

```

$LL74@main:
a1 00 00 00 00  mov    eax, DWORD PTR _Patternlen
03 c2           add    eax, edx
8b 04 30       mov    eax, DWORD PTR [eax+esi]
25 ff ff ff 00  and    eax, 16777215
0f af c7       imul  eax, edi
8b 1c 30       mov    ebx, DWORD PTR [eax+esi]
89 19         mov    DWORD PTR [ecx], ebx
8b 44 30 04    mov    eax, DWORD PTR [eax+esi+4]
89 41 04       mov    DWORD PTR [ecx+4], eax
83 c2 03       add    edx, 3
03 cf         add    ecx, edi
ff 4c 24 1c    dec   DWORD PTR tv977[esp+376]
75 d7         jne   SHORT $LL74@main

```

Note2: The niftiness lies in readiness for multi-threading and mostly in boosting the search by having the BBs.

The decompressor would upload (at burst speed) the BB data and then read-and-decode one-by-one the triads (BB pool/array indexes), that is a simple copying. The pool houses up to 256\*256\*256 BBs/elements.

Note7: My benchmark text file OSHO.TXT 206,908,949 bytes where OSHO.TXT.SS 116,871,584 bytes for order 6: Decompressing OSHO.TXT.SS to RAM without Dumping to DRIVE time: 1704 clocks or 118579 KB/s, an awful result.

For order 4 enforced: 156,174,067 OSHO.TXT.SS is being decompressed at 192804 KB/s.

For order 7 enforced: 122,297,608 OSHO.TXT.SS is being decompressed at 85618 KB/s.

For order 8 enforced: 149,243,106 OSHO.TXT.SS is being decompressed at 115396 KB/s.

Obviously the fastest cache size is crucial, for OSHO.TXT 12MB BB pool vs 1MB L2 cache disbalance is the cause for this badly inferior performance compared to LZ L1 (32KB) cache-friendly variants. The testing machine is Toshiba Satellite with Intel Merom 2166MHz.

Note9: Major (but still inferior) decompressing tweak since r.1+++ , this time Microsoft v16 excels: For order 7 enforced: 122,297,608 OSHO.TXT.SS is being decompressed at 170083 KB/s.

# SIMPLICIUS SIMPLICISSIMUS

A BUILDING-BLOCKS TEXT DECOMPRESSOR, REVISION 2-

Free download at [www.sanmayce.com](http://www.sanmayce.com) – on Intel Merom-1M 2166 MHz it decompresses OSHO.TXT.SS at 159MB/s.

```
$LL74@main:
a1 00 00 00 00  mov     eax, DWORD PTR _Patternlen
03 c2           add     eax, edx
8b 04 30       mov     eax, DWORD PTR [eax+esi]
25 ff ff ff 00  and     eax, 16777215
0f af c7       imul   eax, edi
8b 1c 30       mov     ebx, DWORD PTR [eax+esi]
89 19         mov     DWORD PTR [ecx], ebx
8b 44 30 04    mov     eax, DWORD PTR [eax+esi+4]
89 41 04       mov     DWORD PTR [ecx+4], eax
83 c2 03       add     edx, 3
03 cf         add     ecx, edi
ff 4c 24 1c    dec     DWORD PTR tv977[esp+376]
75 d7         jne     SHORT $LL74@main
```

Note2: The niftiness lies in readiness for multi-threading and mostly in boosting the search by having the BBs.

The decompressor would upload (at burst speed) the BB data and then read-and-decode one-by-one the triads (BB pool/array indexes), that is a simple copying. The pool houses up to 256\*256\*256 BBs/elements.

Note7: My benchmark text file OSHO.TXT 206,908,949 bytes where OSHO.TXT.SS 116,871,584 bytes for order 6: Decompressing OSHO.TXT.SS to RAM without Dumping to DRIVE time: 1704 clocks or 118579 KB/s, an awful result.

For order 4 enforced: 156,174,067 OSHO.TXT.SS is being decompressed at 192804 KB/s.

For order 7 enforced: 122,297,608 OSHO.TXT.SS is being decompressed at 85618 KB/s.

For order 8 enforced: 149,243,106 OSHO.TXT.SS is being decompressed at 115396 KB/s.

Obviously the fastest cache size is crucial, for OSHO.TXT 12MB BB pool vs 1MB L2 cache disbalance is the cause for this badly inferior performance compared to LZ L1 (32KB) cache-friendly variants. The testing machine is Toshiba Satellite with Intel Merom 2166MHz.

Note9: Major (but still inferior) decompressing tweak since r.1+++ , this time Microsoft v16 excels: For order 7 enforced: 122,297,608 OSHO.TXT.SS is being decompressed at 170083 KB/s.

# SIMPLICIUS SIMPLICISSIMUS

A BUILDING-BLOCKS TEXT DECOMPRESSOR, REVISION 2-

Free download at [www.sanmayce.com](http://www.sanmayce.com) – on Intel Merom-1M 2166 MHz it decompresses OSHO.TXT.SS at 159MB/s.

```
0001 // Building-Blocks_DUMPER rev.2-
0002
0003 /*
0004 C:\workTemp\LEPREC-1\VISUAL-1\LEA56D-1>dir osho.txt
0005 Volume in drive C is H320_Vo12
0006 Volume Serial Number is A094-FAE2
0007
0008 Directory of C:\workTemp\LEPREC-1\VISUAL-1\LEA56D-1
0009
0010 10/04/2010  06:32 AM          18 osho.txt
0011             1 File(s)          18 bytes
0012             0 Dir(s)  1,159,110,656 bytes free
0013
0014 C:\workTemp\LEPREC-1\VISUAL-1\LEA56D-1>type osho.txt
0015 bxr bxr boban dodo
0016 C:\workTemp\LEPREC-1\VISUAL-1\LEA56D-1>c1 /Ox Building-Blocks_DUMPER.c
0017 Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86
0018 Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.
0019
0020 Building-Blocks_DUMPER.c
0021 Microsoft (R) Incremental Linker Version 7.10.3077
0022 Copyright (C) Microsoft Corporation. All rights reserved.
0023
0024 /out:Building-Blocks_DUMPER.exe
0025 Building-Blocks_DUMPER.obj
0026
0027 C:\workTemp\LEPREC-1\VISUAL-1\LEA56D-1>Building-Blocks_DUMPER.exe
0028 Building-Blocks_DUMPER rev.1, written by Kaze.
0029 Sorting 16 Pointers to Building-Blocks 3 chars in size ...
0030 Allocated memory for pointers-to-words in MB: 1
0031 Writing Sorted Building-Blocks to BB003.txt ...
0032 3|18-3+1|13|3
0033 Sorting 15 Pointers to Building-Blocks 4 chars in size ...
0034 Allocated memory for pointers-to-words in MB: 1
0035 Writing Sorted Building-Blocks to BB004.txt ...
0036 4|18-4+1|13|2
0037 Sorting 14 Pointers to Building-Blocks 5 chars in size ...
0038 Allocated memory for pointers-to-words in MB: 1
0039 Writing Sorted Building-Blocks to BB005.txt ...
0040 5|18-5+1|13|1
0041
0042 C:\workTemp\LEPREC-1\VISUAL-1\LEA56D-1>type BB003.txt
0043 bo
0044 bx
0045 do
0046 an
0047 ban
0048 bob
0049 bxr
0050 dod
0051 n d
0052 oba
0053 odo
0054 r b
0055 xr
0056
0057 C:\workTemp\LEPREC-1\VISUAL-1\LEA56D-1>type BB004.txt
0058 bob
0059 bxr
0060 dod
0061 an d
0062 ban
0063 boban
0064 bxr
0065 dodo
0066 n do
0067 oban
0068 r bo
0069 r bx
0070 xr b
0071
0072 C:\workTemp\LEPREC-1\VISUAL-1\LEA56D-1>type BB005.txt
0073 boban
0074 bxr
0075 dodo
0076 an do
0077 ban d
0078 boban
0079 bxr b
0080 n dod
0081 oban
0082 r bob
0083 r bxr
0084 xr bo
0085 xr bx
0086
0087 C:\workTemp\LEPREC-1\VISUAL-1\LEA56D-1>
0088 */
```

```
0089
0090 /*
0091 C:\workTemp\LEPREC-1\VISUAL-1\LEA56D-1>type osho.txt
0092 bxr bxr boban dodo
0093 C:\workTemp\LEPREC-1\VISUAL-1\LEA56D-1>dir osho.txt
0094 18 osho.txt
0095
0096 C:\workTemp\LEPREC-1\VISUAL-1\LEA56D-1>c1 /Ox Building-Blocks_DUMPER.c
0097 Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86
0098 Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.
0099
0100 Building-Blocks_DUMPER.c
0101 Microsoft (R) Incremental Linker Version 7.10.3077
0102 Copyright (C) Microsoft Corporation. All rights reserved.
0103
0104 /out:Building-Blocks_DUMPER.exe
0105 Building-Blocks_DUMPER.obj
0106
0107 C:\workTemp\LEPREC-1\VISUAL-1\LEA56D-1>Building-Blocks_DUMPER.exe
0108 Building-Blocks_DUMPER rev.1, written by Kaze.
0109 Sorting 16 Pointers to Building-Blocks 3 chars in size ...
0110 Allocated memory for pointers-to-words in MB: 1
0111 Writing Sorted Building-Blocks to BB003.txt ...
0112
0113 In %c:
0114 C:\workTemp\LEPREC-1\VISUAL-1\LEA56D-1>type BB003.txt
0115 bo
0116 bx
0117 do
0118 an
0119 ban
0120 bob
0121 bxr
0122 bxr
0123 dod
0124 n d
0125 oba
0126 odo
0127 r b
0128 r b
0129 xr
0130 xr
0131
0132 In HEX:
0133 C:\workTemp\LEPREC-1\VISUAL-1\LEA56D-1>type BB003.txt
0134 20626f
0135 206278
0136 20646f
0137 616e20
0138 62616e
0139 626f62
0140 627872
0141 627872
0142 646f64
0143 6e2064
0144 6f6261
0145 6f646f
0146 722062
0147 722062
0148 787220
0149 787220
0150
0151 C:\workTemp\LEPREC-1\VISUAL-1\LEA56D-1>Building-Blocks_DUMPER.exe
0152 Building-Blocks_DUMPER rev.1, written by Kaze.
0153 Sorting 16 Pointers to Building-Blocks 3 chars in size ...
0154 Allocated memory for pointers-to-words in MB: 1
0155 Writing Sorted Building-Blocks to BB003.txt ...
0156 3|18-3+1|13|3
0157
0158 C:\workTemp\LEPREC-1\VISUAL-1\LEA56D-1>type BB003.txt
0159 bo
0160 bx
0161 do
0162 an
0163 ban
0164 bob
0165 bxr
0166 dod
0167 n d
0168 oba
0169 odo
0170 r b
0171 xr
0172
0173 C:\workTemp\LEPREC-1\VISUAL-1\LEA56D-1>type osho.txt
0174 */
0175
0176 /*
```

```
0177
0178 D:\_KAZE_new-stuff\Building-Blocks_DUMPER_r1+_Simplicius_Simplicissimus\Intel>Compile_Intel.bat
0179
0180 D:\_KAZE_new-stuff\Building-Blocks_DUMPER_r1+_Simplicius_Simplicissimus\Intel>icl /Ox /TcBuilding-Blocks_DUMPER.c /FaB
0181 uilding-Blocks_DUMPER /w /FACS
0182 Intel(R) C++ Compiler XE for applications running on IA-32, version 12.0.4.196 Build 20110427
0183 Copyright (C) 1985-2011 Intel Corporation. All rights reserved.
0184 30 DAY EVALUATION LICENSE
0185
0186 icl: NOTE: The evaluation period for this product ends on 25-aug-2011 UTC.
0187 Building-Blocks_DUMPER.c
0188 Microsoft (R) Incremental Linker Version 10.00.30319.01
0189 Copyright (C) Microsoft Corporation. All rights reserved.
0190
0191 -out:Building-Blocks_DUMPER.exe
0192 Building-Blocks_DUMPER.obj
0193
0194 D:\_KAZE_new-stuff\Building-Blocks_DUMPER_r1+_Simplicius_Simplicissimus\Intel>Building-Blocks_DUMPER.exe
0195 Building-Blocks_DUMPER rev.1+, written by Kaze.
0196 Note1: This is the precursor of Simplicius_Simplicissimus - a superfast-low-performance TEXT decompressor.
0197 Note2: The niftiness lies in readiness for multi-threading and mostly in boosting the search by having the BBS.
0198 The decompressor would upload (at burst speed) the BB data and then read-and-decode one-by-one the
0199 triads (BB pool/array indexes), that is a simple copying. The pool houses up to 256*256*256 BBS/elements.
0200 Note3: Compiler used: Intel(R) C++ Compiler XE for applications running on IA-32, Version 12.0.4.196 Build 20110427.
0201 Note4: Compile line: icl /Ox /TcBuilding-Blocks_DUMPER.c /FaBuilding-Blocks_DUMPER /w /FACS
0202 Note5: Maximum size of input text file is arbitrary - 384MB, in fact 384MB+*384MB must be less than 19??MB.
0203 Note6: The file format is given by typing the compressed file e.g. D:\>type OSHO.TXT.SS.
0204 The header is 270 bytes long followed by BBS and triads/indexes and the eventual remainder/literals.
0205 Note7: My benchmark text file OSHO.TXT 206,908,949 bytes where OSHO.TXT.SS 116,871,584 bytes:
0206 Decompressing OSHO.TXT.SS to RAM without Dumping to DRIVE time: 2736 clocks, an awful result, grmb1.
0207 Obviously the fastest cache size is crucial, for OSHO.TXT 12MB BB pool vs 1MB L1 cache disbalance is the
0208 cause for this badly inferior performance comparing with LZ cache-friendly variants.
0209 Note8: For more fast decompressing idea(s) you may contact me at sanmayce@sanmayce.com freely.
0210
0211 Usage: Building-Blocks_DUMPER textfilename
0212 Example1: Building-Blocks_DUMPER The_Little_Match_Girl.txt
0213 Example2: Building-Blocks_DUMPER The_Little_Match_Girl.txt.SS
0214 Note: when the extension is not .SS then compression is commenced otherwise decompression.
0215
0216 D:\_KAZE_new-stuff\Building-Blocks_DUMPER_r1+_Simplicius_Simplicissimus\Intel>Building-Blocks_DUMPER.exe OSHO.TXT
0217 Building-Blocks_DUMPER rev.1+, written by Kaze.
0218 Note1: This is the precursor of Simplicius_Simplicissimus - a superfast-low-performance TEXT decompressor.
0219 Note2: The niftiness lies in readiness for multi-threading and mostly in boosting the search by having the BBS.
0220 The decompressor would upload (at burst speed) the BB data and then read-and-decode one-by-one the
0221 triads (BB pool/array indexes), that is a simple copying. The pool houses up to 256*256*256 BBS/elements.
0222 Note3: Compiler used: Intel(R) C++ Compiler XE for applications running on IA-32, Version 12.0.4.196 Build 20110427.
0223 Note4: Compile line: icl /Ox /TcBuilding-Blocks_DUMPER.c /FaBuilding-Blocks_DUMPER /w /FACS
0224 Note5: Maximum size of input text file is arbitrary - 384MB, in fact 384MB+*384MB must be less than 19??MB.
0225 Note6: The file format is given by typing the compressed file e.g. D:\>type OSHO.TXT.SS.
0226 The header is 270 bytes long followed by BBS and triads/indexes and the eventual remainder/literals.
0227 Note7: My benchmark text file OSHO.TXT 206,908,949 bytes where OSHO.TXT.SS 116,871,584 bytes:
0228 Decompressing OSHO.TXT.SS to RAM without Dumping to DRIVE time: 2736 clocks, an awful result, grmb1.
0229 Obviously the fastest cache size is crucial, for OSHO.TXT 12MB BB pool vs 1MB L1 cache disbalance is the
0230 cause for this badly inferior performance comparing with LZ cache-friendly variants.
0231 Note8: For more fast decompressing idea(s) you may contact me at sanmayce@sanmayce.com freely.
0232
0233 Allocating 384MB ... OK
0234
0235 Examining BB orders 3 to 16 whether they are codeable within 3bytes i.e. 16,777,216 ...
0236 Size of incoming file or Strnglen: 206908949
0237
0238 Sorting 206908947 Pointers to building-Blocks 3 chars in size ...
0239 Allocated memory for pointers-to-words in MB: 790
0240 Writing Sorted Building-Blocks to BB003.txt ...
0241 Patternlen:3|206908949-3+1|linecounterRL:46486|206862461
0242 Gain(reduced-size) or Strnglen-(Patternlen*linecounterRL+3*(Strnglen/Patternlen)+(Strnglen%Patternlen)): -139458
0243
0244 Sorting 206908946 Pointers to building-Blocks 4 chars in size ...
0245 Allocated memory for pointers-to-words in MB: 790
0246 Writing Sorted Building-Blocks to BB004.txt ...
0247 Patternlen:4|206908949-4+1|linecounterRL:248019|206660927
0248 Gain(reduced-size) or Strnglen-(Patternlen*linecounterRL+3*(Strnglen/Patternlen)+(Strnglen%Patternlen)): 50735161
0249
0250 Sorting 206908945 Pointers to building-Blocks 5 chars in size ...
0251 Allocated memory for pointers-to-words in MB: 790
0252 Writing Sorted Building-Blocks to BB005.txt ...
0253 Patternlen:5|206908949-5+1|linecounterRL:855682|206053263
0254 Gain(reduced-size) or Strnglen-(Patternlen*linecounterRL+3*(Strnglen/Patternlen)+(Strnglen%Patternlen)): 78485168
0255
0256 Sorting 206908944 Pointers to building-Blocks 6 chars in size ...
0257 Allocated memory for pointers-to-words in MB: 790
0258 Writing Sorted Building-Blocks to BB006.txt ...
0259 Patternlen:6|206908949-6+1|linecounterRL:2236138|204672806
0260 Gain(reduced-size) or Strnglen-(Patternlen*linecounterRL+3*(Strnglen/Patternlen)+(Strnglen%Patternlen)): 90037644
0261
0262 Sorting 206908943 Pointers to building-Blocks 7 chars in size ...
0263 Allocated memory for pointers-to-words in MB: 790
0264 Writing Sorted Building-Blocks to BB007.txt ...
```

```
0265 Patternlen:7|206908949-7+1|linecounterRL:4803152|202105791
0266 Gain(reduced-size) or Strnglen-(Patternlen*linecounterRL+3*(Strnglen/Patternlen)+(Strnglen%Patternlen)): 84611620
0267
0268 Sorting 206908942 Pointers to Building-Blocks 8 chars in size ...
0269 Allocated memory for pointers-to-words in MB: 790
0270 Writing Sorted Building-Blocks to BB008.txt ...
0271 Patternlen:8|206908949-8+1|linecounterRL:8956496|197952446
0272 Gain(reduced-size) or Strnglen-(Patternlen*linecounterRL+3*(Strnglen/Patternlen)+(Strnglen%Patternlen)): 57666122
0273
0274 Sorting 206908941 Pointers to Building-Blocks 9 chars in size ...
0275 Allocated memory for pointers-to-words in MB: 790
0276 Writing Sorted Building-Blocks to BB009.txt ...
0277 Patternlen:9|206908949-9+1|linecounterRL:15006172|191902769
0278 Gain(reduced-size) or Strnglen-(Patternlen*linecounterRL+3*(Strnglen/Patternlen)+(Strnglen%Patternlen)): 2883750
0279
0280 Sorting 206908940 Pointers to Building-Blocks 10 chars in size ...
0281 Allocated memory for pointers-to-words in MB: 790
0282 Writing Sorted Building-Blocks to BB010.txt ...
0283 Patternlen:10|206908949-10+1|linecounterRL:22992127|183916813
0284 Gain(reduced-size) or Strnglen-(Patternlen*linecounterRL+3*(Strnglen/Patternlen)+(Strnglen%Patternlen)): -85085012
0285
0286 Maximum compression (or minimum expansion) for order: 6
0287
0288 Creating OSHO.TXT.SS ...
0289
0290 Building-Blocks_DUMPER .SS dumping time: 42720 clocks
0291 Building-Blocks_DUMPER total time: 1869829 clocks
0292
0293 D:\_KAZE_new-stuff\Building-Blocks_DUMPER_r1+_Simplicius_Simplicissimus\Intel>dir
0294 volume in drive D is s640_vo15
0295 volume Serial Number is F85D-148B
0296
0297 Directory of D:\_KAZE_new-stuff\Building-Blocks_DUMPER_r1+_Simplicius_Simplicissimus\Intel
0298
0299 08/22/2011 01:08 AM <DIR> .
0300 08/22/2011 01:08 AM <DIR> ..
0301 08/22/2011 12:41 AM 139,458 BB003.txt
0302 08/22/2011 12:45 AM 992,076 BB004.txt
0303 08/22/2011 12:49 AM 4,278,410 BB005.txt
0304 08/22/2011 12:52 AM 13,416,828 BB006.txt
0305 08/22/2011 12:56 AM 33,622,064 BB007.txt
0306 08/22/2011 01:00 AM 71,651,968 BB008.txt
0307 08/22/2011 01:04 AM 135,055,548 BB009.txt
0308 08/22/2011 01:08 AM 229,921,270 BB010.txt
0309 08/22/2011 12:33 AM 72,814 Building-Blocks_DUMPER.c
0310 08/22/2011 12:38 AM 917,642 Building-Blocks_DUMPER.cod
0311 08/22/2011 12:38 AM 103,424 Building-Blocks_DUMPER.exe
0312 08/22/2011 12:38 AM 34,431 Building-Blocks_DUMPER.obj
0313 08/22/2011 12:34 AM 72 Compile_Intel.bat
0314 08/21/2011 08:57 AM 206,908,949 OSHO.TXT
0315 08/22/2011 01:09 AM 116,871,584 OSHO.TXT.SS
0316 15 File(s) 813,986,538 bytes
0317 2 Dir(s) 128,145,555,456 bytes free
0318
0319 D:\_KAZE_new-stuff\Building-Blocks_DUMPER_r1+_Simplicius_Simplicissimus\Intel>type OSHO.TXT.SS
0320 [Simplicius_Simplicissimus_revision_1]
0321 The file format after the ASCII code 026:
0322 1 byte for BB_Order,
0323 4bytes for Original_Size,
0324 4bytes for BB length,
0325 BB itself.
0326 3*(Original_Size/BB_Order) bytes of indexes followed by the (Original_Size%BB_Order) bytes of literals.
0327 D:\_KAZE_new-stuff\Building-Blocks_DUMPER_r1+_Simplicius_Simplicissimus\Intel>Building-Blocks_DUMPER.exe OSHO.TXT.SS
0328 Building-Blocks_DUMPER rev.1+, written by Kaze.
0329 Note1: This is the precursor of Simplicius_Simplicissimus - a superfast-low-performance TEXT decompressor.
0330 Note2: The niftiness lies in readiness for multi-threading and mostly in boosting the search by having the BBS.
0331 The decompressor would upload (at burst speed) the BB data and then read-and-decode one-by-one the
0332 triads (BB pool/array indexes), that is a simple copying. The pool houses up to 256*256*256 BBS/elements.
0333 Note3: Compiler used: Intel(R) C++ Compiler XE for applications running on IA-32, Version 12.0.4.196 Build 20110427.
0334 Note4: Compile line: icl /Ox /TcBuilding-Blocks_DUMPER.c /FaBuilding-Blocks_DUMPER /w /FACS
0335 Note5: Maximum size of input text file is arbitrary - 384MB, in fact 384MB+*384MB must be less than 19??MB.
0336 Note6: The file format is given by typing the compressed file e.g. D:\>type OSHO.TXT.SS.
0337 The header is 270 bytes long followed by BBS and triads/indexes and the eventual remainder/literals.
0338 Note7: My benchmark text file OSHO.TXT 206,908,949 bytes where OSHO.TXT.SS 116,871,584 bytes:
0339 Decompressing OSHO.TXT.SS to RAM without Dumping to DRIVE time: 2736 clocks, an awful result, grmb1.
0340 Obviously the fastest cache size is crucial, for OSHO.TXT 12MB BB pool vs 1MB L1 cache disbalance is the
0341 cause for this badly inferior performance comparing with LZ cache-friendly variants.
0342 Note8: For more fast decompressing idea(s) you may contact me at sanmayce@sanmayce.com freely.
0343
0344 Allocating 384MB ... OK
0345
0346 Simplicius_Simplicissimus is decompressing OSHO.TXT ...
0347 Allocating 256MB for BBS pool + 384MB for RAM-to-RAM decompression ... OK
0348 BBS order: 6
0349 Original file size: 206908949
0350 BBS pool size: 13416828
0351
0352 Decompression to RAM without Dumping to DRIVE time: 2876 clocks
```

```

0353 Decompression to RAM without Dumping to DRIVE performance: 70257 KB/s
0354 Total time: 6595 clocks
0355
0356 D:\_KAZE_new-stuff\Building-Blocks_DUMPER_r1+_Simplicius_Simplicissimus\Intel>Building-Blocks_DUMPER.exe OSHO.TXT.SS
0357 Building-Blocks_DUMPER rev.1+, written by Kaze.
0358 Note1: This is the precursor of Simplicius_Simplicissimus - a superfast-low-performance TEXT decompressor.
0359 Note2: The niftiness lies in readiness for multi-threading and mostly in boosting the search by having the BBS.
0360 The decompressor would upload (at burst speed) the BB data and then read-and-decode one-by-one the
0361 triads (BB pool/array indexes), that is a simple copying. The pool houses up to 256*256*256 BBS/elements.
0362 Note3: Compiler used: Intel(R) C++ Compiler XE for applications running on IA-32, Version 12.0.4.196 Build 20110427.
0363 Note4: Compile line: icl /ox /TcBuilding-Blocks_DUMPER.c /FaBuilding-Blocks_DUMPER /w /FACS
0364 Note5: Maximum size of input text file is arbitrary - 384MB, in fact 384MB+4*384MB must be less than 19??MB.
0365 Note6: The file format is given by typing the compressed file e.g. D:\>type OSHO.TXT.SS.
0366 The header is 270 bytes long followed by BBS and triads/indexes and the eventual remainder/literals.
0367 Note7: My benchmark text file OSHO.TXT 206,908,949 bytes where OSHO.TXT.SS 116,871,584 bytes:
0368 Decompressing OSHO.TXT.SS to RAM without Dumping to DRIVE time: 2736 clocks, an awful result, grmb1.
0369 Obviously the fastest cache size is crucial, for OSHO.TXT 12MB BB pool vs 1MB L1 cache disbalance is the
0370 cause for this badly inferior performance comparing with LZ cache-friendly variants.
0371 Note8: For more fast decompressing idea(s) you may contact me at sanmayce@sanmayce.com freely.
0372
0373 Allocating 384MB ... OK
0374
0375 Simplicius_Simplicissimus is decompressing OSHO.TXT ...
0376 Allocating 256MB for BBS pool + 384MB for RAM-to-RAM decompression ... OK
0377 BBS order: 6
0378 Original file size: 206908949
0379 BBS pool size: 13416828
0380
0381 Decompression to RAM without Dumping to DRIVE time: 2767 clocks
0382 Decompression to RAM without Dumping to DRIVE performance: 73024 KB/s
0383 Total time: 5251 clocks
0384
0385 D:\_KAZE_new-stuff\Building-Blocks_DUMPER_r1+_Simplicius_Simplicissimus\Intel>Building-Blocks_DUMPER.exe OSHO.TXT.SS
0386 Building-Blocks_DUMPER rev.1+, written by Kaze.
0387 Note1: This is the precursor of Simplicius_Simplicissimus - a superfast-low-performance TEXT decompressor.
0388 Note2: The niftiness lies in readiness for multi-threading and mostly in boosting the search by having the BBS.
0389 The decompressor would upload (at burst speed) the BB data and then read-and-decode one-by-one the
0390 triads (BB pool/array indexes), that is a simple copying. The pool houses up to 256*256*256 BBS/elements.
0391 Note3: Compiler used: Intel(R) C++ Compiler XE for applications running on IA-32, Version 12.0.4.196 Build 20110427.
0392 Note4: Compile line: icl /ox /TcBuilding-Blocks_DUMPER.c /FaBuilding-Blocks_DUMPER /w /FACS
0393 Note5: Maximum size of input text file is arbitrary - 384MB, in fact 384MB+4*384MB must be less than 19??MB.
0394 Note6: The file format is given by typing the compressed file e.g. D:\>type OSHO.TXT.SS.
0395 The header is 270 bytes long followed by BBS and triads/indexes and the eventual remainder/literals.
0396 Note7: My benchmark text file OSHO.TXT 206,908,949 bytes where OSHO.TXT.SS 116,871,584 bytes:
0397 Decompressing OSHO.TXT.SS to RAM without Dumping to DRIVE time: 2736 clocks, an awful result, grmb1.
0398 Obviously the fastest cache size is crucial, for OSHO.TXT 12MB BB pool vs 1MB L1 cache disbalance is the
0399 cause for this badly inferior performance comparing with LZ cache-friendly variants.
0400 Note8: For more fast decompressing idea(s) you may contact me at sanmayce@sanmayce.com freely.
0401
0402 Allocating 384MB ... OK
0403
0404 Simplicius_Simplicissimus is decompressing OSHO.TXT ...
0405 Allocating 256MB for BBS pool + 384MB for RAM-to-RAM decompression ... OK
0406 BBS order: 6
0407 Original file size: 206908949
0408 BBS pool size: 13416828
0409
0410 Decompression to RAM without Dumping to DRIVE time: 2861 clocks
0411 Decompression to RAM without Dumping to DRIVE performance: 70625 KB/s
0412 Total time: 5517 clocks
0413
0414 D:\_KAZE_new-stuff\Building-Blocks_DUMPER_r1+_Simplicius_Simplicissimus\Intel>
0415 */
0416
0417 #define DumboDUMP
0418 #define Quicks
0419
0420 #include <stdio.h>
0421 #include <stdlib.h>
0422 #include <string.h>
0423 #include <time.h>
0424
0425 #ifndef NULL
0426 #define NULL ((void*)0)
0427 #endif
0428
0429 typedef unsigned char char_t;
0430 typedef char_t *string;
0431
0432 int PatternLen;
0433
0434 clock_t clocks1, clocks2;
0435 clock_t clocks3, clocks4;
0436 double TotalRoughSearchTime = 0;
0437 double TotalRoughSearchTime2 = 0;
0438
0439 // SINHA fragment[
0440

```

```

0441 #define swapKAZE(a, b) { t = *(a); *(a) = *(b); *(b) = t; }
0442
0443 static void InsertSortKAZE(string *a, int n, int d) //void insort(unsigned char **a, int n, int d)
0444 { string *pi, *pj, s, t; //unsigned char **pi, **pj, *s, *t;
0445 for (pi = a + 1; --n > 0; pi++)
0446 for (pj = pi; pj > a; pj--) {
0447 /* Inline strcmp: break if *(pj-1) <= *pj */
0448 for (s=(pj-1)+d, t=*pj+d; *s==*t && *s!=0; s++, t++)
0449 ;
0450 if (*s <= *t)
0451 break;
0452 swapKAZE(pj, pi-1);
0453 }
0454 }
0455
0456 //int cmpit(unsigned char **h1, unsigned char **h2)
0457 //{
0458 // return( strcmp(*h1, *h2) );
0459 //}
0460
0461 int scmp( unsigned char *s1, unsigned char *s2 )
0462 {
0463 while( *s1 != '\0' && *s1 == *s2 )
0464 {
0465 s1++;
0466 s2++;
0467 }
0468 return( *s1-*s2 );
0469 }
0470
0471 static void simplesort(string a[], int n, int b)
0472 {
0473 int i, j;
0474 string tmp;
0475
0476 for (i = 1; i < n; i++)
0477 for (j = i; j > 0 && scmp(a[j-1]+b, a[j]+b) > 0; j--)
0478 { tmp = a[j]; a[j] = a[j-1]; a[j-1] = tmp; }
0479 }
0480
0481 int memcmpKAZE (
0482 const void * buf1,
0483 const void * buf2,
0484 size_t count
0485 )
0486 {
0487 if (!count)
0488 return(0);
0489
0490 while ( --count && *(char *)buf1 == *(char *)buf2 ) {
0491 buf1 = (char *)buf1 + 1;
0492 buf2 = (char *)buf2 + 1;
0493 }
0494
0495 return( *((unsigned char *)buf1) - *((unsigned char *)buf2) );
0496 }
0497
0498 void * memcpyKAZE (
0499 void * dst,
0500 const void * src,
0501 size_t count
0502 )
0503 {
0504 void * ret = dst;
0505
0506 /*
0507 * copy from lower addresses to higher addresses
0508 */
0509 while (count--) {
0510 *(char *)dst = *(char *)src;
0511 dst = (char *)dst + 1;
0512 src = (char *)src + 1;
0513 }
0514 return(ret);
0515 }
0516
0517 static void simplesortFIXEDLength(string a[], int n, int b, int FIXEDLength)
0518 {
0519 int i, j;
0520 string tmp;
0521
0522 for (i = 1; i < n; i++)
0523 for (j = i; j > 0 && memcmpKAZE(a[j-1]+b, a[j]+b, FIXEDLength) > 0; j--)
0524 { tmp = a[j]; a[j] = a[j-1]; a[j-1] = tmp; }
0525 }
0526
0527
0528 // SINHA fragment[

```

```

0529
0530 // mkqsort.c BEGIN *****
0531 /*
0532 Multikey quicksort, a radix sort algorithm for arrays of character
0533 strings by Bentley and Sedgewick.
0534
0535 J. Bentley and R. Sedgewick. Fast algorithms for sorting and
0536 searching strings. In Proceedings of 8th Annual ACM-SIAM Symposium
0537 on Discrete Algorithms, 1997.
0538
0539 http://www.CS.Princeton.EDU/~rs/strings/index.html
0540
0541 The code presented in this file has been tested with care but is
0542 not guaranteed for any purpose. The writer does not offer any
0543 warranties nor does he accept any liabilities with respect to
0544 the code.
0545
0546 Ranjan Sinha, 1 jan 2003.
0547
0548 School of Computer Science and Information Technology,
0549 RMIT University, Melbourne, Australia
0550 rsinha@cs.rmit.edu.au
0551
0552 */
0553
0554 #include "sortstring.h"
0555
0556 /* MULTIKEY QUICKSORT */
0557
0558 #ifndef min
0559 #define min(a, b) ((a)<=(b) ? (a) : (b))
0560 #endif
0561
0562 /* ssort2 -- Faster Version of Multikey Quicksort */
0563
0564 void vecswap(unsigned char **a, unsigned char **b, int n)
0565 { while (n-- > 0) {
0566     unsigned char *t = *a;
0567     *a++ = *b;
0568     *b++ = t;
0569 } }
0570
0571 #define swap2(a, b) { t = *(a); *(a) = *(b); *(b) = t; }
0572 #define ptr2char(i) (*(i) + depth)
0573
0574 unsigned char **med3func(unsigned char **a, unsigned char **b, unsigned char **c, int depth)
0575 { int va, vb, vc;
0576   if ((va=ptr2char(a)) == (vb=ptr2char(b)))
0577     return a;
0578   if ((vc=ptr2char(c)) == va || vc == vb)
0579     return c;
0580   return va < vb ?
0581     (vb < vc ? b : (va < vc ? c : a)) :
0582     (vb > vc ? b : (va < vc ? a : c));
0583 }
0584 #define med3(a, b, c) med3func(a, b, c, depth)
0585
0586 void insort(unsigned char **a, int n, int d)
0587 { unsigned char **pi, **pj; *s, *t;
0588   for (pi = a + 1; --n > 0; pi++)
0589     for (pj = pi; pj > a; pj--) {
0590       /* Inline strcmp: break if *(pj-1) <= *pj */
0591       for (s=*(pj-1)+d, t=*pj+d; *s==*t && *s!=0; s++, t++)
0592         ;
0593       if (*s <= *t)
0594         break;
0595       swap2(pj, pj-1);
0596     }
0597 }
0598
0599 void mkqsort(unsigned char **a, int n, int depth)
0600 { int d, r, partval;
0601   unsigned char **pa, **pb, **pc, **pd, **pl, **pm, **pn, *t;
0602
0603   millions of equal 7bytes chunks: shitty recursive approach !!!!!!!!!!!
0604
0605   /*
0606   The next dpeths are for file hs_ref_GRch37.p5_chrX.fa:
0607   ...
0608   depth:8
0609   depth:9
0610   depth:9
0611   depth:10
0612   depth:10
0613   depth:11
0614   depth:10
0615   depth:9

```

```

0616 depth:10
0617 depth:10
0618 depth:11
0619 depth:11
0620 depth:12
0621 depth:10
0622 depth:10
0623 depth:11
0624 depth:10
0625 depth:9
0626 depth:10
0627 depth:10
0628 depth:10
0629 depth:11
0630 depth:10
0631 depth:11
0632 depth:11
0633 depth:11
0634 depth:11
0635 depth:12
0636 depth:12
0637 depth:12
0638 depth:12
0639 depth:13
0640 depth:13
0641 depth:14
0642 depth:14
0643 depth:15
0644 depth:15
0645 depth:16
0646 depth:16
0647 depth:17
0648 depth:1
0649 depth:1
0650 depth:2
0651 depth:3
0652 depth:4
0653 depth:5
0654 depth:6
0655 depth:7
0656 depth:8
0657 depth:9
0658 depth:10
0659 depth:11
0660 depth:12
0661 depth:13
0662 depth:14
0663 depth:15
0664 depth:16
0665 depth:17
0666 depth:18
0667 depth:19
0668 depth:20
0669 depth:21
0670 depth:22
0671 depth:23
0672 depth:24
0673 depth:25
0674 depth:26
0675 depth:27
0676 depth:28
0677 depth:29
0678 depth:30
0679 depth:31
0680 depth:32
0681 depth:33
0682 depth:34
0683 depth:35
0684 depth:36
0685 depth:37
0686 depth:38
0687 depth:39
0688 depth:40
0689 ...
0690 depth:903
0691 depth:904
0692 depth:905
0693 depth:906
0694 depth:907
0695 depth:907
0696 ...
0697 depth:2572
0698 depth:2573
0699 depth:2574
0700 depth:2575
0701 ...
0702 depth:3380
0703 depth:3381

```

```

0704 depth:3382
0705 depth:3383
0706 depth:3470
0707 depth:3471
0708 depth:3472
0709 depth:3473
0710 ...
0711 depth:21310
0712 depth:21311
0713 depth:21312
0714 ...
0715 depth:21416
0716 depth:21417
0717 depth:21418
0718
0719 and self-exit at this depth! That is the program terminates itself as if no error occurred!
0720 */
0721
0722 if (n < 20) {
0723     //inssort(a, n, depth);
0724     simplesortFIXEDLength(a, n, depth, Patternlen);
0725     return;
0726 }
0727 p1 = a;
0728 pm = a + (n/2);
0729 pn = a + (n-1);
0730 if (n > 30) { /* On big arrays, pseudomedian of 9 */
0731     d = (n/8);
0732     p1 = med3(p1, p1+d, p1+2*d);
0733     pm = med3(pm-d, pm, pm+d);
0734     pn = med3(pn-2*d, pn-d, pn);
0735 }
0736 pm = med3(p1, pm, pn);
0737 swap2(a, pm);
0738 partval = ptr2char(a);
0739 pa = pb = a + 1;
0740 pc = pd = a + n-1;
0741 for (;;) {
0742     while (pb <= pc && (r = ptr2char(pb)-partval) <= 0) {
0743         if (r == 0) { swap2(pa, pb); pa++; }
0744         pb++;
0745     }
0746     while (pb <= pc && (r = ptr2char(pc)-partval) >= 0) {
0747         if (r == 0) { swap2(pc, pd); pd--; }
0748         pc--;
0749     }
0750     if (pb > pc) break;
0751     swap2(pb, pc);
0752     pb++;
0753     pc--;
0754 }
0755 pn = a + n;
0756 r = min(pa-a, pb-pa); vecswap2(a, pb-r, r);
0757 r = min(pd-pc, pn-pd-1); vecswap2(pb, pn-r, r);
0758 if ((r = pb-pa) > 1)
0759     mkqsort(a, r, depth);
0760 if (ptr2char(a+r) != 0)
0761     mkqsort(a+r, pa-a+pn-pd-1, depth+1);
0762 if ((r = pd-pc) > 1)
0763     mkqsort(a+n-r, r, depth);
0764 }
0765
0766 void mkqsort_main(unsigned char **a, int n) { mkqsort(a, n, 0); }
0767 // mkqsort.c END *****
0768
0769 // why Sinha uses int instead of long?!!
0770 static int readlines(char *file_name, string **lines)
0771 {
0772     int nlines = 0;
0773     size_t size;
0774     FILE *in_file;
0775     string basep, cur, next;
0776     string *ASbackup;
0777
0778     if (!(in_file = fopen(file_name, "rb"))) {
0779         printf("Leprechaun: Can't open file %s \n", file_name);
0780         exit(-1);
0781     }
0782     fseek(in_file, 0, SEEK_END);
0783     size = ftell(in_file);
0784     fseek(in_file, 0, SEEK_SET);
0785     if (!(basep = (string) malloc(size*sizeof(char_t)))) return -1;
0786     printf("Allocated memory for words in MB: %lu\n", ((size*sizeof(char_t))>>20)+1);
0787     if (fread(basep, 1, size, in_file) < size) {
0788         printf("Leprechaun: Can't read file %s \n", file_name);
0789         exit(-1);
0790     }
0791     fclose(in_file);

```

```

0792
0793 // GET nlines:
0794 cur = basep;
0795 while (cur < basep + size) {
0796     next = cur;
0797     while ((next < basep + size) && (*next != '\n')) {next++;}
0798     *--next = '\0'; // This is ala DOS i.e. Windows
0799     // 1310 not 10(\n=10)
0800     cur = next + 2;
0801     nlines++;
0802 }
0803
0804 // printf("%lu\n", (unsigned long)*lines); -> backup = *lines = 0
0805 ASbackup = (string *)malloc(nlines*sizeof(string)); // sizeof(string) is 4
0806 if( ASbackup == NULL )
0807 { puts("Leprechaun: Needed memory allocation denied!\n"); return( 1 ); }
0808 printf("Allocated memory for pointers-to-words in MB: %lu\n", ((nlines*sizeof(string))>>20)+1 );
0809 *lines = ASbackup;
0810 //printf("%lu\n", (unsigned long)*lines); -> backup = *lines = ASbackup = 6946888
0811
0812 // Upload nlines times:
0813 nlines = 0;
0814 cur = basep;
0815 while (cur < basep + size) {
0816     next = cur;
0817     while ((next < basep + size) && (*next != '\n')) {next++;}
0818     *--next = '\0'; // This is ala DOS i.e. Windows
0819     // 1310 not 10(\n=10)
0820     ASbackup[nlines] = cur;
0821     cur = next + 2;
0822     nlines++;
0823 }
0824 return nlines;
0825 }
0826
0827 void x64toKAZE ( /* stdcall is faster and smaller... Might as well use it for the helper. */
0828     unsigned long long val,
0829     char *buf,
0830     unsigned radix,
0831     int is_neg
0832 )
0833 {
0834     char *p; /* pointer to traverse string */
0835     char *firstdig; /* pointer to first digit */
0836     char temp; /* temp char */
0837     unsigned digval; /* value of digit */
0838
0839     p = buf;
0840
0841     if ( is_neg )
0842     {
0843         *p++ = '-'; /* negative, so output '-' and negate */
0844         val = (unsigned long long)(-(long long)val);
0845     }
0846
0847     firstdig = p; /* save pointer to first digit */
0848
0849     do {
0850         digval = (unsigned) (val % radix);
0851         val /= radix; /* get next digit */
0852
0853         /* convert to ascii and store */
0854         if (digval > 9)
0855             *p++ = (char) (digval - 10 + 'a'); /* a letter */
0856         else
0857             *p++ = (char) (digval + '0'); /* a digit */
0858     } while (val > 0);
0859
0860     /* we now have the digit of the number in the buffer, but in reverse
0861     order. Thus we reverse them now. */
0862
0863     *p-- = '\0'; /* terminate string; p points to last digit */
0864
0865     do {
0866         temp = *p;
0867         *p = *firstdig;
0868         *firstdig = temp; /* swap *p and *firstdig */
0869         --p;
0870         ++firstdig; /* advance to next two digits */
0871     } while (firstdig < p); /* repeat until halfway */
0872 }
0873
0874 /* Actual functions just call conversion helper with neg flag set correctly,
0875 and return pointer to buffer. */
0876
0877 char * _i64toKAZE (
0878     long long val,
0879     char *buf,

```

```

0880     int radix
0881     )
0882 {
0883     x64toaKAZE((unsigned long long)val, buf, radix, (radix == 10 && val < 0));
0884     return buf;
0885 }
0886
0887 char * _ui64toaKAZE (
0888     unsigned long long val,
0889     char *buf,
0890     int radix
0891     )
0892 {
0893     x64toaKAZE(val, buf, radix, 0);
0894     return buf;
0895 }
0896
0897 char * _ui64toaKAZEzerocomma (
0898     unsigned long long val,
0899     char *buf,
0900     int radix
0901     )
0902 {
0903     char *p;
0904     char temp;
0905     int txpman;
0906     int pxnman;
0907     x64toaKAZE(val, buf, radix, 0);
0908     p = buf;
0909     do {
0910         } while (*++p != '\0');
0911     p--; // p points to last digit
0912         // buf points to first digit
0913     buf[26] = 0;
0914     txpman = 1;
0915     pxnman = 0;
0916     do
0917     { if (buf <= p)
0918       { temp = *p;
0919         buf[26-txpman] = temp; pxnman++;
0920         p--;
0921         if (pxnman % 3 == 0)
0922         { txpman++;
0923           buf[26-txpman] = (char) ('.','.');
0924         }
0925         }
0926     else
0927     { buf[26-txpman] = (char) ('0'); pxnman++;
0928       if (pxnman % 3 == 0)
0929       { txpman++;
0930         buf[26-txpman] = (char) ('.','.');
0931       }
0932     }
0933     txpman++;
0934     } while (txpman <= 26);
0935     return buf;
0936 }
0937
0938 char * _ui64toaKAZEcomma (
0939     unsigned long long val,
0940     char *buf,
0941     int radix
0942     )
0943 {
0944     char *p;
0945     char temp;
0946     int txpman;
0947     int pxnman;
0948     x64toaKAZE(val, buf, radix, 0);
0949     p = buf;
0950     do {
0951         } while (*++p != '\0');
0952     p--; // p points to last digit
0953         // buf points to first digit
0954     buf[26] = 0;
0955     txpman = 1;
0956     pxnman = 0;
0957     while (buf <= p)
0958     { temp = *p;
0959       buf[26-txpman] = temp; pxnman++;
0960       p--;
0961       if (pxnman % 3 == 0 && buf <= p)
0962       { txpman++;
0963         buf[26-txpman] = (char) ('.','.');
0964       }
0965     }
0966     txpman++;
0967     return buf+26-(txpman-1);

```

```

0968 }
0969
0970
0971
0972 int compvs2003 (
0973     const void * buf1,
0974     const void * buf2
0975     )
0976 {
0977     unsigned long * buf1Nested = buf1;
0978     unsigned long * buf2Nested = buf2;
0979     void * buf1Fetched;
0980     void * buf2Fetched;
0981
0982     buf1Fetched = *buf1Nested;
0983     buf2Fetched = *buf2Nested;
0984
0985     return( memcmpKAZE ( buf1Fetched, buf2Fetched, Patternlen ) );
0986 }
0987 }
0988
0989
0990
0991 // Qsortvs2003 [ =====
0992
0993
0994
0995 /****
0996 *swap(a, b, width) - swap two elements
0997 *
0998 *Purpose:
0999 *   swaps the two array elements of size width
1000 *
1001 *Entry:
1002 *   char *a, *b = pointer to two elements to swap
1003 *   size_t width = width in bytes of each array element
1004 *
1005 *Exit:
1006 *   returns void
1007 *
1008 *Exceptions:
1009 *
1010 *****/
1011
1012 static void swapvs2003 (
1013     char *a,
1014     char *b,
1015     size_t width
1016     )
1017 {
1018     char tmp;
1019
1020     if ( a != b )
1021     /* Do the swap one character at a time to avoid potential alignment
1022        problems. */
1023     while ( width-- ) {
1024         tmp = *a;
1025         *a++ = *b;
1026         *b++ = tmp;
1027     }
1028 }
1029
1030
1031
1032
1033 /****
1034 *shortsort(hi, lo, width, comp) - insertion sort for sorting short arrays
1035 *
1036 *Purpose:
1037 *   sorts the sub-array of elements between lo and hi (inclusive)
1038 *   side effects: sorts in place
1039 *   assumes that lo < hi
1040 *
1041 *Entry:
1042 *   char *lo = pointer to low element to sort
1043 *   char *hi = pointer to high element to sort
1044 *   size_t width = width in bytes of each array element
1045 *   int (*comp)() = pointer to function returning analog of strcmp for
1046 *                   strings, but supplied by user for comparing the array elements.
1047 *                   it accepts 2 pointers to elements and returns neg if l<2, 0 if
1048 *                   l=2, pos if l>2.
1049 *
1050 *Exit:
1051 *   returns void
1052 *
1053 *Exceptions:
1054 *
1055 *****/

```

```

1056
1057 static void shortsrtvs2003 (
1058     char *lo,
1059     char *hi,
1060     size_t width,
1061     int (*compvs2003)(const void *, const void *)
1062 )
1063 {
1064     char *p, *max;
1065
1066     /* Note: in assertions below, i and j are always inside original bound of
1067     array to sort. */
1068
1069     while (hi > lo) {
1070         /* A[i] <= A[j] for i <= j, j > hi */
1071         max = lo;
1072         for (p = lo+width; p <= hi; p += width) {
1073             /* A[i] <= A[max] for lo <= i < p */
1074             if (compvs2003(p, max) > 0) {
1075                 max = p;
1076             }
1077             /* A[i] <= A[max] for lo <= i <= p */
1078         }
1079
1080         /* A[i] <= A[max] for lo <= i <= hi */
1081         swapvs2003(max, hi, width);
1082
1083         /* A[i] <= A[hi] for i <= hi, so A[i] <= A[j] for i <= j, j >= hi */
1084         hi -= width;
1085
1086         /* A[i] <= A[j] for i <= j, j > hi, loop top condition established */
1087     }
1088     /* A[i] <= A[j] for i <= j, j > lo, which implies A[i] <= A[j] for i < j,
1089     so array is sorted */
1090 }
1091
1092
1093
1094
1095
1096
1097 /* this parameter defines the cutoff between using quick sort and
1098 insertion sort for arrays; arrays with lengths shorter or equal to the
1099 below value use insertion sort */
1100
1101 #define CUTOFF 8 /* testing shows that this is good value */
1102
1103 /**
1104  *qsort(base, num, wid, comp) - quicksort function for sorting arrays
1105  *
1106  *Purpose:
1107  * quicksort the array of elements
1108  * side effects: sorts in place
1109  * maximum array size is number of elements times size of elements,
1110  * but is limited by the virtual address space of the processor
1111  *
1112  *Entry:
1113  * char *base = pointer to base of array
1114  * size_t num = number of elements in the array
1115  * size_t width = width in bytes of each array element
1116  * int (*comp)() = pointer to function returning analog of strcmp for
1117  * strings, but supplied by user for comparing the array elements.
1118  * it accepts 2 pointers to elements and returns neg if l<2, 0 if
1119  * l=2, pos if l>2.
1120  *
1121  *Exit:
1122  * returns void
1123  *
1124  *Exceptions:
1125  *
1126  ****
1127
1128 /* sort the array between lo and hi (inclusive) */
1129
1130 #define STKSIZ (8*sizeof(void*) - 2)
1131
1132 void qsortvs2003 (
1133     void *base,
1134     size_t num,
1135     size_t width,
1136     int (*compvs2003)(const void *, const void *)
1137 )
1138 {
1139     /* Note: the number of stack entries required is no more than
1140     1 + log2(num), so 30 is sufficient for any array */
1141     char *lo, *hi; /* ends of sub-array currently sorting */
1142     char *mid; /* points to middle of subarray */
1143     char *loguy, *higuy; /* traveling pointers for partition step */

```

```

1144     size_t size; /* size of the sub-array */
1145     char *lostk[STKSIZ], *histk[STKSIZ];
1146     int stkptr; /* stack for saving sub-array to be processed */
1147
1148     if (num < 2 || width == 0)
1149         return; /* nothing to do */
1150
1151     stkptr = 0; /* initialize stack */
1152
1153     lo = base;
1154     hi = (char *)base + width * (num-1); /* initialize limits */
1155
1156     /* this entry point is for pseudo-recursion calling: setting
1157     lo and hi and jumping to here is like recursion, but stkptr is
1158     preserved, locals aren't, so we preserve stuff on the stack */
1159     recurse:
1160
1161     size = (hi - lo) / width + 1; /* number of el's to sort */
1162
1163     /* below a certain size, it is faster to use a O(n^2) sorting method */
1164     if (size <= CUTOFF) {
1165         shortsrtvs2003(lo, hi, width, compvs2003);
1166     }
1167     else {
1168         /* First we pick a partitioning element. The efficiency of the
1169         algorithm demands that we find one that is approximately the median
1170         of the values, but also that we select one fast. We choose the
1171         median of the first, middle, and last elements, to avoid bad
1172         performance in the face of already sorted data, or data that is made
1173         up of multiple sorted runs appended together. Testing shows that a
1174         median-of-three algorithm provides better performance than simply
1175         picking the middle element for the latter case. */
1176
1177         mid = lo + (size / 2) * width; /* find middle element */
1178
1179         /* Sort the first, middle, last elements into order */
1180         if (compvs2003(lo, mid) > 0) {
1181             swapvs2003(lo, mid, width);
1182         }
1183         if (compvs2003(lo, hi) > 0) {
1184             swapvs2003(lo, hi, width);
1185         }
1186         if (compvs2003(mid, hi) > 0) {
1187             swapvs2003(mid, hi, width);
1188         }
1189
1190         /* We now wish to partition the array into three pieces, one consisting
1191         of elements <= partition element, one of elements equal to the
1192         partition element, and one of elements > than it. This is done
1193         below; comments indicate conditions established at every step. */
1194
1195         loguy = lo;
1196         higuy = hi;
1197
1198         /* Note that higuy decreases and loguy increases on every iteration,
1199         so loop must terminate. */
1200         for (;;) {
1201             /* lo <= loguy < hi, lo < higuy <= hi,
1202             A[i] <= A[mid] for lo <= i <= loguy,
1203             A[i] > A[mid] for higuy <= i < hi,
1204             A[hi] >= A[mid] */
1205
1206             /* The doubled loop is to avoid calling comp(mid,mid), since some
1207             existing comparison funcs don't work when passed the same
1208             value for both pointers. */
1209
1210             if (mid > loguy) {
1211                 do {
1212                     loguy += width;
1213                 } while (loguy < mid && compvs2003(loguy, mid) <= 0);
1214             }
1215             if (mid <= loguy) {
1216                 do {
1217                     loguy += width;
1218                 } while (loguy <= hi && compvs2003(loguy, mid) <= 0);
1219             }
1220
1221             /* lo < loguy <= hi+1, A[i] <= A[mid] for lo <= i < loguy,
1222             either loguy > hi or A[loguy] > A[mid] */
1223
1224             do {
1225                 higuy -= width;
1226             } while (higuy > mid && compvs2003(higuy, mid) > 0);
1227
1228             /* lo <= higuy < hi, A[i] > A[mid] for higuy < i < hi,
1229             either higuy == lo or A[higuy] <= A[mid] */
1230
1231             if (higuy < loguy)

```

```

1232     break;
1233
1234     /* if loguy > hi or higuy == lo, then we would have exited, so
1235     A[loguy] > A[mid], A[higuy] <= A[mid],
1236     loguy <= hi, higuy > lo */
1237
1238     swapvs2003(loguy, higuy, width);
1239
1240     /* If the partition element was moved, follow it. Only need
1241     to check for mid == higuy, since before the swap,
1242     A[loguy] > A[mid] implies loguy != mid. */
1243
1244     if (mid == higuy)
1245         mid = loguy;
1246
1247     /* A[loguy] <= A[mid], A[higuy] > A[mid]; so condition at top
1248     of loop is re-established */
1249 }
1250
1251 /* A[i] <= A[mid] for lo <= i < loguy,
1252 A[i] > A[mid] for higuy < i < hi,
1253 A[hi] >= A[mid]
1254 higuy < loguy
1255 implying:
1256 higuy == loguy-1
1257 or higuy == hi - 1, loguy == hi + 1, A[hi] == A[mid] */
1258
1259 /* Find adjacent elements equal to the partition element. The
1260 doubled loop is to avoid calling comp(mid,mid), since some
1261 existing comparison funcs don't work when passed the same value
1262 for both pointers. */
1263
1264 higuy += width;
1265 if (mid < higuy) {
1266     do {
1267         higuy -= width;
1268     } while (higuy > mid && compvs2003(higuy, mid) == 0);
1269 }
1270 if (mid >= higuy) {
1271     do {
1272         higuy -= width;
1273     } while (higuy > lo && compvs2003(higuy, mid) == 0);
1274 }
1275
1276 /* OK, now we have the following:
1277     loguy < loguy
1278     lo <= higuy <= hi
1279     A[i] <= A[mid] for lo <= i <= higuy
1280     A[i] == A[mid] for higuy < i < loguy
1281     A[i] > A[mid] for loguy <= i < hi
1282     A[hi] >= A[mid] */
1283
1284 /* We've finished the partition, now we want to sort the subarrays
1285 [lo, higuy] and [loguy, hi].
1286 We do the smaller one first to minimize stack usage.
1287 We only sort arrays of length 2 or more.*/
1288
1289 if ( (higuy - lo >= hi - loguy) ) {
1290     if (lo < higuy) {
1291         lostk[stkptr] = lo;
1292         histk[stkptr] = higuy;
1293         ++stkptr;
1294     }
1295     /* save big recursion for later */
1296
1297     if (loguy < hi) {
1298         lo = loguy;
1299         goto recurse; /* do small recursion */
1300     }
1301 } else {
1302     if (loguy < hi) {
1303         lostk[stkptr] = loguy;
1304         histk[stkptr] = hi;
1305         ++stkptr;
1306     }
1307     /* save big recursion for later */
1308
1309     if (lo < higuy) {
1310         hi = higuy;
1311         goto recurse; /* do small recursion */
1312     }
1313 }
1314
1315 /* We have sorted the array, except for any pending sorts on the stack.
1316 Check if there are any, and do them. */
1317
1318 --stkptr;
1319 if (stkptr >= 0) {

```

```

1320     lo = lostk[stkptr];
1321     hi = histk[stkptr];
1322     goto recurse; /* pop subarray from stack */
1323 }
1324 else
1325     return; /* all subarrays done */
1326 }
1327
1328
1329
1330 // Qsortvs2003 ] =====
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352 long Railgunhits=0;
1353
1354 #define ASIZE 256
1355
1356 // ## Boyer-Moore-Horspool algorithm [
1357 long HORSPOOL(y, x, n, m)
1358     char *y, *x;
1359     long n;
1360     int m;
1361     {
1362     long i;
1363     int a, j, bm_bc[ASIZE];
1364     unsigned char ch, lastch;
1365
1366     /* Preprocessing */
1367     for (a=0; a < ASIZE; a++) bm_bc[a]=m;
1368     for (j=0; j < m-1; j++) bm_bc[x[j]]=m-j-1;
1369
1370     /* Searching */
1371     lastch=x[m-1];
1372     i=0;
1373     while (i <= n-m) {
1374         ch=y[i+m-1];
1375         if (ch ==lastch)
1376             //if (memcmp(&y[i],x,m-1) == 0) OUTPUT(i);
1377             if (memcmp(&y[i],x,m-1) == 0) return(i);
1378             i+=bm_bc[ch];
1379     }
1380     return(-1);
1381     }
1382 long Boyer_Moore_Horspool_Kaze(y, x, n, m)
1383     char *y, *x;
1384     long n;
1385     int m;
1386     {
1387     long i;
1388     int a, j, bm_bc[ASIZE];
1389     unsigned char ch;
1390     //unsigned char ch, lastch;
1391     //unsigned char firstch;
1392
1393     /* Preprocessing */
1394     for (a=0; a < ASIZE; a++) bm_bc[a]=m;
1395     for (j=0; j < m-1; j++) bm_bc[x[j]]=m-j-1;
1396
1397     /* Searching */
1398     //lastch=x[m-1];
1399     //firstch=x[0];
1400     i=0;
1401     while (i <= n-m) {
1402         ch=y[i+m-1];
1403         //if (ch ==lastch)
1404             //if (memcmp(&y[i],x,m-1) == 0) OUTPUT(i);
1405         // Below line gives: 315KB/clock
1406         //if (ch ==lastch && y[i] == firstch && memcmp(&y[i],x,m-1) == 0) return(i); // Kaze: The idea(to prevent execution of slower
1407         'memcmp') is borrowed from Karp-Rabin i.e. to perform a slower check only when the target "looks like".

```

```

1407 // Below line gives: 328KB/clock
1408     if (ch ==x[m-1] && y[i] == x[0] && memcmp(&y[i],x,m-1) == 0) return(i); // Kaze: The idea(to prevent execution of slower 'memcmp')
    is borrowed from Karp-Rabin i.e. to perform a slower check only when the target "looks like".
1409     i+=bm_bc[ch];
1410 }
1411 return(-1);
1412 }
1413 // ### Boyer-Moore-Horspool algorithm ]
1414
1415 // ### Brute force 'Dummy' algorithm [
1416 long Brute_Force_Dummy(char *y, char *x, long n, int m) {
1417     long i, j;
1418     /* Searching */
1419     for (i=0; i <= n-m; i++) {
1420         j=0;
1421         while (j < m && y[i+j] == x[j]) j++;
1422         if (j >= m) return(i);
1423     }
1424     return(-1);
1425 }
1426 // ### Brute force 'Dummy' algorithm ]
1427
1428 // ### Karp-Rabin algorithm [
1429 #define REHASH(a, b, h) (((h) - (a)*d) << 1) + (b)
1430 long Karp_Rabin(char *y, char *x, long n, int m) {
1431     int d, hx, hy, i, j;
1432     /* Preprocessing */
1433     /* computes d = 2^(m-1) with
1434     the left-shift operator */
1435     for (d = i = 1; i < m; ++i)
1436         d = (d<<1);
1437     for (hx = hy = i = 0; i < m; ++i) {
1438         hx = ((hx<<1) + x[i]);
1439         hy = ((hy<<1) + y[i]);
1440     }
1441     /* Searching */
1442     j = 0;
1443     while (j <= n-m) {
1444         if (hx == hy && memcmp(x, y + j, m) == 0) return(j);
1445         hx = REHASH(y[j], y[j + m], hy);
1446         ++j;
1447     }
1448     return(-1);
1449 }
1450 // ### Karp-Rabin algorithm ]
1451
1452 // ### Karp-Rabin-kaze algorithm [
1453 char * karpRabinKaze (char * pbTarget,
1454 char * pbPattern,
1455 unsigned long cbTarget,
1456 unsigned long cbPattern)
1457 {
1458     unsigned int i;
1459     char * pbTargetMax = pbTarget + cbTarget;
1460     char * pbPatternMax = pbPattern + cbPattern;
1461     unsigned long ulBaseToPowerMod = 1;
1462     register unsigned long ulHashPattern = 0;
1463     unsigned long ulHashTarget = 0;
1464     long hits = 0;
1465     //unsigned long count;
1466     //char * buf1;
1467     //char * buf2;
1468     if (cbPattern > cbTarget)
1469         return(NULL);
1470     // Compute the power of the left most character in base ulBase
1471     //for (i = 1; i < cbPattern; i++) ulBaseToPowerMod = (ulBase * ulBaseToPowerMod);
1472     // Calculate the hash function for the src (and the first dst)
1473     while (pbPattern < pbPatternMax)
1474     {
1475         // Below lines give 366KB/clock for 'underdog':
1476         //ulHashPattern = (ulHashPattern*ulBase + *pbPattern);
1477         //ulHashTarget = (ulHashTarget*ulBase + *pbTarget);
1478         pbPattern++;
1479         pbTarget++;
1480     }
1481     // Below lines give 436KB/clock for 'underdog' + requirement pattern to be 4 chars min.:
1482     //ulHashPattern = ( (* (long *) (pbPattern-cbPattern)) & 0xffffffff ) + *(pbPattern-1);
1483     //ulHashTarget = ( (* (long *) (pbTarget-cbPattern)) & 0xffffffff ) + *(pbTarget-1);
1484 }

```

```

1494 // Below lines give 482KB/clock for 'underdog' + requirement pattern to be 2 chars min.:
1495 //ulHashPattern = ( (* (unsigned short *) (pbPattern-cbPattern)) | *(pbPattern-1) );
1496 //ulHashTarget = ( (* (unsigned short *) (pbTarget-cbPattern)) | *(pbTarget-1) );
1497 // Below lines give 482KB/clock for 'underdog' + requirement pattern to be 2 chars min.:
1498 //ulHashPattern = ( (* (unsigned short *) (pbPattern-cbPattern)) & 0xffff0 ) + *(pbPattern-1);
1499 //ulHashTarget = ( (* (unsigned short *) (pbTarget-cbPattern)) & 0xffff0 ) + *(pbTarget-1);
1500 // Below lines give 605KB/clock for 'underdog' + requirement pattern to be 2 chars min.:
1501 //ulHashPattern = ( (* (unsigned short *) (pbPattern-cbPattern)) << 8 ) + *(pbPattern-1);
1502 //ulHashTarget = ( (* (unsigned short *) (pbTarget-cbPattern)) << 8 ) + *(pbTarget-1);
1503 // Below lines give 668KB/clock for 'underdog':
1504 ulHashPattern = ( (* (char *) (pbPattern-cbPattern)) << 8 ) + *(pbPattern-1);
1505 ulHashTarget = ( (* (char *) (pbTarget-cbPattern)) << 8 ) + *(pbTarget-1);
1506
1507 // Dynamically produce hash values for the string as we go
1508 for ( ;; )
1509 {
1510     if ( (ulHashPattern == ulHashTarget) && !memcmp(pbPattern-cbPattern, pbTarget-cbPattern, (unsigned int)cbPattern) )
1511     // if ( ulHashPattern == ulHashTarget ) {
1512     //
1513     // count = cbPattern;
1514     // buf1 = pbPattern-cbPattern;
1515     // buf2 = pbTarget-cbPattern;
1516     // while ( --count && *(char *)buf1 == *(char *)buf2 ) {
1517     //     buf1 = (char *)buf1 + 1;
1518     //     buf2 = (char *)buf2 + 1;
1519     // }
1520     // if ( *( (unsigned char *)buf1 ) - *( (unsigned char *)buf2 ) == 0) hits++;
1521     // }
1522     // return((pbTarget-cbPattern));
1523     // hits++;
1524
1525     if (pbTarget == pbTargetMax)
1526         return(NULL);
1527
1528     // Below line gives 482KB/clock for 'underdog' + requirement pattern to be 2 chars min.:
1529     //ulHashTarget = ( (* (unsigned short *) (pbTarget+1-cbPattern)) | *pbTarget );
1530     // Below line gives 436KB/clock for 'underdog' + requirement pattern to be 4 chars min.:
1531     //ulHashTarget = ( (* (long *) (pbTarget+1-cbPattern)) & 0xffffffff ) + *pbTarget;
1532 //; Line 696
1533     movsx esi, BYTE PTR [ebx]
1534     mov ecx, DWORD PTR [edx+1]
1535     and ecx, -256 ; ffffffff00h
1536     add ecx, esi
1537 // Below line gives 482KB/clock for 'underdog' + requirement pattern to be 2 chars min.:
1538 //ulHashTarget = ( (* (unsigned short *) (pbTarget+1-cbPattern)) & 0xffff0 ) + *pbTarget;
1539 //; Line 691
1540     movsx esi, BYTE PTR [ebx]
1541     xor ecx, ecx
1542     mov cx, WORD PTR [edx+1]
1543     and ecx, 65280 ; 0000ff00h
1544     add ecx, esi
1545 // Below line gives 605KB/clock for 'underdog' + requirement pattern to be 2 chars min.:
1546 //ulHashTarget = ( (* (unsigned short *) (pbTarget+1-cbPattern)) << 8 ) + *pbTarget;
1547 // Below line gives 668KB/clock for 'underdog':
1548 ulHashTarget = ( (* (char *) (pbTarget+1-cbPattern)) << 8 ) + *pbTarget;
1549 //; Line 718
1550     movsx ecx, BYTE PTR [eax+1]
1551     movsx edx, BYTE PTR [ebp]
1552     shl ecx, 8
1553     add ecx, edx
1554 // Below line gives 366KB/clock for 'underdog':
1555 //ulHashTarget = (ulHashTarget - *(pbTarget-cbPattern)*ulBaseToPowerMod)*ulBase + *pbTarget;
1556     pbTarget++;
1557 }
1558 // ### Karp-Rabin-kaze algorithm ]
1559
1560 // ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm [
1561 // Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
1562 char * Railgun (char * pbTarget,
1563 char * pbPattern,
1564 unsigned long cbTarget,
1565 unsigned long cbPattern)
1566 {
1567     char * pbTargetMax = pbTarget + cbTarget;
1568     register unsigned long ulHashPattern;
1569     unsigned long ulHashTarget;
1570     unsigned long count;
1571     unsigned long countsSTATIC, countRemainder;
1572     long i; //BMH needed
1573     int a, j, bm_bc[ASIZE]; //BMH needed
1574     unsigned char ch; //BMH needed
1575     unsigned char lastch, firstch; //BMH needed
1576     if (cbPattern > cbTarget)

```

```

1582     return(NULL);
1583 }
1584     countSTATIC = cbPattern-2;
1585
1586 // Doesn't work when cbPattern = 1
1587 if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than
    'Boyer_Moore_Horspool'.
1588 {
1589     pbTarget = pbTarget+cbPattern;
1590     ulHashPattern = ( (*char *) (pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
1591
1592     for ( ;; )
1593     {
1594         // The line below gives for 'cbPattern'>=1:
1595         // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
1596         // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
1597 /*
1598     if ( ulHashPattern == ( (*char *) (pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned
1599 int)cbPattern )
1600         return((long)(pbTarget-cbPattern));
1601 */
1602
1603     // The fragment below gives for 'cbPattern'>=2:
1604     // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
1605     // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock
1606     if ( ulHashPattern == ( (*char *) (pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
1607         count = countSTATIC;
1608         while ( count && *(char *) (pbPattern+1+(countSTATIC-count)) == *(char *) (pbTarget-cbPattern+1+(countSTATIC-count)) ) {
1609             count--;
1610         }
1611         if ( count == 0 ) return((pbTarget-cbPattern));
1612     }
1613
1614     // The fragment below gives for 'cbPattern'>=2:
1615     // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
1616     // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
1617 /*
1618     if ( ulHashPattern == ( (*char *) (pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
1619         count = countSTATIC>>2;
1620         countRemainder = countSTATIC % 4;
1621
1622         while ( count && *(unsigned long *) (pbPattern+1+((count-1)<<2)) == *(unsigned long *) (pbTarget-cbPattern+1+((count-1)<<2)) ) {
1623             count--;
1624         }
1625         //if (count == 0) { // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skilllessness'; 312KB/clock when
1626         //1+1x4+2+1 bytes pattern: 'underdog'), otherwise 368KB/clock.
1627         while ( countRemainder && *(char *) (pbPattern+1+(countSTATIC-countRemainder)) == *(char *) (pbTarget-cbPattern+1+(countSTATIC-
1628         countRemainder)) ) {
1629             countRemainder--;
1630         }
1631         //if ( countRemainder == 0 ) return((long)(pbTarget-cbPattern));
1632         if ( count+countRemainder == 0 ) return((long)(pbTarget-cbPattern));
1633         //}
1634     }
1635 */
1636
1637     pbTarget++;
1638     if (pbTarget > pbTargetMax)
1639         return(NULL);
1640 }
1641 else
1642 {
1643     /* Preprocessing */
1644     for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
1645     for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
1646
1647     /* Searching */
1648     //lastch=pbPattern[cbPattern-1];
1649     //firstch=pbPattern[0];
1650     i=0;
1651     while (i <= cbTarget-cbPattern) {
1652         ch=pbTarget[i+cbPattern-1];
1653         //if (ch == lastch)
1654         //if (memcmp(&pbTarget[i],pbPattern,cbPattern-1) == 0) OUTPUT(i);
1655         //if (ch == lastch && pbTarget[i] == firstch && memcmp(&pbTarget[i],pbPattern,cbPattern-1) == 0) return(i); // Kaze: The idea(to
1656         prevent execution of slower 'memcmp') is borrowed from karp-rabin i.e. to perform a slower check only when the target "looks like".
1657         if (ch == pbPattern[cbPattern-1] && pbTarget[i] == pbPattern[0])
1658         {
1659             count = countSTATIC;
1660             while ( count && *(char *) (pbPattern+1+(countSTATIC-count)) == *(char *) (&pbTarget[i]+1+(countSTATIC-count)) ) {
1661                 count--;
1662             }
1663             if ( count == 0 ) return(pbTarget+i);
1664         }
1665         i+=bm_bc[ch];
1666     }
1667 }

```

```

1665     return(NULL);
1666 }
1667 }
1668 // ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm ]
1669
1670
1671 // ### Railgun_totalhits [
1672 char * Railgun_totalhits (char * pbTarget,
1673     char * pbPattern,
1674     unsigned long cbTarget,
1675     unsigned long cbPattern)
1676 {
1677     char * pbTargetMax = pbTarget + cbTarget;
1678     register unsigned long ulHashPattern;
1679     unsigned long ulHashTarget;
1680     unsigned long count;
1681     unsigned long countsSTATIC, countRemainder;
1682
1683     long i; //BMH needed
1684     int a, j, bm_bc[ASIZE]; //BMH needed
1685     unsigned char ch; //BMH needed
1686     // unsigned char lastch, firstch; //BMH needed
1687
1688     if (cbPattern > cbTarget)
1689         return(NULL);
1690
1691     countSTATIC = cbPattern-2;
1692
1693     // Doesn't work when cbPattern = 1
1694     if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than
        'Boyer_Moore_Horspool'.
1695     {
1696         pbTarget = pbTarget+cbPattern;
1697         ulHashPattern = ( (*char *) (pbPattern))<<8 ) + *(pbPattern+(cbPattern-1));
1698
1699         for ( ;; )
1700         {
1701             // The line below gives for 'cbPattern'>=1:
1702             // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
1703             // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
1704 /*
1705     if ( ulHashPattern == ( (*char *) (pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned
1706 int)cbPattern )
1707         return((long)(pbTarget-cbPattern));
1708 */
1709
1710     // The fragment below gives for 'cbPattern'>=2:
1711     // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
1712     // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock
1713     if ( ulHashPattern == ( (*char *) (pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
1714         count = countSTATIC;
1715         while ( count && *(char *) (pbPattern+1+(countSTATIC-count)) == *(char *) (pbTarget-cbPattern+1+(countSTATIC-count)) ) {
1716             count--;
1717         }
1718         if ( count == 0 ) Railgunhits++; //return((pbTarget-cbPattern));
1719     }
1720
1721     // The fragment below gives for 'cbPattern'>=2:
1722     // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
1723     // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
1724 /*
1725     if ( ulHashPattern == ( (*char *) (pbTarget-cbPattern))<<8 ) + *(pbTarget-1) ) {
1726         count = countSTATIC>>2;
1727         countRemainder = countSTATIC % 4;
1728
1729         while ( count && *(unsigned long *) (pbPattern+1+((count-1)<<2)) == *(unsigned long *) (pbTarget-cbPattern+1+((count-1)<<2)) ) {
1730             count--;
1731         }
1732         //if (count == 0) { // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skilllessness'; 312KB/clock when
1733         //1+1x4+2+1 bytes pattern: 'underdog'), otherwise 368KB/clock.
1734         while ( countRemainder && *(char *) (pbPattern+1+(countSTATIC-countRemainder)) == *(char *) (pbTarget-cbPattern+1+(countSTATIC-
1735         countRemainder)) ) {
1736             countRemainder--;
1737         }
1738         //if ( countRemainder == 0 ) return((long)(pbTarget-cbPattern));
1739         if ( count+countRemainder == 0 ) return((long)(pbTarget-cbPattern));
1740         //}
1741     }
1742 */
1743
1744     pbTarget++;
1745     if (pbTarget > pbTargetMax)
1746         return(NULL);
1747 }
1748 else
1749 {

```

```

1749 /* Preprocessing */
1750 for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
1751 for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
1752
1753 /* Searching */
1754 //lastch=pbPattern[cbPattern-1];
1755 //firstch=pbPattern[0];
1756 i=0;
1757 while (i <= cbTarget-cbPattern) {
1758     ch=pbTarget[i+cbPattern-1];
1759     //if (ch ==lastch)
1760     //if (memcmp(&pbTarget[i],pbPattern,cbPattern-1) == 0) OUTPUT(i);
1761     //if (ch == lastch && pbTarget[i] == firstch && memcmp(&pbTarget[i],pbPattern,cbPattern-1) == 0) return(i); // Kaze: The idea(to
prevent execution of slower 'memcmp') is borrowed from Karp-Rabin i.e. to perform a slower check only when the target "looks like".
1762     if (ch == pbPattern[cbPattern-1] && pbTarget[i] == pbPattern[0])
1763     {
1764         count = countSTATIC;
1765         while ( count && *(char *) (pbPattern+1+(countSTATIC-count)) == *(char *) (&pbTarget[i]+(countSTATIC-count)) ) {
1766             count--;
1767         }
1768         if ( count == 0) Railgunhits++; //return(pbTarget+i);
1769     }
1770     i+=bm_bc[ch];
1771 }
1772 return(NULL);
1773 }
1774 }
1775 // ### Railgun_totalhits ]
1776
1777
1778 // ### Karp-Rabin-kaze_BOOSTED algorithm [
1779 char * karpRabinKaze_BOOSTED (char * pbTarget,
1780 char * pbPattern,
1781 unsigned long cbTarget,
1782 unsigned long cbPattern)
1783 {
1784     char * pbTargetMax = pbTarget + cbTarget;
1785     register unsigned long ulHashPattern;
1786     unsigned long ulHashTarget;
1787
1788     if (cbPattern > cbTarget)
1789         return(NULL);
1790
1791     pbTarget = pbTarget+cbPattern;
1792     ulHashPattern = ( *(char *) (pbPattern)<<8 ) + *(pbPattern+(cbPattern-1));
1793
1794     for ( ;; )
1795     {
1796         // Kaze: The idea(FAILED) here is to add an additional(second) layer in order to prevent execution of slower hash calculation(i.e.
1797         first layer) which(hash) prevents execution of even slower 'memcmp'.
1798         // The line below gives: 314KB/clock
1799         //if ( *pbPattern == *(char *) (pbTarget-cbPattern) && (ulHashPattern == ( *(char *) (pbTarget-cbPattern)<<8 ) + *(pbTarget-1) ) &&
1800         !memcmp(pbPattern, pbTarget-cbPattern, (unsigned int)cbPattern) )
1801         // The line below gives: 370KB/clock
1802         if ( (ulHashPattern == ( *(char *) (pbTarget-cbPattern)<<8 ) + *(pbTarget-1) ) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned
int)cbPattern) )
1803             return((pbTarget-cbPattern));
1804
1805         pbTarget++;
1806         if (pbTarget > pbTargetMax)
1807             return(NULL);
1808     }
1809 }
1810 // ### Karp-Rabin-kaze_BOOSTED algorithm ]
1811
1812 char * strstr_Microsoft (
1813     const char * str1,
1814     const char * str2
1815 )
1816 {
1817     char *cp = (char *) str1;
1818     char *s1, *s2;
1819
1820     if ( !*str2 )
1821         return((char *)str1);
1822
1823     while (*cp)
1824     {
1825         s1 = cp;
1826         s2 = (char *) str2;
1827
1828         while ( *s1 && *s2 && !(*s1-*s2) )
1829             s1++, s2++;
1830
1831         if (!*s2)
1832             return(cp);
1833
1834         cp++;

```

```

1833     }
1834     return(NULL);
1835 }
1836
1837 char *
1838 strstr_GNU_C_Library (phaystack, pneedle)
1839     const char *phaystack;
1840     const char *pneedle;
1841 {
1842     const unsigned char *haystack, *needle;
1843     char b;
1844     const unsigned char *rneedle;
1845
1846     haystack = (const unsigned char *) phaystack;
1847
1848     if ((b = *(needle = (const unsigned char *) pneedle)))
1849     {
1850         char c;
1851         haystack--; /* possible ANSI violation */
1852
1853         {
1854             char a;
1855             do
1856             {
1857                 if (!(a = *++haystack))
1858                     goto ret0;
1859                 while (a != b);
1860             }
1861             if (!(c = *++needle))
1862                 goto foundneedle;
1863             ++needle;
1864             goto jin;
1865
1866             for (;;)
1867             {
1868                 {
1869                     char a;
1870                     if (0)
1871                         jin: {
1872                             if ((a = *++haystack) == c)
1873                                 goto crest;
1874                         }
1875                     else
1876                         a = *++haystack;
1877                     do
1878                     {
1879                         for (; a != b; a = *++haystack)
1880                         {
1881                             if (!a)
1882                                 goto ret0;
1883                             if ((a = *++haystack) == b)
1884                                 break;
1885                             if (!a)
1886                                 goto ret0;
1887                         }
1888                         while ((a = *++haystack) != c);
1889                     }
1890                 }
1891                 crest:
1892                 {
1893                     char a;
1894                     {
1895                         const unsigned char *rhaystack;
1896                         if (*(rhaystack = haystack-- + 1) == (a = *(rneedle = needle)))
1897                             do
1898                             {
1899                                 if (!a)
1900                                     goto foundneedle;
1901                                 if (*++rhaystack != (a = *++needle))
1902                                     break;
1903                                 if (!a)
1904                                     goto foundneedle;
1905                             }
1906                             while (*++rhaystack == (a = *++needle));
1907                             needle = rneedle; /* took the register-poor approach */
1908                         }
1909                     if (!a)
1910                         break;
1911                 }
1912             }
1913         }
1914         foundneedle:
1915         return (char *) haystack;
1916     }
1917     return 0;
1918 }
1919
1920

```

```

1921 // ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm [
1922 // Caution: For better speed the case 'if (cbPattern==1)' was removed, so Pattern must be longer than 1 char.
1923 char * Ra1gun_Quaduplet (char * pbTarget,
1924     char * pbPattern,
1925     unsigned long cbTarget,
1926     unsigned long cbPattern)
1927 {
1928     char * pbTargetMax = pbTarget + cbTarget;
1929     register unsigned long ulHashPattern;
1930     unsigned long ulHashTarget;
1931     unsigned long count;
1932     unsigned long countSTATIC;
1933     unsigned long countRemainder;
1934
1935 /*
1936     const unsigned char SINGLET = *(char *) (pbPattern);
1937     const unsigned long Quaduplet2nd = SINGLET<<8;
1938     const unsigned long Quaduplet3rd = SINGLET<<16;
1939     const unsigned long Quaduplet4th = SINGLET<<24;
1940 */
1941     unsigned char SINGLET;
1942     unsigned long Quaduplet2nd;
1943     unsigned long Quaduplet3rd;
1944     unsigned long Quaduplet4th;
1945
1946     unsigned long AdvanceHopperGrass;
1947
1948     long i; //BMH needed
1949     int a, j, bm_bc[ASIZE]; //BMH needed
1950     unsigned char ch; //BMH needed
1951     unsigned char lastch, firstch; //BMH needed
1952
1953     if (cbPattern > cbTarget)
1954         return(NULL);
1955
1956     // Doesn't work when cbPattern = 1
1957     // The next IF-fragment works very well with cbPattern>1, OBVIOUSLY IT MUST BE UNROLLED(but crippled with less functionality) SINCE either
1958     // cbPattern=2 or cbPattern=3!
1959     if ( cbPattern<4 ) { // This IF makes me unhappy: it slows down from 390KB/clock to 367KB/clock for 'fast' pattern. This fragment(for 2..3
1960     // pattern lengths) is needed because I need a function different than strchr but sticking to strchr i.e. lengths above 1 are to be handled.
1961     // pbTarget = pbTarget+cbPattern;
1962     // ulHashPattern = ( *(char *) (pbPattern)<<8 ) + *(pbPattern+(cbPattern-1));
1963     // countSTATIC = cbPattern-2;
1964     for ( ;; )
1965     {
1966         // The line below gives for 'cbPattern'>=1:
1967         // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/543
1968         // Karp_Rabin_Kaze_4_OCTETS performance: 372KB/clock
1969         if ( (ulHashPattern == ( *(char *) (pbTarget-cbPattern)<<8 ) + *(pbTarget-1) ) && !memcmp(pbPattern, pbTarget-cbPattern, (unsigned
1970         int)cbPattern) )
1971             return((long)(pbTarget-cbPattern));
1972
1973         // The fragment below gives for 'cbPattern'>=2:
1974         // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/546
1975         // Karp_Rabin_Kaze_4_OCTETS performance: 370KB/clock
1976
1977         if ( ulHashPattern == ( *(char *) (pbTarget-cbPattern)<<8 ) + *(pbTarget-1) ) {
1978             count = countSTATIC;
1979             while ( count && *(char *) (pbPattern+1+(countSTATIC-count)) == *(char *) (pbTarget-cbPattern+1+(countSTATIC-count)) ) {
1980                 count--;
1981             }
1982             if ( count == 0 ) return((pbTarget-cbPattern));
1983         }
1984
1985         // The fragment below gives for 'cbPattern'>=2:
1986         // Karp_Rabin_Kaze_4_OCTETS_hits/Karp_Rabin_Kaze_4_OCTETS_clocks: 4/554
1987         // Karp_Rabin_Kaze_4_OCTETS performance: 364KB/clock
1988         if ( ulHashPattern == ( *(char *) (pbTarget-cbPattern)<<8 ) + *(pbTarget-1) ) {
1989             count = countSTATIC>2;
1990             countRemainder = countSTATIC % 4;
1991
1992             while ( count && *(unsigned long *) (pbPattern+1+((count-1)<<2)) == *(unsigned long *) (pbTarget-cbPattern+1+((count-1)<<2)) ) {
1993                 count--;
1994             }
1995
1996             //if (count == 0) { // Disastrous degradation only from this line(317KB/clock when 1+2x4+2+1 bytes pattern: 'skilllessness'; 312KB/clock when
1997             // 1+1x4+2+1 bytes pattern: 'underdog' ); otherwise 368KB/clock.
1998             while ( countRemainder && *(char *) (pbPattern+1+(countSTATIC-countRemainder)) == *(char *) (pbTarget-cbPattern+1+(countSTATIC-
1999             countRemainder)) ) {
2000                 countRemainder--;
2001             }
2002             //if ( countRemainder == 0 ) return((long)(pbTarget-cbPattern));
2003             if ( count+countRemainder == 0 ) return((long)(pbTarget-cbPattern));
2004         }
2005     }

```

```

2004 */
2005
2006     pbTarget++;
2007     if (pbTarget > pbTargetMax)
2008         return(NULL);
2009 }
2010 } else { //if ( cbPattern<4)
2011     if (cbTarget<961) // This value is arbitrary(don't know how exactly), it ensures(at least must) better performance than
2012     // 'Boyer_Moore_Horspool'.
2013     {
2014         pbTarget = pbTarget+cbPattern;
2015         ulHashPattern = *(unsigned long *) (pbPattern);
2016         countSTATIC = cbPattern-1;
2017
2018         //SINGLET = *(char *) (pbPattern);
2019         SINGLET = ulHashPattern & 0xFF;
2020         Quaduplet2nd = SINGLET<<8;
2021         Quaduplet3rd = SINGLET<<16;
2022         Quaduplet4th = SINGLET<<24;
2023     }
2024     for ( ;; )
2025     {
2026         AdvanceHopperGrass = 0;
2027         ulHashTarget = *(unsigned long *) (pbTarget-cbPattern);
2028
2029         if ( ulHashPattern == ulHashTarget ) { // Three unnecessary comparisons here, but 'AdvanceHopperGrass' must be calculated - it has a
2030         // higher priority.
2031             count = countSTATIC;
2032             while ( count && *(char *) (pbPattern+1+(countSTATIC-count)) == *(char *) (pbTarget-cbPattern+1+(countSTATIC-count)) ) {
2033                 if ( countSTATIC==AdvanceHopperGrass+count && SINGLET != *(char *) (pbTarget-cbPattern+1+(countSTATIC-count)) ) AdvanceHopperGrass++;
2034                 count--;
2035             }
2036             if ( count == 0 ) return((pbTarget-cbPattern));
2037         } else { // The goal here: to avoid memory accesses by stressing the registers.
2038             if ( Quaduplet2nd != (ulHashTarget & 0x0000FF00) ) {
2039                 AdvanceHopperGrass++;
2040                 if ( Quaduplet3rd != (ulHashTarget & 0x00FF0000) ) {
2041                     AdvanceHopperGrass++;
2042                     if ( Quaduplet4th != (ulHashTarget & 0xFF000000) ) AdvanceHopperGrass++;
2043                 }
2044             }
2045         }
2046         AdvanceHopperGrass++;
2047     }
2048     pbTarget = pbTarget + AdvanceHopperGrass;
2049     if (pbTarget > pbTargetMax)
2050         return(NULL);
2051 } else { //if ( cbTarget<961)
2052     countSTATIC = cbPattern-2;
2053     /* Preprocessing */
2054     for (a=0; a < ASIZE; a++) bm_bc[a]=cbPattern;
2055     for (j=0; j < cbPattern-1; j++) bm_bc[pbPattern[j]]=cbPattern-j-1;
2056
2057     /* Searching */
2058     //lastch=pbPattern[cbPattern-1];
2059     //firstch=pbPattern[0];
2060     i=0;
2061     while ( i <= cbTarget-cbPattern ) {
2062         ch=pbTarget[i+cbPattern-1];
2063         //if (ch == lastch)
2064         //if ( memcmp(&pbTarget[i], pbPattern, cbPattern-1) == 0 ) OUTPUT(i);
2065         //if (ch == lastch && pbTarget[i] == firstch && memcmp(&pbTarget[i], pbPattern, cbPattern-1) == 0) return(i); // Kaze: The idea(to
2066         // prevent execution of slower 'memcmp') is borrowed from Karp-Rabin i.e. to perform a slower check only when the target "looks like".
2067         if (ch == pbPattern[cbPattern-1] && pbTarget[i] == pbPattern[0])
2068             {
2069                 count = countSTATIC;
2070                 while ( count && *(char *) (pbPattern+1+(countSTATIC-count)) == *(char *) (&pbTarget[i]+1+(countSTATIC-count)) ) {
2071                     count--;
2072                 }
2073                 if ( count == 0 ) return(pbTarget+i);
2074             }
2075         i+=bm_bc[ch];
2076     }
2077     return(NULL);
2078 } //if ( cbTarget<961)
2079 } //if ( cbPattern<4)
2080 // ### Mix(2in1) of Karp-Rabin & Boyer-Moore-Horspool algorithm ]
2081
2082 void strip_ext(unsigned char *str)
2083 {
2084     int i;
2085     for(i = strlen(str)-1; str[i] != '.' && i >= 0; i--)
2086         ;
2087     if(i > 0)
2088         str[i] = '\0';

```

```

2089 }
2090
2091 int main( int argc, char *argv[] )
2092
2093 {
2094 FILE *fp_inLINE;
2095 FILE *fp_outLINE;
2096 int Bozans;
2097 long ThunderwithL, ThunderwithR;
2098 char *Strng;
2099 char *BB;
2100 long Strnglen;
2101 long StrnglenTRAVERSED;
2102 char Pattern[20+2000]; // skilllessness=12 human consciousness=19 I should have known=19
2103 // In the East, enlightenment is described as a state of ultimate=62
2104 //int Patternlen; // Make it global in sake of QSORT
2105 long LinesEncountered=0;
2106 long BruteForceDummyhits=0;
2107 long KarpRabinkazehits=0;
2108 long KarpRabinkaze_BOOSTEDhits=0;
2109 long Karp_Rabin_Kaze_4_OCTETShits=0;
2110 long KarpRabinhits=0;
2111 long HORSPOOLhits=0;
2112 long HORSPOOL_kazehits=0;
2113 long strstrMicrosofthits=0;
2114 long strstrGNUCLibraryhits=0;
2115
2116 unsigned char *bufEXTENSION;
2117
2118 //int i, j;
2119 int i,j;
2120 register int iREGISTER;
2121 register int jREGISTER;
2122
2123 char *Dumbobox[8][2] = { "an\0", "to\0",
2124 "TDK\0", "che\0",
2125 "fast\0", "easy\0",
2126 "grmb\0", "emai\0",
2127 "pasting\0", "amazing\0",
2128 "underdog\0", "superdog\0",
2129 "participants\0", "skilllessness\0",
2130 "I should have known\0", "human consciousness\0"
2131 };
2132
2133 long FoundIn;
2134 char *FoundInPTR, *FoundInPTRNEW;
2135 char l1toADigits[27]; // 9,223,372,036,854,775,807: 1(sign or carry)+19(digits)+1('0')+6(,)
2136 char BuildingBlocksBox[10] = "BB---.txt\0";
2137
2138 int nLines = 0;
2139 string basep, cur, next;
2140 string *ASbackup;
2141
2142 unsigned long nLines1, linecounterNotRL, linecounterRL, jPrevLine, jFirst;
2143
2144 signed long GainMax=0x80000000; //-2147483648
2145 int GainOrder=0; register int GainOrderREGISTER;
2146 signed int Nataliakills;
2147 int YabbaYabba = 0;
2148 long LeftPointBS, RightPointBS, PivotPointBS;
2149 char SShheader[270];
2150
2151 printf("Simplicius.Simplicissimus.Septupleton rev.2-, written by Kaze.\n");
2152 printf("Note1: This is the precursor of Simplicius.Simplicissimus - a superfast-low-performance TEXT decompressor.\n");
2153 printf("Note2: The niftiness lies in readiness for multi-threading and mostly in boosting the search by having the BBS.\n");
2154 printf("Note3: The decompressor would upload (at burst speed) the BB data and then read-and-decode one-by-one the\n");
2155 printf("triads (BB pool/array indexes), that is a simple copying. The pool houses up to 256*256*256 BBS/elements.\n");
2156 printf("Note4: Compiler used: Intel(R) C++ Compiler XE for applications running on IA-32, Version 12.0.4.196 Build 20110427.\n");
2157 printf("Note5: Compile line: cl /Ox /TCBuilding-Blocks_DUMPER.c /FABuilding-Blocks_DUMPER /w /FACS\n");
2158 printf("Note6: Maximum size of input text file is arbitrary - 384MB, in fact 384MB+4*384MB must be less than 192?MB.\n");
2159 printf("Note7: The file format is given by typing the compressed file e.g. D:\>type OSHO.TXT.SS.\n");
2160 printf("Note8: The header is 270 bytes long followed by BBS and triads/indexes and the eventual remainder/literals.\n");
2161 printf("Note9: My benchmark text file OSHO.TXT 206,908,949 bytes where OSHO.TXT.SS is 116,871,584 bytes for order 6.\n");
2162 printf("Decompressing OSHO.TXT.SS to RAM without Dumping to DRIVE time: 1704 clocks or 118579 KB/s, an awful result.\n");
2163 printf("For order 4 enforced: 156,174,067 OSHO.TXT.SS is being decompressed at 192804 KB/s.\n");
2164 printf("For order 7 enforced: 122,297,608 OSHO.TXT.SS is being decompressed at 85618 KB/s.\n");
2165 printf("For order 8 enforced: 149,243,106 OSHO.TXT.SS is being decompressed at 115396 KB/s.\n");
2166 printf("Obviously the fastest cache size is crucial, for OSHO.TXT 12MB BB pool vs 1MB L2 cache disbalance is the\n");
2167 printf("cause for this badly inferior performance compared to LZ L1 (32KB) cache-friendly variants.\n");
2168 printf("The test machine is Toshiba Satellite with Intel Merom 2166MHz, with rough 'mempcy' performance: 1646 MB/s.\n");
2169 printf("Note8: Compile line: cl /Ox /TCBuilding-Blocks_DUMPER.c /FABuilding-Blocks_DUMPER /w /FACS\n");
2170 printf("Note9: Major (but still inferior) decompressing tweak since r.1+++ , this time Microsoft v16 excels:\n");
2171 printf("For order 7 enforced: 122,297,608 OSHO.TXT.SS is being decompressed at 170083 KB/s.\n");
2172 printf("NoteA: Simplicius.Simplicissimus stands up for his name, the decompressing fragment is 13 instructions small:\n");
2173 printf("$LL74@main:\n");
2174 printf("a1 00 00 00 mov eax, DWORD PTR _Patternlen\n");
2175 printf("03 c2 add eax, edx\n");
2176 printf("8b 04 30 mov eax, DWORD PTR [eax+esi]\n");

```

```

2177 printf(" 25 ff ff ff 00 and eax, 16777215\n");
2178 printf(" 0f af c7 imul eax, edi\n");
2179 printf(" 8b 1c 30 mov ebx, DWORD PTR [eax+esi]\n");
2180 printf(" 89 19 mov DWORD PTR [ecx], ebx\n");
2181 printf(" 8b 44 30 04 mov eax, DWORD PTR [eax+esi+4]\n");
2182 printf(" 89 41 04 mov DWORD PTR [ecx+4], eax\n");
2183 printf(" 83 c2 03 add edx, 3\n");
2184 printf(" 03 cf add ecx, edi\n");
2185 printf(" ff 4c 24 1c dec DWORD PTR tv977[esp+376]\n");
2186 printf(" 75 d7 jne SHORT $LL74@main\n");
2187 printf("NoteB: For more fast decompressing idea(s) you may contact me at sanmayce@sanmayce.com freely.\n");
2188 printf("NoteC: Got it? The B note has two meanings: 1] for faster 2] for additional/other fast.\n");
2189 printf("NoteD: Got it? The C note has two meanings: 1] for quicker/tighter 2] for additional/other quick/tight.\n");
2190 printf("NoteE: Since r.2- a new Quicksort function is in use due to the bugginess of the old.\n");
2191 printf("NoteF: A stupid bug (one triad was missing) was crushed also.\n");
2192
2193
2194 Pattern[0]=0x00;
2195
2196 if( argc != 2 )
2197 { printf( "\nUsage: Building-Blocks_DUMPER textfilename\n" );
2198 printf( "Example1: Building-Blocks_DUMPER The_Little_Match_Girl.txt\n" );
2199 printf( "Example2: Building-Blocks_DUMPER The_Little_Match_Girl.txt.SS\n" );
2200 printf( "Note: When the extension is not .SS then compression is commenced otherwise decompression.\n" );
2201 return( 1 ); }
2202
2203 if( ( fp_inLINE = fopen( argv[1], "rb" ) ) == NULL )
2204 { printf( "Building-Blocks_DUMPER: Can't open file %s \n", argv[1] ); return( 1 ); }
2205
2206 printf( "\nAllocating 384MB ... " );
2207 Strng = (char *)malloc( 384*1024*1024 +1); // +1 because one byte is read beyond pool!
2208 if( Strng == NULL )
2209 { puts( "Building-Blocks_DUMPER: Needed memory allocation denied!\n" ); return( 1 ); }
2210 printf( "OK\n" );
2211
2212 clocks3 = clock();
2213
2214 // DECOMPRESSOR [
2215 if(toupper(argv[1][strlen(argv[1])-1]) == 's' && toupper(argv[1][strlen(argv[1])-2]) == 's' && argv[1][strlen(argv[1])-3] == '.') {
2216 bufEXTENSION = (unsigned char *) malloc(sizeof(unsigned char) * (strlen(argv[1])+1)); //when checking for .SS
2217 strcpy(bufEXTENSION, argv[1]);
2218 strip_ext(bufEXTENSION);
2219 printf("\nSimplicius.Simplicissimus is decompressing %s...\n",bufEXTENSION);
2220 if( ( fp_outLINE = fopen( bufEXTENSION, "wb" ) ) == NULL )
2221 { printf( "Building-Blocks_DUMPER: Can't open file %s \n", bufEXTENSION ); return( 1 ); }
2222
2223 printf( "Allocating 256MB for BBS pool + 384MB for RAM-to-RAM decompression ... " ); // 16order*16,777,216BBS + (384MB/3order)*3bytes
2224 BB = (char *)malloc( 256*1024*1024 + 384*1024*1024 +1); // +1 because one byte is read beyond pool!
2225 if( BB == NULL )
2226 { puts( "Building-Blocks_DUMPER: Needed memory allocation denied!\n" ); return( 1 ); }
2227 printf( "OK\n" );
2228
2229 // Uploading BB file to BB pool:
2230
2231 fread(SSheader, 1, 270, fp_inLINE);
2232 fread((unsigned char *)&GainOrder+0, 1, 1, fp_inLINE);
2233 fread((unsigned char *)&Strnglen+0, 1, 4, fp_inLINE);
2234 fread((unsigned char *)&Patternlen+0, 1, 4, fp_inLINE);
2235 fread(BB, 1, Patternlen, fp_inLINE);
2236
2237 printf("BBS order: %lu\nOriginal file size: %lu\nBBS pool size: %lu\n,GainOrder,Strnglen,Patternlen);
2238
2239 /*
2240 // Not buffered input, slow slow slow - SHOULD BE quickened! No need of 384MB RAM-to-RAM block! [
2241 for( j = 0; j < Strnglen / GainOrder; j++ )
2242 {
2243 ThunderwithL=0; // Zeroing the highest byte i.e. the fourth
2244 fread((unsigned char *)&ThunderwithL+0, 1, 3, fp_inLINE);
2245 fwrite((unsigned char *)&BB+GainOrder*ThunderwithL, 1, GainOrder, fp_outLINE);
2246 }
2247 ThunderwithL=Strnglen % GainOrder;
2248 while (ThunderwithL) {
2249 fread((unsigned char *)&ThunderwithR+0, 1, 1, fp_inLINE);
2250 fwrite((unsigned char *)&ThunderwithR+0, 1, 1, fp_outLINE);
2251 ThunderwithL--;
2252 }
2253 // Not buffered input, slow slow slow - SHOULD BE quickened! No need of 384MB RAM-to-RAM block! ]
2254 /*
2255
2256 // Buffered input, just for OSHO.TXT because for max order 16 and max BBS 16,777,216 the BB pool cannot house the triads for bigger files! [
2257 fread((unsigned char *)&BB+Patternlen, 1, 3*(Strnglen / GainOrder), fp_inLINE);
2258
2259
2260 goto SubUGLYSLOW;
2261 // UGLY SLOW !!! [
2262 iREGISTER=Strnglen / GainOrder;
2263 GainOrderREGISTER=GainOrder;
2264

```

```

2265 if (Strnglen % GainOrderREGISTER)
2266 fread((unsigned char *)BB+Patternlen+3*(Strnglen / GainOrderREGISTER), 1, Strnglen % GainOrderREGISTER, fp_inLINE);
2267 for( jREGISTER = 0; jREGISTER < iREGISTER; jREGISTER++ )
2268 {
2269 //ThunderwithL=0; // Zeroing the highest byte i.e. the fourth
2270 //memcpy (&thunderwithL, BB+Patternlen+jREGISTER*3, 3);
2271 memcpy (Strng+GainOrderREGISTER*jREGISTER, BB+GainOrderREGISTER*( 0x00FFFFFF & *(unsigned long *) (BB+Patternlen+jREGISTER*3) ),
GainOrderREGISTER);
2272 }
2273 if (Strnglen % GainOrderREGISTER)
2274 memcpy (Strng+GainOrderREGISTER*(Strnglen / GainOrderREGISTER), BB+Patternlen+3*(Strnglen / GainOrderREGISTER), Strnglen % GainOrderREGISTER);
2275
2276 // UGLY SLOW !!! ]
2277
2278 // Microsoft compiler 16 gives:
2279 /*
2280 ; 1740 : for( jREGISTER = 0; jREGISTER < iREGISTER; jREGISTER++ )
2281
2282 016f1 85 db test ebx, ebx
2283 016f3 7e 4e jle SHORT $LN69@main
2284 016f5 8b 4c 24 2c mov eax, DWORD PTR _Strng$[esp+376]
2285 016f9 8b 4c 24 1c mov ecx, DWORD PTR tv1216[esp+376]
2286 016fd 33 db xor ebx, ebx
2287 016ff 89 44 24 14 mov DWORD PTR tv1897[esp+376], eax
2288 01703 89 4c 24 24 mov DWORD PTR tv1057[esp+376], ecx
2289 01707 eb 07 8d a4 24
2290 00 00 00 00 npad 9
2291 $LL71@main:
2292
2293 ; 1741 : {
2294 ; 1742 : //ThunderwithL=0; // Zeroing the highest byte i.e. the fourth
2295 ; 1743 : //memcpy (&thunderwithL, BB+Patternlen+jREGISTER*3, 3);
2296 ; 1744 : memcpy (Strng+GainOrderREGISTER*jREGISTER, BB+GainOrderREGISTER*( 0x00FFFFFF & *(unsigned long *) (BB+Patternlen+jREGISTER*3) ),
GainOrderREGISTER);
2297
2298 01710 8b 15 00 00 00
2299 00 mov edx, DWORD PTR _Patternlen
2300 01716 8d 04 13 lea eax, DWORD PTR [ebx+edx]
2301 01719 8b 0c 30 mov ecx, DWORD PTR [eax+esi]
2302 0171c 8b 54 24 14 mov edx, DWORD PTR tv1897[esp+376]
2303 01720 81 e1 ff ff ff
2304 00 and ecx, 16777215 ; 00ffffffH
2305 01726 0f af c7 imul ecx, edi
2306 01729 57 push edi
2307 0172a 03 ce add ecx, esi
2308 0172c 51 push ecx
2309 0172d 52 push edx
2310 0172e e8 00 00 00 00 call _memcpy
2311 01733 01 7c 24 20 add DWORD PTR tv1897[esp+388], edi
2312 01737 83 c4 0c add esp, 12 ; 0000000cH
2313 0173a 83 c3 03 add ebx, 3
2314 0173d ff 4c 24 24 dec DWORD PTR tv1057[esp+376]
2315 01741 75 cd jne SHORT $LL71@main
2316 $LN69@main:
2317
2318 ; 1745 : }
2319 */
2320
2321 SubUGLYSLOW:
2322 goto SubSubUGLYSLOW;
2323 // SUB UGLY SLOW !!! [
2324 iREGISTER=Strnglen / GainOrder;
2325 GainOrderREGISTER=GainOrder;
2326
2327 if (Strnglen % GainOrderREGISTER)
2328 fread((unsigned char *)BB+Patternlen+3*(Strnglen / GainOrderREGISTER), 1, Strnglen % GainOrderREGISTER, fp_inLINE);
2329 for( jREGISTER = 0; jREGISTER < iREGISTER; jREGISTER++ ) // Unroll by 4 triads in next r.1+,,, that is, read into 3 32bit registers 4
triads i.e. 12bytes and >> << &.
2330 {
2331 //ThunderwithL=0; // Zeroing the highest byte i.e. the fourth
2332 //memcpy (&thunderwithL, BB+Patternlen+jREGISTER*3, 3);
2333 //memcpy (Strng+GainOrderREGISTER*jREGISTER, BB+GainOrderREGISTER*( 0x00FFFFFF & *(unsigned long *) (BB+Patternlen+jREGISTER*3) ),
GainOrderREGISTER);
2334
2335 // r.1+,,, [
2336 //Dumbo64REGISTER = *(long long *) (BB+GainOrderREGISTER*( 0x00FFFFFF & *(unsigned long *) (BB+Patternlen+jREGISTER*3) ));
2337 /*(long long *) (Strng+GainOrderREGISTER*jREGISTER)=Dumbo64REGISTER;
2338 *(long long *) (Strng+GainOrderREGISTER*jREGISTER) = *(long long *) (BB+GainOrderREGISTER*( 0x00FFFFFF & *(unsigned long
*) (BB+Patternlen+jREGISTER*3) ));
2339 // r.1+,,, ]
2340 }
2341 if (Strnglen % GainOrderREGISTER)
2342 memcpy (Strng+GainOrderREGISTER*(Strnglen / GainOrderREGISTER), BB+Patternlen+3*(Strnglen / GainOrderREGISTER), Strnglen % GainOrderREGISTER);
2343
2344 // SUB UGLY SLOW !!! ]
2345
2346 // Microsoft compiler 16 gives:
2347 /*

```

```

2348 ; 1811 : for( jREGISTER = 0; jREGISTER < iREGISTER; jREGISTER++ ) // Unroll by 4 triads in next r.1+,,, that is, read into 3 32bit
registers 4 triads i.e. 12bytes and >> << &.
2349
2350 0173d 85 db test ebx, ebx
2351 0173f 7e 38 jle SHORT $LN72@main
2352 01741 8b 4c 24 18 mov ecx, DWORD PTR _Strng$[esp+376]
2353 01745 33 d2 xor edx, edx
2354 01747 89 5c 24 1c mov DWORD PTR tv977[esp+376], ebx
2355 0174b eb 03 8d 49 00 npad 5
2356 $LL74@main:
2357
2358 ; 1812 : {
2359 ; 1813 : //ThunderwithL=0; // Zeroing the highest byte i.e. the fourth
2360 ; 1814 : //memcpy (&thunderwithL, BB+Patternlen+jREGISTER*3, 3);
2361 ; 1815 : //memcpy (Strng+GainOrderREGISTER*jREGISTER, BB+GainOrderREGISTER*( 0x00FFFFFF & *(unsigned long *) (BB+Patternlen+jREGISTER*3) ),
GainOrderREGISTER);
2362 ; 1816 :
2363 ; 1817 : // r.1+,,, [
2364 ; 1818 : //Dumbo64REGISTER = *(long long *) (BB+GainOrderREGISTER*( 0x00FFFFFF & *(unsigned long *) (BB+Patternlen+jREGISTER*3) ));
2365 ; 1819 : /*(long long *) (Strng+GainOrderREGISTER*jREGISTER)=Dumbo64REGISTER;
2366 ; 1820 : *(long long *) (Strng+GainOrderREGISTER*jREGISTER) = *(long long *) (BB+GainOrderREGISTER*( 0x00FFFFFF & *(unsigned long
*) (BB+Patternlen+jREGISTER*3) ));
2367
2368 01750 a1 00 00 00 00 mov eax, DWORD PTR _Patternlen
2369 01755 03 c2 add eax, edx
2370 01757 8b 04 30 mov eax, DWORD PTR [eax+esi]
2371 0175a 25 ff ff ff 00 and eax, 16777215 ; 00ffffffH
2372 0175f 0f af c7 imul eax, edi
2373 01762 8b 1c 30 mov ebx, DWORD PTR [eax+esi]
2374 01765 89 19 mov DWORD PTR [ecx], ebx
2375 01767 8b 44 30 04 mov eax, DWORD PTR [eax+esi+4]
2376 0176b 89 41 04 mov DWORD PTR [ecx+4], eax
2377 0176e 83 c2 03 add edx, 3
2378 01771 03 cf add ecx, edi
2379 01773 ff 4c 24 1c dec DWORD PTR tv977[esp+376]
2380 01777 75 d7 jne SHORT $LL74@main
2381 $LN72@main:
2382
2383 ; 1821 : // r.1+,,, ]
2384 ; 1822 : }
2385 */
2386
2387 SubSubUGLYSLOW: /// Notice that 42<64, but the threefold unrolling boosts (despite of exceeding 64) from 16300KB/s to 17000KB/s.
2388 // SUB SUB UGLY SLOW !!! [
2389 iREGISTER=Strnglen / GainOrder; //iREGISTER=iREGISTER-(iREGISTER%3); Fixed BUG here, how stupid!
2390 GainOrderREGISTER=GainOrder;
2391
2392 if (Strnglen % GainOrderREGISTER)
2393 fread((unsigned char *)BB+Patternlen+3*(Strnglen / GainOrderREGISTER), 1, Strnglen % GainOrderREGISTER, fp_inLINE);
2394 for( jREGISTER = 0; jREGISTER < iREGISTER; jREGISTER++ )
2395 {
2396 //ThunderwithL=0; // Zeroing the highest byte i.e. the fourth
2397 //memcpy (&thunderwithL, BB+Patternlen+jREGISTER*3, 3);
2398 //memcpy (Strng+GainOrderREGISTER*jREGISTER, BB+GainOrderREGISTER*( 0x00FFFFFF & *(unsigned long *) (BB+Patternlen+jREGISTER*3) ),
GainOrderREGISTER);
2399
2400 // r.1+,,, [
2401 //Dumbo64REGISTER = *(long long *) (BB+GainOrderREGISTER*( 0x00FFFFFF & *(unsigned long *) (BB+Patternlen+jREGISTER*3) ));
2402 /*(long long *) (Strng+GainOrderREGISTER*jREGISTER)=Dumbo64REGISTER;
2403 *(long long *) (Strng+GainOrderREGISTER*jREGISTER) = *(long long *) (BB+GainOrderREGISTER*( 0x00FFFFFF & *(unsigned long
*) (BB+Patternlen+jREGISTER*3) ));
2404 jREGISTER++;
2405 *(long long *) (Strng+GainOrderREGISTER*jREGISTER) = *(long long *) (BB+GainOrderREGISTER*( 0x00FFFFFF & *(unsigned long
*) (BB+Patternlen+jREGISTER*3) ));
2406 jREGISTER++;
2407 *(long long *) (Strng+GainOrderREGISTER*jREGISTER) = *(long long *) (BB+GainOrderREGISTER*( 0x00FFFFFF & *(unsigned long
*) (BB+Patternlen+jREGISTER*3) ));
2408 // r.1+,,, ]
2409 }
2410 if ((Strnglen / GainOrder)%3==1)
2411 {
2412 jREGISTER++;
2413 *(long long *) (Strng+GainOrderREGISTER*jREGISTER) = *(long long *) (BB+GainOrderREGISTER*( 0x00FFFFFF & *(unsigned long
*) (BB+Patternlen+jREGISTER*3) ));
2414 }
2415 if ((Strnglen / GainOrder)%3==2)
2416 {
2417 jREGISTER++;
2418 *(long long *) (Strng+GainOrderREGISTER*jREGISTER) = *(long long *) (BB+GainOrderREGISTER*( 0x00FFFFFF & *(unsigned long
*) (BB+Patternlen+jREGISTER*3) ));
2419 jREGISTER++;
2420 *(long long *) (Strng+GainOrderREGISTER*jREGISTER) = *(long long *) (BB+GainOrderREGISTER*( 0x00FFFFFF & *(unsigned long
*) (BB+Patternlen+jREGISTER*3) ));
2421 }
2422
2423 if (Strnglen % GainOrderREGISTER)
2424 memcpy (Strng+GainOrderREGISTER*(Strnglen / GainOrderREGISTER), BB+Patternlen+3*(Strnglen / GainOrderREGISTER), Strnglen % GainOrderREGISTER);
2425

```

```

2426 // SUB SUB UGLY SLOW !!! ]
2427
2428
2429 clocks1 = clock();
2430 TotalRoughSearchTime2 = clocks1 - clocks3; TotalRoughSearchTime2++;
2431 printf( "\ndecompression to RAM without Dumping to DRIVE time: %lu clocks\n", (long)(TotalRoughSearchTime2));
2432 printf( "decompression to RAM without Dumping to DRIVE performance: %lu KB/s or %lu MB/s\n",
         (long)(Strnglen/TotalRoughSearchTime2)*1000)>>10, ((long)((Strnglen/TotalRoughSearchTime2)*1000)>>10)>>10);
2433 fwrite(Strng, 1, Strnglen, fp_outLINE);
2434 // Buffered input, just for OSHO.TXT because for max order 16 and max BBS 16,777,216 the BB pool cannot house the triads for bigger files! ]
2435
2436 clocks2 = clock();
2437 TotalRoughSearchTime = clocks2 - clocks3; TotalRoughSearchTime++;
2438 printf( "Total time: %lu clocks\n", (long)(TotalRoughSearchTime));
2439
2440 printf( "\nbenchmarking 'memcpy' by copying 197MB (OSHO.TXT size) ten times...\n");
2441 clocks3 = clock();
2442 for (i=0; i<10; i++)
2443     memcpy( Strng, BB, 197*1024*1024);
2444 clocks2 = clock();
2445 clocks2 = clock();
2446 TotalRoughSearchTime = clocks2 - clocks3; TotalRoughSearchTime++;
2447 printf( "simplicius says for 'memcpy' performance: %lu MB/s\n", 1970000/(long)(TotalRoughSearchTime));
2448 printf( "simplicius says for Decompression Ratio: %lu%%\n",
         ((long)((Strnglen/TotalRoughSearchTime2)*1000)>>10)*100)/((1970000/(long)(TotalRoughSearchTime))*1024) );
2449
2450 fclose( fp_inLINE);
2451 fclose( fp_outLINE);
2452 free( BB);
2453 free( Strng);
2454 return -1;
2455 }
2456 // DECOMPRESSOR ]
2457
2458 fseek( fp_inLINE, 0, SEEK_END);
2459 Strnglen = ftell( fp_inLINE);
2460 fseek( fp_inLINE, 0, SEEK_SET);
2461 if ( Strnglen>384*1024*1024)
2462 { printf( "Building-Blocks_DUMPER: File cannot be loaded, it exceeds 384MB limit.\n" ); return( 1 ); }
2463 fread( Strng, 1, Strnglen, fp_inLINE);
2464 fclose( fp_inLINE);
2465
2466 //printf( "Input Pattern(up to 19 chars): "); gets( Pattern); // char * _cdecl gets( char *);
2467 //printf( "%s\n", _ui64toaKAZEzerocomma( 7, l1ToaDigits, 16)+(26-2));
2468 //printf( "%s\n", _ui64toaKAZEzerocomma( Pattern[0], l1ToaDigits, 16) +(26-2));
2469 //printf( "%s\n", _ui64toaKAZEzerocomma( Pattern[1], l1ToaDigits, 16) +(26-2));
2470 //printf( "%s\n", _ui64toaKAZEzerocomma( Pattern[2], l1ToaDigits, 16) +(26-2));
2471 //exit(0);
2472 //Patternlen = strlen( &Pattern[0] );
2473
2474 // Replacing CR with NULL i.e. 13->0
2475 // for (ThunderwithL=0; ThunderwithL<Strnglen; ThunderwithL++)
2476 //     if ( Strng[ThunderwithL] == 13) Strng[ThunderwithL] = 0;
2477 //ThunderwithL=0; ThunderwithR=0;
2478 //printf( "Doing Search for Pattern(%dbytes) into String(%dbytes) line-by-line...\n", Patternlen, Strnglen);
2479
2480 //Patternlen=3; // 3..31
2481 //printf( "\nexaming BB orders 3 to 16 whether they are codeable within 3bytes i.e. 16,777,216...\n");
2482 printf( "\nenforcing BB order 7...\n");
2483 printf( "Size of incoming file or Strnglen: %d\n", Strnglen);
2484 //for( Patternlen = 3; Patternlen <=16; Patternlen++)
2485 for( Patternlen = 7; Patternlen <=7; Patternlen++)
2486 { // MAIN CYCLE [
2487
2488     memcpy( &BuildingBlocksBox[2], _ui64toaKAZEzerocomma( Patternlen, l1ToaDigits, 10) +(26-3), 3);
2489
2490     printf( "\nSorting %lu Pointers to Building-Blocks %lu chars in size...\n", Strnglen-Patternlen+1, Patternlen);
2491
2492     nlines = Strnglen-Patternlen+1;
2493     ASbackp = (string *) malloc( nlines*sizeof(string) ); // sizeof(string) is 4
2494     if( ASbackp == NULL )
2495     { puts( "Building-Blocks_DUMPER: Needed memory allocation denied!\n" ); return( 1 ); }
2496     printf( "Allocated memory for pointers-to-words in MB: %lu\n", ((nlines*sizeof(string))>>20)+1 );
2497
2498
2499     // ASSIGNING POINTERS ... [
2500     cur = Strng;
2501     for( j = 0; j < nlines; j++ )
2502     { ASbackp[j] = cur;
2503       //printf( "%d %d %d\n", *(ASbackp[j]+0), *(ASbackp[j]+1), *(ASbackp[j]+2));
2504       cur = cur + 1;
2505     }
2506     // ASSIGNING POINTERS ... ]
2507
2508
2509
2510 // SORTING POINTERS ... [
2511 #if defined( Quicks)

```

```

2512 //void mkqsort_main(unsigned char **a, int n) { mkqsort(a, n, 0); }
2513 //mkqsort_main(ASbackp, nlines); // Buggy!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
2514 qsortVS2003(ASbackp, nlines, 4, compVS2003);
2515
2516 #else
2517 //simplsortFIXEDLength(string a[], int n, int b, int FIXEDLength)
2518 simplsortFIXEDLength(ASbackp, nlines, 0, Patternlen);
2519 #endif
2520 // SORTING POINTERS ... ]
2521
2522
2523 // WRITING SORTED ... [
2524     if ( ( fp_inLINE = fopen( &BuildingBlocksBox[0], "wb+" ) ) == NULL )
2525     { printf( "Building-Blocks_DUMPER: Can't create file %s\n", &BuildingBlocksBox[0] ); return( 1 ); }
2526     printf( "Writing Sorted Building-Blocks to %s...\n", &BuildingBlocksBox[0]);
2527     goto SORTED_witout_DUPLICATES;
2528
2529     for( j = 0; j < nlines; j++ )
2530     { //Slot = kuxHash3plus(ASbackp[j]);
2531       //printf( fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma( Slot, l1ToaDigits, 10)+(26-5));
2532       //printf( fp_inLINE, "%s\n", ASbackp[j]);
2533       for ( i=0; i<Patternlen; i++)
2534         fprintf( fp_inLINE, "%c", *(ASbackp[j]+i));
2535       //printf( fp_inLINE, "%s", _ui64toaKAZEzerocomma( *(ASbackp[j]+i), l1ToaDigits, 16) +(26-2));
2536       fprintf( fp_inLINE, "\n" );
2537       // fwrite( CRdLfa, 2, 1, fp_out );
2538       //if ( strlen( backp[j] ) >=3 ) {
2539         //   fprintf( fp_out2, "%s", backp[j]);
2540         //   fwrite( CRdLfa, 2, 1, fp_out2 );
2541       //}
2542     }
2543 // WRITING SORTED ... ]
2544 SORTED_witout_DUPLICATES:
2545
2546 // REMOVING DUPLICATES ... [
2547     linecounterNotRL = nlines;
2548     nlines1 = linecounterNotRL;
2549     linecounterRL = 0;
2550
2551     jFirst = 0;
2552
2553     if ( linecounterNotRL == 1 ) {
2554         //printf( fp_inLINE, "%s\n", ASbackp[jFirst]);
2555     }
2556 #if defined( DumbodUMP)
2557     for ( i=0; i<Patternlen; i++)
2558         fprintf( fp_inLINE, "%c", *(ASbackp[jFirst+i]);
2559     //printf( fp_inLINE, "\n" );
2560 #endif
2561     linecounterRL++;
2562
2563     jPrevLine = jFirst;
2564     linecounterNotRL--;
2565     for( j = jFirst + 1; j < nlines1; j++ )
2566     {
2567         if ( linecounterNotRL != 0 ) {
2568             linecounterNotRL--;
2569
2570             //if ( strcmpKAZE( ASbackp[jPrevLine], ASbackp[j] ) != 0 ) {
2571             //if ( memcmpKAZE( ASbackp[jPrevLine], ASbackp[j], Patternlen ) != 0 ) {
2572                 //printf( fp_inLINE, "%s\n", ASbackp[jPrevLine]);
2573             #if defined( DumbodUMP)
2574                 for ( i=0; i<Patternlen; i++)
2575                     fprintf( fp_inLINE, "%c", *(ASbackp[jPrevLine+i]);
2576                 //printf( fp_inLINE, "\n" );
2577             #endif
2578             linecounterRL++;
2579
2580             if ( linecounterNotRL == 0 ) {
2581                 //printf( fp_inLINE, "%s\n", ASbackp[j]);
2582             #if defined( DumbodUMP)
2583                 for ( i=0; i<Patternlen; i++)
2584                     fprintf( fp_inLINE, "%c", *(ASbackp[j+i]);
2585                 //printf( fp_inLINE, "\n" );
2586             #endif
2587             linecounterRL++;
2588
2589             }PrevLine = j;
2590         }
2591     }
2592 // REMOVING DUPLICATES ... ]
2593
2594     printf( "Patternlen: %d | %d+1 | linecounterRL: %d | %d\n", Patternlen, Strnglen, Patternlen, linecounterRL, Strnglen-Patternlen+1-linecounterRL);
2595     if ( linecounterRL>256*256*256 ) { //3 bytes encode 2^24=256*256*256 B-Blocks.
2596         printf( "\nBuilding-Blocks_DUMPER: Overflow! BBS exceed the 16,777,216 limit.\n");
2597         free( ASbackp);
2598         return -1;
2599     }

```

```

2600 }
2601 printf("Gain(reduced-size) or Strnglen-(Patternlen*linecounterRL+3*(Strnglen/Patternlen)+(Strnglen%Patternlen)): %d\n",Strnglen-
(Patternlen*linecounterRL+3*(Strnglen/Patternlen)+(Strnglen%Patternlen));
2602
2603 if ( (signed long)(Strnglen-(Patternlen*linecounterRL+3*(Strnglen/Patternlen)+(Strnglen%Patternlen)))>GainMax) {GainMax=(signed
long)(Strnglen-(Patternlen*linecounterRL+3*(Strnglen/Patternlen)+(Strnglen%Patternlen)); GainOrder=Patternlen;}
2604
2605 free(ASBackup);
2606 fclose(fp_inLINE);
2607 } // MAIN CYCLE
2608 //clocks2 = clock(); TotalRoughSearchTime = clocks2 - clocks1; TotalRoughSearchTime++;
2609 //printf("Railgun_hits/Railgun_clocks: %lu/%lu\n", Railgunhits>4, (long)(TotalRoughSearchTime)>>4);
2610 //printf("\nBuilding-Blocks_DUMPER total time: %lu clocks\n", (long)(TotalRoughSearchTime));
2611 printf("\nMaximum compression (or minimum expansion) for order: %lu\n", GainOrder);
2612
2613 clocks3 = clock();
2614 // [ SS file creating:
2615
2616 bufEXTENSION = (unsigned char *) malloc(sizeof (unsigned char) * (strlen(argv[1])+1+3)); //when creating .SS
2617
2618 if(bufEXTENSION == NULL)
2619 { printf("Building-Blocks_DUMPER: Can't allocate memory.\n"); return(1); }
2620
2621 strcpy(bufEXTENSION, argv[1]);
2622 strcat(bufEXTENSION, ".SS");
2623
2624 printf("\nCreating %s ... \n", bufEXTENSION);
2625 if( ( fp_outLINE = fopen(bufEXTENSION, "wb") ) == NULL )
2626 { printf("Building-Blocks_DUMPER: Can't open file %s \n", bufEXTENSION ); return(1); }
2627
2628 fprintf(fp_outLINE, "[Simplicius_Simplicissimus_revison_1]\n"); // 0x1A = 026 ASCII CODE TEXT EOF
2629 fprintf(fp_outLINE, "The file format after the ASCII code 026: \n1 byte for BB_Order, \n4bytes for Original_Size, \n4bytes for BB length,
\nBB itself, \n3*(Original_Size/BB_Order) bytes of indexes followed by the (Original_Size%BB_Order) bytes of literals.%c",0x1A); // 0x1A =
026 ASCII CODE TEXT EOF
2630
2631 // Uploading BB file to BB pool:
2632 memcpy(&BuildingBlocksBox[2],_ui64toakAZEzerocomma (GainOrder, 1)toabigits, 10) +(26-3),3);
2633 if( ( fp_inLINE = fopen (&BuildingBlocksBox[0], "rb") ) == NULL )
2634 { printf("Building-Blocks_DUMPER: Can't upload file %s\n", &BuildingBlocksBox[0] ); return(1); }
2635
2636 fseek(fp_inLINE, 0, SEEK_END);
2637 Patternlen = ftell(fp_inLINE);
2638 fseek(fp_inLINE, 0, SEEK_SET);
2639
2640 BB = (char *)malloc( 768*1024*1024 );
2641 if( BB == NULL )
2642 { puts("Building-Blocks_DUMPER: Needed memory allocation denied!\n"); return(1); }
2643
2644 if (Patternlen>768*1024*1024)
2645 { printf("Building-Blocks_DUMPER: File cannot be loaded, it exceeds 768MB limit.\n"); return(1); }
2646
2647 fread(BB, 1, Patternlen, fp_inLINE);
2648 fclose(fp_inLINE);
2649
2650 putc*((unsigned char *)&GainOrder+0), fp_outLINE);
2651
2652 putc*((unsigned char *)&strnglen+0), fp_outLINE);
2653 putc*((unsigned char *)&strnglen+1), fp_outLINE);
2654 putc*((unsigned char *)&strnglen+2), fp_outLINE);
2655 putc*((unsigned char *)&strnglen+3), fp_outLINE);
2656
2657 putc*((unsigned char *)&Patternlen+0), fp_outLINE);
2658 putc*((unsigned char *)&Patternlen+1), fp_outLINE);
2659 putc*((unsigned char *)&Patternlen+2), fp_outLINE);
2660 putc*((unsigned char *)&Patternlen+3), fp_outLINE);
2661
2662 fwrite(BB, 1, Patternlen, fp_outLINE);
2663
2664 // Must dump triads and the eventual remainder/literals.
2665 // 'BB' pool with length 'Patternlen' has 'Patternlen/GainOrder' BBs.
2666 // 'Strng' pool with length 'Strnglen' must be traversed with step 'GainOrder' assigning indexes/triads to BBs ...
2667
2668 // [ Triads dumping
2669
2670 // Maximum slow search:
2671 //Creating googbooks-eng-us-all-4gram-20090715-graffith_K_distinct.txt.SS ...
2672 //Building-Blocks_DUMPER .SS dumping time: 1538673 clocks
2673 //Building-Blocks_DUMPER total time: 1685829 clocks
2674 goto NextNextInLine;
2675 for (ThunderwithR=0; ThunderwithR<Strnglen/GainOrder; ThunderwithR++)
2676 for (ThunderwithL=0; ThunderwithL<Patternlen/GainOrder; ThunderwithL++)
2677 {
2678 if (memcmp(&BB[ThunderwithL*GainOrder+0],&Strng[ThunderwithR*GainOrder+0],GainOrder) == 0) {
2679 putc*((unsigned char *)&ThunderwithL+0), fp_outLINE);
2680 putc*((unsigned char *)&ThunderwithL+1), fp_outLINE);
2681 putc*((unsigned char *)&ThunderwithL+2), fp_outLINE);
2682 // Notice that only 3 lower Little-Endian bytes are dumped!
2683 break;

```

```

2684 }
2685 }
2686 goto LastLine;
2687 NextInLine:
2688 // Brute-force slow search: Railgun_Quaduplet
2689 //Creating googbooks-eng-us-all-4gram-20090715-graffith_K_distinct.txt.SS ...
2690 //Building-Blocks_DUMPER .SS dumping time: 573392 clocks
2691 //Building-Blocks_DUMPER total time: 700579 clocks
2692 for (ThunderwithR=0; ThunderwithR<Strnglen/GainOrder; ThunderwithR++)
2693 {
2694 FoundInPTR=BB;
2695 while ( FoundInPTRNEW = Railgun_Quaduplet(FoundInPTR,&Strng[ThunderwithR*GainOrder+0],Patternlen,GainOrder) ) {
2696 if ( (FoundInPTRNEW-BB) % GainOrder == 0 ) {
2697 ThunderwithL = (unsigned long)(FoundInPTRNEW-BB) / GainOrder;
2698 putc*((unsigned char *)&ThunderwithL+0), fp_outLINE);
2699 putc*((unsigned char *)&ThunderwithL+1), fp_outLINE);
2700 putc*((unsigned char *)&ThunderwithL+2), fp_outLINE);
2701 // Notice that only 3 lower Little-Endian bytes are dumped!
2702 break;
2703 }
2704 FoundInPTR = FoundInPTRNEW+1;
2705 }
2706 }
2707 goto LastLine;
2708 NextNextInLine:
2709 // Binary-Search:
2710 //Creating googbooks-eng-us-all-4gram-20090715-graffith_K_distinct.txt.SS ...
2711 //Building-Blocks_DUMPER .SS dumping time: 1923 clocks
2712 //Building-Blocks_DUMPER total time: 132563 clocks
2713 for (ThunderwithR=0; ThunderwithR<Strnglen/GainOrder; ThunderwithR++)
2714 {
2715 YabbaYabba = 0;
2716 LeftPointBS = 0;
2717 RightPointBS = Patternlen/GainOrder - 1;
2718 PivotPointBS = 0;
2719 do {
2720 if (RightPointBS - LeftPointBS <= 1) {
2721 Nataliakills = memcmp(&Strng[ThunderwithR*GainOrder+0],&BB[LeftPointBS*GainOrder+0],GainOrder);
2722 if ( Nataliakills == 0) { YabbaYabba = 1; PivotPointBS = LeftPointBS; break; }
2723 Nataliakills = memcmp(&Strng[ThunderwithR*GainOrder+0],&BB[RightPointBS*GainOrder+0],GainOrder);
2724 if ( Nataliakills == 0) { YabbaYabba = 1; PivotPointBS = RightPointBS; break; }
2725 break; }
2726 PivotPointBS = (LeftPointBS + RightPointBS) >> 1;
2727 Nataliakills = memcmp(&Strng[ThunderwithR*GainOrder+0],&BB[PivotPointBS*GainOrder+0],GainOrder);
2728 if ( Nataliakills == 0) { YabbaYabba = 1; break; }
2729 else if ( Nataliakills < 0) { RightPointBS = PivotPointBS; }
2730 else { LeftPointBS = PivotPointBS; }
2731 } while (1 == 1);
2732 if ( YabbaYabba != 0 ) {
2733 ThunderwithL = PivotPointBS;
2734 putc*((unsigned char *)&ThunderwithL+0), fp_outLINE);
2735 putc*((unsigned char *)&ThunderwithL+1), fp_outLINE);
2736 putc*((unsigned char *)&ThunderwithL+2), fp_outLINE);
2737 // Notice that only 3 lower Little-Endian bytes are dumped!
2738 }
2739 }
2740
2741 /* Binary search from 'LOGerINFINITY_COUNTRY_CITY.php':
2742 $WRDpresent = 0;
2743 $LeftPoint = 0;
2744 $RightPoint = $linecncr - 1;
2745 $q = 0;
2746 do {
2747 if ($RightPoint - $LeftPoint <= 1) {
2748 $SplitIsSTRINGlefta = (float) $array77a[$LeftPoint];
2749 $SplitIsSTRINGleftb = (float) $array77b[$LeftPoint];
2750 $SplitIsSTRINGrighta = (float) $array77a[$RightPoint];
2751 $SplitIsSTRINGrightb = (float) $array77b[$RightPoint];
2752 if ($long >= $SplitIsSTRINGlefta && $long <= $SplitIsSTRINGleftb) { $WRDpresent = 1; $q = $LeftPoint; break; }
2753 if ($long >= $SplitIsSTRINGrighta && $long <= $SplitIsSTRINGrightb) { $WRDpresent = 1; $q = $RightPoint; break; }
2754 break; }
2755 $q = ($LeftPoint + $RightPoint) >> 1;
2756 $SplitIsSTRINGa = (float) $array77a[$q];
2757 $SplitIsSTRINGb = (float) $array77b[$q];
2758 if (($long >= $SplitIsSTRINGa) && ($long <= $SplitIsSTRINGb)) { $WRDpresent = 1; break; }
2759 if ($long > $SplitIsSTRINGb) { $LeftPoint = $q; }
2760 if ($long < $SplitIsSTRINGa) { $RightPoint = $q; }
2761 } while (1 == 1);
2762 */
2763 LastLine:
2764
2765 // [ Triads dumping
2766
2767 // [ Literals dumping
2768 ThunderwithL=Strnglen % GainOrder;
2769 while (ThunderwithL) {
2770 putc*((unsigned char *)Strng+Strnglen-ThunderwithL ), fp_outLINE);

```

```

2771 ThunderwithL--;
2772 }
2773 // ] Literals dumping
2774
2775 // ] ss file creating:
2776
2777 clocks2 = clock();
2778
2779 TotalRoughSearchTime = clocks2 - clocks3; TotalRoughSearchTime++;
2780 printf( "\nBuilding-Blocks_DUMPER .SS dumping time: %lu clocks\n", (long)(TotalRoughSearchTime));
2781
2782 TotalRoughSearchTime = clocks2 - clocks1; TotalRoughSearchTime++;
2783 printf( "Building-Blocks_DUMPER total time: %lu clocks\n", (long)(TotalRoughSearchTime));
2784
2785 /*
2786 */
2787 printf( "Searching for Pattern(%dbytes) into String(%dbytes) at each position ... \n", Patternlen, Strnglen);
2788 if( ( fp_inLINE = fopen( &BuildingBlocksBox[0], "wb+" ) ) == NULL )
2789 { printf( "Building-Blocks_DUMPER: Can't create file %s\n", &BuildingBlocksBox[0] ); return( 1 ); }
2790 printf( "writing Building-Blocks with corresponding hits(overlapped) to %s ... \n", &BuildingBlocksBox[0]);
2791
2792 Railgunhits=0;
2793 clocks3 = clock();
2794 for ( i=0;i<=Strnglen-Patternlen; i++)
2795 {
2796 // printf( "Searching for " );
2797 // for ( j=0;j<Patternlen; j++)
2798 // printf( "%s", _ui64toakAZEzerocomma (Strng[0+i+j], l1ToaDigits, 16) +(26-2));
2799 // printf( " ... \n");
2800 printf( "Remain: %lu \r", Strnglen-Patternlen - i + 1);
2801 FoundInPTR = Railgun_totalhits(&Strng[0], &Strng[0+i], Strnglen, Patternlen);
2802 for ( j=0;j<Patternlen; j++)
2803 fprintf(fp_inLINE, "%s", _ui64toakAZEzerocomma (Strng[0+i+j], l1ToaDigits, 16) +(26-2));
2804 fprintf(fp_inLINE, " %d\n", Railgunhits);
2805 Railgunhits=0;
2806 }
2807 clocks2 = clock(); TotalRoughSearchTime = clocks2 - clocks1; TotalRoughSearchTime++;
2808 //printf( "Railgun_hits/Railgun_clocks: %lu/%lu\n", Railgunhits>>4, (long)(TotalRoughSearchTime)>>4);
2809 printf( "\nBuilding-Blocks_DUMPER total time for creating %s: %lu clocks\n", &BuildingBlocksBox[0], (long)(TotalRoughSearchTime));
2810 */
2811
2812
2813 free(BB);
2814 free(bufEXTENSION);
2815 free(Strng);
2816 return(0);
2817 }
2818
2819 /*
2820 Order 4:
2821 206,908,949 OSHO.TXT
2822 156,174,067 OSHO.TXT.SS
2823 Sorting 206908946 Pointers to Building-Blocks 4 chars in size ...
2824 Allocated memory for pointers-to-words in MB: 790
2825 Writing Sorted Building-Blocks to BB004.txt ...
2826 Patternlen:4|206908949-4+1|linecounterRL:248019|206660927
2827 Gain(reduced-size) or Strnglen-(Patternlen*linecounterRL+3*(Strnglen/Patternlen)+(Strnglen*Patternlen)): 50735161
2828 BBS pool size: 992076
2829 Decompression to RAM without Dumping to DRIVE time: 1564 clocks
2830 Decompression to RAM without Dumping to DRIVE performance: 129194 KB/s
2831
2832 Order 7:
2833 206,908,949 OSHO.TXT
2834 122,297,608 OSHO.TXT.SS
2835 Sorting 206908943 Pointers to Building-Blocks 7 chars in size ...
2836 Allocated memory for pointers-to-words in MB: 790
2837 Writing Sorted Building-Blocks to BB007.txt ...
2838 Patternlen:7|206908949-7+1|linecounterRL:4803152|202105791
2839 Gain(reduced-size) or Strnglen-(Patternlen*linecounterRL+3*(Strnglen/Patternlen)+(Strnglen*Patternlen)): 84611620
2840 BBS pool size: 33622064
2841 Decompression to RAM without Dumping to DRIVE time: 3282 clocks
2842 Decompression to RAM without Dumping to DRIVE performance: 61565 KB/s
2843
2844 Order 8:
2845 206,908,949 OSHO.TXT
2846 149,243,106 OSHO.TXT.SS
2847 Sorting 206908942 pointers to building-Blocks 8 chars in size ...
2848 Allocated memory for pointers-to-words in MB: 790
2849 Writing Sorted Building-Blocks to BB008.txt ...
2850 Patternlen:8|206908949-8+1|linecounterRL:8956496|197952446
2851 Gain(reduced-size) or Strnglen-(Patternlen*linecounterRL+3*(Strnglen/Patternlen)+(Strnglen*Patternlen)): 57666122
2852 BBS pool size: 71651968
2853 Decompression to RAM without Dumping to DRIVE time: 3001 clocks
2854 Decompression to RAM without Dumping to DRIVE performance: 67330 KB/s
2855 */

```