
            }
        } else {
            for ( ;; ) {
                if ( uHashPattern == ( (*char *)pbTarget-2)<<8 ) + *(pbTarget-1) ) return((pbTarget-2));
                if ( (char)(uHashPattern>>8) != *(pbTarget-1) ) pbTarget++;
                pbTarget++;
                if (pbTarget > pbTargetMax) return(NULL);
            }
        }
    } else { //if ( cbPattern<4 )
        if ( cbPattern<=NeedleThreshold2vs4swampLITE ) {
            // BMH order 2, needle should be >=4:
            uHashPattern = *(uint32_t *)(pbPattern); // First four bytes
            for (i=0; i < 256*256; i++) {bm_Horspool_Order2[i]=0;}
            for (i=0; i < cbPattern-1; i++) bm_Horspool_Order2[(unsigned short *)pbPattern+i]=1;
            i=0;
            while (i <= cbTarget-cbPattern) {
                Gulliver = 1; // 'Gulliver' is the skip
                if ( bm_Horspool_Order2[(unsigned short *)pbTarget[i+cbPattern-1]] != 0 ) {
                    if ( bm_Horspool_Order2[(unsigned short *)pbTarget[i+cbPattern-1-2]] == 0 ) Gulliver = cbPattern-(2-1)-2; else
                        if ( (*(uint32_t *)&bTarget[i] == uHashPattern) { // This fast check ensures not missing a match (for remainder) when going under 0 in loop below:
                            count = cbPattern-4+1;
                            while ( count > 0 && *(uint32_t *)pbPattern+count-1 == *(uint32_t *)&bTarget[i]+(count-1)) )
                                count = count-4;
                            if ( count <= 0 ) return(pbTarget+i);
                        }
                } else Gulliver = cbPattern-(2-1);
                i = i + Gulliver;
                //GlobalI++; // Comment it, it is only for stats.
            }
            return(NULL);
        } else { // if ( cbPattern<=NeedleThreshold2vs4swampLITE )
// Swampwalker_BAILOUT heuristic order 4 (Needle should be bigger than 4) [
// Needle: 1234567890qwertyuiopasdfghjklzxcv          PRIMALposition=01 PRIMALlength=33   '1234567890qwertyuiopasdfghjklzxcv'
// Needle: vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv           PRIMALposition=29 PRIMALlength=04   'vvvv'
// Needle: vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv           PRIMALposition=08 PRIMALlength=20   'vvvBOOMSHAKALAKAvvvv'
// Needle: Trollland                                  PRIMALposition=01 PRIMALlength=09   'Trollland'
// Needle: Swampwalker                                 PRIMALposition=01 PRIMALlength=11   'Swampwalker'
// Needle: licenselessness                             PRIMALposition=01 PRIMALlength=15   'licenselessness'
// Needle: alfalfa                                     PRIMALposition=02 PRIMALlength=06   'lalfal'
// Needle: Sandokan                                    PRIMALposition=01 PRIMALlength=08   'Sandokan'
// Needle: shazamish                                   PRIMALposition=01 PRIMALlength=09   'shazamish'
// Needle: Simplicivus Simplicissimus                 PRIMALposition=06 PRIMALlength=20   'icius Simplicissimus'
// Needle: domilliaquadringenquattuorquinagintillion PRIMALposition=01 PRIMALlength=32   'domilliaquadringenquattuorinqu'
// Needle: boom-boom                                   PRIMALposition=02 PRIMALlength=08   'oom-boom'
// Needle: vvvvv                                       PRIMALposition=01 PRIMALlength=04   'vvvv'
// Needle: 12345                                        PRIMALposition=01 PRIMALlength=05   '12345'
// Needle: likey-likey                               PRIMALposition=03 PRIMALlength=09   'key-likey'
// Needle: BOOOOONM                                      PRIMALposition=03 PRIMALlength=05   'OOOOM'
// Needle: aaaaaB00000OM                              PRIMALposition=02 PRIMALlength=09   'aaaab00000'
// Needle: B0000OMaaaaaa                              PRIMALposition=03 PRIMALlength=09   '0000maaaa'
```



```

649 PRIMALlength=0;
650 for (i=0+(1); i < cbPattern-((4)-1)+(1)-(1); i++) { // -(1) because the last BB order 4 has no counterpart(s)
651     FoundAtPosition = cbPattern - ((4)-1) + 1;
652     PRIMALpositionCANDIDATE=i;
653     while ( PRIMALpositionCANDIDATE <= (FoundAtPosition-1) ) {
654         j = PRIMALpositionCANDIDATE + 1;
655         while ( j <= (FoundAtPosition-1) ) {
656             if ( *(uint32_t *) (pbPattern+PRIMALpositionCANDIDATE-(1)) == *(uint32_t *) (pbPattern+j-(1)) ) FoundAtPosition = j;
657             j++;
658         }
659         PRIMALpositionCANDIDATE++;
660     }
661     PRIMALlengthCANDIDATE = (FoundAtPosition-1)-i+1+((4)-1);
662     if (PRIMALlengthCANDIDATE >= PRIMALlength) {PRIMALposition=i; PRIMALlength = PRIMALlengthCANDIDATE;}
663     if (cbPattern-i+1 <= PRIMALlength) break;
664     if (PRIMALlength > 128) break; // Bail Out for 129[+]
665 }
666 // Swampwalker_BAILOUT heuristic order 4 (Needle should be bigger than 4) ]
667
668 // 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
669 'cbPattern':
670 PRIMALlengthCANDIDATE = cbPattern;
671 cbPattern = PRIMALlength;
672 pbPattern = pbPattern + (PRIMALposition-1);
673
674 // Revision 2 commented section [
675 /*
676 if (cbPattern-1 <= 255) {
677     // BMH Order 2 [
678     u1HashPattern = *(uint32_t *) (pbPattern); // First four bytes
679     for (i=0; i < 256*256; i++) {bm_Horspool_Order2[i]= cbPattern-1;} // cbPattern-(Order-1) for Horspool; 'memset' if not optimized
680     for (i=0; i < cbPattern-1; i++) bm_Horspool_Order2[*(unsigned short *) (pbPattern+i)]=i; // Rightmost appearance/position is needed
681     i=0;
682     while (i <= cbTarget-cbPattern) {
683         Gulliver = bm_Horspool_Order2[*(unsigned short *) &pbTarget[i+cbPattern-1]];
684         if ( Gulliver != cbPattern-1 ) { // CASE #2: if equal means the pair (char order 2) is not found i.e. Gulliver remains
685             intact, skip the whole pattern and fall back (Order-1) chars i.e. one char for Order 2
686             if ( Gulliver == cbPattern-2 ) { // CASE #1: means the pair (char order 2) is found
687                 if ( *(uint32_t *) &pbTarget[i] == u1HashPattern ) {
688                     count = cbPattern-4+1;
689                     while ( count > 0 && *(uint32_t *) (pbPattern+count-1) == *(uint32_t *) (&pbTarget[i]+(count-1)) )
690                         count = count-4;
691                     // If we miss to hit then no need to compare the original: Needle
692                     if ( count <= 0 ) {
693                         // I have to add out-of-range checks...
694                         // i-(PRIMALposition-1) >= 0
695                         // &pbTarget[i-(PRIMALposition-1)] <= pbTargetMax - 4
696                         // i-(PRIMALposition-1)+(count-1) >= 0
697                         // &pbTarget[i-(PRIMALposition-1)+(count-1)] <= pbTargetMax - 4
698                         if ( (i-(PRIMALposition-1) >= 0) && (&pbTarget[i-(PRIMALposition-1)] <= pbTargetMax - 4) && (&pbTarget[i-(PRIMALposition-1)+(count-1)] <= pbTargetMax - 4) ) {
699                             if ( *(uint32_t *) &pbTarget[i-(PRIMALposition-1)] == *(uint32_t *) (pbPattern-(PRIMALposition-1)) ) { // This fast check ensures not missing a match
700                                 (for remainder) when going under 0 in loop below:
701                                 count = PRIMALlengthCANDIDATE-4+1;
702                                 while ( count > 0 && *(uint32_t *) (pbPattern-(PRIMALposition-1)+count-1) == *(uint32_t *) (&pbTarget[i-(PRIMALposition-1)]+(count-1)) )
703                                     count = count-4;
704                                 if ( count <= 0 ) return(pbTarget+i-(PRIMALposition-1));
705                             }
706                         }
707                     } else
708                         Gulliver = 1;
709                     } else
710                         Gulliver = cbPattern - Gulliver - 2; // CASE #3: the pair is found and not as suffix i.e. rightmost position
711                     }
712                     i = i + Gulliver;
713                     //GlobalI++; // Comment it, it is only for stats.
714                 }
715                 return(NULL);
716             }
717             // BMH order 2 ]
718             } else {
719                 // BMH order 2, needle should be >=4:
720                 u1HashPattern = *(uint32_t *) (pbPattern); // First four bytes
721                 for (i=0; i < 256*256; i++) {bm_Horspool_Order2[i]=0;}
722                 for (i=0; i < cbPattern-1; i++) bm_Horspool_Order2[*(unsigned short *) (pbPattern+i)]=i;
723                 i=0;
724                 while (i <= cbTarget-cbPattern) {
725                     Gulliver = 1; // 'Gulliver' is the skip
726                     if ( bm_Horspool_Order2[*(unsigned short *) &pbTarget[i+cbPattern-1]] != 0 ) {
727                         if ( bm_Horspool_Order2[*(unsigned short *) &pbTarget[i+cbPattern-1-2]] == 0 ) Gulliver = cbPattern-(2-1)-2; else
728                         if ( *(uint32_t *) &pbTarget[i] == u1HashPattern ) { // This fast check ensures not missing a match (for
729                             remainder) when going under 0 in loop below:
730                             count = cbPattern-4+1;
731                             while ( count > 0 && *(uint32_t *) (pbPattern+count-1) == *(uint32_t *) (&pbTarget[i]+(count-
732                                 1)) )
733                                 count = count-4;
734                             // If we miss to hit then no need to compare the original: Needle
735                             if ( count <= 0 ) {
736                                 // I have to add out-of-range checks...

```

```

732 // i-(PRIMALposition-1) >= 0
733 // &pbTarget[i-(PRIMALposition-1)] <= pbTargetMax - 4
734 // i-(PRIMALposition-1)+(count-1) >= 0
735 // &pbTarget[i-(PRIMALposition-1)+(count-1)] <= pbTargetMax - 4
736 if ( (i-(PRIMALposition-1) >= 0) && (&pbTarget[i-(PRIMALposition-1)] <= pbTargetMax - 4) && (&pbTarget[i-(PRIMALposition-1)+(count-1)] <= pbTargetMax - 4) ) {
737     if ( *(uint32_t *)&pbTarget[i-(PRIMALposition-1)] == *(uint32_t *)&pbPattern-(PRIMALposition-1)) { // This fast check ensures not missing a match
738         (for remainder) when going under 0 in loop below:
739         count = PRIMALlengthCANDIDATE-4+1;
740         while ( count > 0 && *(uint32_t *)&pbPattern-(PRIMALposition-1)+count-1 == *(uint32_t *)&pbTarget[i-(PRIMALposition-1)]+(count-1) )
741             count = count-4;
742         if ( count <= 0 ) return(pbTarget+i-(PRIMALposition-1));
743     }
744 }
745 }
746 }
747 } else Gulliver = cbPattern-(2-1);
748 i = i + Gulliver;
749 //GlobalI++; // Comment it, it is only for stats.
750 }
751 return(NULL);
752 }
753 */
754 // Revision 2 commented section ]
755
756 if ( cbPattern<=NeedleThreshold2vs4swampLITE ) {
757
758     // BMH order 2, needle should be >=4:
759     ulHashPattern = *(uint32_t *)&pbPattern); // First four bytes
760     for (i=0; i < 256*256; i++) {bm_Horspool_Order2[i]=0;}
761     for (i=0; i < cbPattern-1; i++) bm_Horspool_Order2[(unsigned short *)&pbPattern+i]=1;
762     i=0;
763     while (i <= cbTarget-cbPattern) {
764         Gulliver = 1; // 'Gulliver' is the skip
765         if ( bm_Horspool_Order2[(unsigned short *)&pbTarget[i+cbPattern-1-1]] != 0 ) {
766             if ( bm_Horspool_Order2[(unsigned short *)&pbTarget[i+cbPattern-1-2]] == 0 ) Gulliver = cbPattern-(2-1)-2; else
767             {
768                 if ( *(uint32_t *)&pbTarget[i] == ulHashPattern) { // This fast check ensures not missing a match (for
769                     (for remainder) when going under 0 in loop below:
770                     count = cbPattern-4+1;
771                     while ( count > 0 && *(uint32_t *)&pbPattern+count-1 == *(uint32_t *)&pbTarget[i+(count-
772                         1)) )
773                         count = count-4;
774                     // If we miss to hit then no need to compare the original: Needle
775                     if ( count <= 0 ) {
776                         // I have to add out-of-range checks...
777                         // i-(PRIMALposition-1) >= 0
778                         // &pbTarget[i-(PRIMALposition-1)] <= pbTargetMax - 4
779                         // i-(PRIMALposition-1)+(count-1) >= 0
780                         // &pbTarget[i-(PRIMALposition-1)+(count-1)] <= pbTargetMax - 4
781                         if ( (i-(PRIMALposition-1) >= 0) && (&pbTarget[i-(PRIMALposition-1)] <= pbTargetMax - 4) && (&pbTarget[i-(PRIMALposition-1)+(count-1)] <= pbTargetMax - 4) ) {
782                             if ( *(uint32_t *)&pbTarget[i-(PRIMALposition-1)] == *(uint32_t *)&pbPattern-(PRIMALposition-1)) { // This fast check ensures not missing a match
783                                 (for remainder) when going under 0 in loop below:
784                                 count = PRIMALlengthCANDIDATE-4+1;
785                                 while ( count > 0 && *(uint32_t *)&pbPattern-(PRIMALposition-1)+count-1 == *(uint32_t *)&pbTarget[i-(PRIMALposition-1)]+(count-1) )
786                                     count = count-4;
787                                 if ( count <= 0 ) return(pbTarget+i-(PRIMALposition-1));
788                             }
789                         }
790                     } else Gulliver = cbPattern-(2-1);
791                     i = i + Gulliver;
792                     //GlobalI++; // Comment it, it is only for stats.
793                 }
794                 return(NULL);
795             } else { // if ( cbPattern<=NeedleThreshold2vs4swampLITE )
796
797                 // BMH pseudo-order 4, needle should be >=8+2:
798                 ulHashPattern = *(uint32_t *)&pbPattern); // First four bytes
799                 for (i=0; i < 256*256; i++) {bm_Horspool_Order2[i]=0;}
800                 // 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
801                 // of BBs for text 'cbPattern' bytes long, for example, for cbPattern=11 'fastest fox' and Order=4 we have BBs = 11-4+1=8:
802                 // "fast"
803                 // "aste"
804                 // "stes"
805                 // "test"
806                 // "est "
807                 // "st f"
808                 // "t fo"
809                 // " fox"
810                 // for (i=0; i < cbPattern-4+1; i++) bm_Horspool_Order2[( (unsigned short *)&pbPattern+i+0) + *(unsigned short *)&pbPattern+i+2) ) & (
811                 // for (i=0; i < cbPattern-4+1; i++) bm_Horspool_Order2[( (uint32_t *)&pbPattern+i+0)>>16)+(uint32_t *)&pbPattern+i+0)&0xFFFF) ) & (
812                 // Above line is replaced by next one with better hashing:
813                 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;
813         i=0;
814         while (i <= cbTarget-cbPattern) {
815             Gulliver = 1;
816             //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
817                 // Above line is replaced by next one with better hashing:
818                 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
819                     //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 {
820                         // 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'.
821                         // The respective offsets (backwards) are: -10/-8/-6/-4 for 'xa'/'ab'/'bc'/'cd'.
822                         //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 {
823                             // Above line is replaced by next one with better hashing:
824                             // 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 especially for ASCII texts with no symbols in range 128-255.
825                             // Example for genomesque pair TT+TT being shifted by (16-1):
826                             // T           = 01010100
827                             // TT          = 01010100 01010100
828                             // TTTT       = 01010100 01010100 01010100 01010100
829                             // TTTT>>16   = 00000000 00000000 01010100 01010100
830                             // 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.
831                             //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 {
832                                 // 'Maximus' uses branched 'if', again.
833                                 if ( \
834                                     ( 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 \
835                                     || ( 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 \
836                                     ) Gulliver = cbPattern-(2-1)-2-4-2 +1; else {
837                                     // 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:
838                                     // [2x][2a][2b][2c][2d]
839                                     // DWORD #4
840                                     // [2a] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-6]>>16) = !SHR to be avoided! <--
841                                     // [2x] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-6]&0xFFFF) = -----
842                                     // DWORD #3
843                                     // [2b] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]>>16) = !SHR to be avoided! <--
844                                     // [2a] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) = -----
845                                     // DWORD #2
846                                     // [2c] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-2]>>16) = !SHR to be avoided! <--
847                                     // [2b] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-2]&0xFFFF) = -----
848                                     // DWORD #1
849                                     // [2d] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-0]>>16) = -----
850                                     // [2c] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-0]&0xFFFF) = -----
851                                     // So in order to remove 3 SHR instructions the equal extractions are:
852                                     // DWORD #4
853                                     // [2a] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) = !SHR to be avoided! <--
854                                     // [2x] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-6]&0xFFFF) = -----
855                                     // DWORD #3
856                                     // [2b] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-2]&0xFFFF) = !SHR to be avoided! <--
857                                     // [2a] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-4]&0xFFFF) = -----
858                                     // DWORD #2
859                                     // [2c] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-0]&0xFFFF) = !SHR to be avoided! <--
860                                     // [2b] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-2]&0xFFFF) = -----
861                                     // DWORD #1
862                                     // [2d] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-0]>>16) = -----
863                                     // [2c] (*(uint32_t *)&pbTarget[i+cbPattern-1-1-2-0]&0xFFFF) = -----
864                                     //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 {
865                                         // Since the above Decumanus mumbo-jumbo (3 overlapping lookups vs 2 non-overlapping lookups) is not fast enough we go DuoDecumanus or 3x4:
866                                         // [2y][2x][2a][2b][2c][2d]
867                                         // DWORD #3
868                                         // DWORD #2
869                                         // DWORD #1
870                                         //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 {
871                                             if ( *(uint32_t *)&pbTarget[i] == ulHashPattern) {
872                                                 // Order 4 [
873                                                 // Let's try something "outrageous" like comparing with[out] overlap BBS 4bytes long instead of 1 byte
back-to-back:
874                                                 // 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:
875                                                 //0:"fast" if the comparison failed here, 'count' is 1; 'Gulliver' is cbPattern-(4-1)-7
876                                                 //1:"aste" if the comparison failed here, 'count' is 2; 'Gulliver' is cbPattern-(4-1)-6
877                                                 //2:"stes" if the comparison failed here, 'count' is 3; 'Gulliver' is cbPattern-(4-1)-5
878

```

```

879 //3:"test" if the comparison failed here, 'count' is 4; 'Gulliver' is cbPattern-(4-1)-4
880 //4:"est " if the comparison failed here, 'count' is 5; 'Gulliver' is cbPattern-(4-1)-3
881 //5:"st f" if the comparison failed here, 'count' is 6; 'Gulliver' is cbPattern-(4-1)-2
882 //6:"t fo" if the comparison failed here, 'count' is 7; 'Gulliver' is cbPattern-(4-1)-1
883 //7:" fox" if the comparison failed here, 'count' is 8; 'Gulliver' is cbPattern-(4-1)
884 count = cbPattern-4+1;
885 // Below comparison is UNIDirectional:
886 while ( count > 0 && *(uint32_t *) (pbPattern+count-1) == *(uint32_t *) (&pbTarget[i]+(count-
1)) )
887 count = count-4;
888 // count = cbPattern-4+1 = 23-4+1 = 20
889 // boomshakalakazzzzzz[zzzz] 20
890 // boomshakalakazz[zzzz]zzzz 20-4
891 // boomshakala[kazz]zzzzzzzz 20-8 = 12
892 // boomsha[kala]kazzzzzzzzzz 20-12 = 8
893 // boo[msha]kalakazzzzzzzzzz 20-16 = 4
894
895 // If we miss to hit then no need to compare the original: Needle
896 if ( count <= 0 ) {
897 // I have to add out-of-range checks...
898 // i-(PRIMALposition-1) >= 0
899 // &pbTarget[i-(PRIMALposition-1)] <= pbTargetMax - 4
900 // i-(PRIMALposition-1)+(count-1) >= 0
901 // &pbTarget[i-(PRIMALposition-1)+(count-1)] <= pbTargetMax - 4
902 if ( (i-(PRIMALposition-1) >= 0) && (&pbTarget[i-(PRIMALposition-1)] <= pbTargetMax - 4) && (&pbTarget[i-(PRIMALposition-1)+(count-1)] <= pbTargetMax - 4) ) {
903 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:
904 count = PRIMALlengthCANDIDATE-4+1;
905 while ( count > 0 && *(uint32_t *) (pbPattern-(PRIMALposition-1)+count-1) == *(uint32_t *) (&pbTarget[i-(PRIMALposition-1)]+(count-1)) )
906 count = count-4;
907 if ( count <= 0 ) return(pbTarget+i-(PRIMALposition-1));
908 }
909 }
910 }
911
912 // 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.
913 // Below comparison is BIDirectional. It pays off when needle is 8+++ long:
914 for (count = cbPattern-4+1; count > 0; count = count-4) {
915 if ( *(uint32_t *) (pbPattern+count-1) != *(uint32_t *) (&pbTarget[i]+(count-1)) )
916 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] == u1HashPattern)
917 // }
918 // if ( count <= 0 ) return(pbTarget+i);
919 // Checking the order 2 pairs in mismatched DWORD, all the 3:
920 //if ( bm_Horspool_Order2[*(unsigned short *) &pbTarget[i+count-1]] == 0 )
921 //if ( bm_Horspool_Order2[*(unsigned short *) &pbTarget[i+count-1+1]] == 0 )
922 //if ( bm_Horspool_Order2[*(unsigned short *) &pbTarget[i+count-1+1+1]] == 0 )
923 // 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+1]] < 3 ) Gulliver = count; // 1 or bigger,
as it should, THE MIN(count,count+1,count+1+1)
924 // Above compound 'if' guarantees not that Gulliver > 1, an example:
925 // Needle: fastest tax
926 // window: ...fastest tax...
927 // After matching 'tax' vs 'tax' and 'fast' vs 'fast' the mismatched DWORD is
'test' vs 'tast':
928 // '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.
929 // 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'.
930 //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
931 // Above line is replaced by next one with better hashing:
932 //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
933 // Order 4 ]
934 }
935 }
936 } else Gulliver = cbPattern-(2-1)-2; // -2 because we check the 4 rightmost bytes not 2.
937 i = i + Gulliver;
938 //GlobalI++; // Comment it, it is only for stats.
939 }
940 return(NULL);
941
942 // if ( cbPattern<=NeedleThreshold2vs4swampLITE )
943 // if ( cbPattern<=NeedleThreshold2vs4swampLITE )
944 // if ( cbPattern<4 )
945 }
946
947 // Fixed version from 2012-Feb-27.
948 // Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
949 char * Railgun_Doublet (char * pbTarget, char * pbPattern, uint32_t cbTarget, uint32_t cbPattern)
950 {
951 char * pbTargetMax = pbTarget + cbTarget;

```

```

952     register uint32_t ulHashPattern;
953     uint32_t ulHashTarget, count, countSTATIC;
954
955     if (cbPattern > cbTarget) return(NULL);
956
957     countSTATIC = cbPattern-2;
958
959     pbTarget = pbTarget+cbPattern;
960     ulHashPattern = (*(uint16_t *) (pbPattern));
961
962     for ( ;; ) {
963         if ( ulHashPattern == (*(uint16_t *) (pbTarget-cbPattern)) ) {
964             count = countSTATIC;
965             while ( count && *(char *) (pbPattern+2+(countSTATIC-count)) == *(char *) (pbTarget-cbPattern+2+(countSTATIC-count)) ) {
966                 count--;
967             }
968             if ( count == 0 ) return((pbTarget-cbPattern));
969         }
970         pbTarget++;
971         if (pbTarget > pbTargetMax) return(NULL);
972     }
973 }
974
975 // Last change: 2014-Apr-20
976 // If you want to help me to improve it, email me at: sanmayce@sanmayce.com
977 // 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/*