```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void QuickSort(void);                       //  Median-of-three Quicksort

unsigned long Nbr2Sort = 100000000;          //  Number of items to sort.
unsigned char RandNbrs[100000001];           //  Must be + 1 because of using [0] as a sentinel.

unsigned long QSleft[64];                   //  Stack arrays for the left/right
unsigned long QSright[64];                  //  ends of subgroups within QuickSort()

long Mainclocks1, Mainclocks2;
unsigned long MinGroup_GLOBAL, TestChunk;

int strcmpKAZE (
        const char * src,
        const char * dst
        )
{
        int ret = 0 ;

        while( ! (ret = *(unsigned char *)src - *(unsigned char *)dst) && *dst)
                ++src, ++dst;

        if ( ret < 0 )
                ret = -1 ;
        else if ( ret > 0 )
                ret = 1 ;

        return( ret );
}

int main(int argc, char *argv[]) {

   size_t size;
   FILE *in_file;
   char *pointerflushD;
   unsigned long    j;

   char file_name[67] = "pi100m.txt";
   char file_name_unsorted[67] = "Bastumante_unsorted_lX.txt";
   char file_name_sorted[67] = "Bastumante_sorted_lX.txt";
   FILE *fp_out;
   char CRdLFa[2];  // 0..1, 0 = 13, 1 = 10
   CRdLFa[0] = 13; CRdLFa[1] = 10;

  puts( "Bastumante(a quicksort benchmark tool), revision 3, written by Svalqyatchx," );
  puts( "in fact adapted from Bill Butler Durango's source, thanks." );
  puts("");
  puts("Usage: Bastumante [dump]");
  puts("Note1: It sorts 100 millions bytes(in fact first 100 millions digits of Pi).");
  puts("Note2: This program demonstrates the performance of Quicksort algorithm.");
  puts("Note3: Option 'dump' dumps sorted data to file Bastumante_sorted_lX.txt.");
  puts("");
  printf("Me machine 'CPU-Z version 1.29' report:\n");
  printf("\n");
  printf("Number of CPUs:       1\n");
  printf("Name:                 AMD Athlon XP\n");
  printf("Code Name:            Barton\n");
  printf("Specification:        AMD Athlon(tm) XP 2600+\n");
  printf("Technology:           0.13 microns\n");
  printf("CPU Clock Speed:      1919.6 MHz\n");
  printf("Clock multiplier:     x 11.5\n");
```

```
printf("FSB Frequency:        166.9 MHz\n");
printf("Bus Speed:            333.8 MHz\n");
printf("L1 Data Cache:        64 KBytes, 2-way set associative, 64 Bytes line size\n");
printf("L1 Instruction Cache: 64 KBytes, 2-way set associative, 64 Bytes line size\n");
printf("L2 Cache:             512 KBytes, 16-way set associative, 64 Bytes line size\n");
printf("L2 Speed:             1919.6 MHz (Full)\n");
printf("L2 Location:          On Chip\n");
printf("\n");
printf("Motherboard manufacturer: Gigabyte Technology Co., Ltd.\n");
printf("Motherboard model:        7VT880-RZ, x.x\n");
printf("Chipset:                  VIA KT880 rev. 00\n");
printf("\n");
printf("# of memory modules: 2\n");
printf("Module 0:            Kingston DDR-SDRAM PC3200 - 512 MBytes\n");
printf("Module 1:            Kingston DDR-SDRAM PC3200 - 512 MBytes\n");
printf("\n");
printf("Software: Microsoft Windows XP Professional Service Pack 2\n");
printf("\n");
printf("Me machine 'EVEREST v4.20.1170' report:\n");
printf("\n");
printf("      Memory Controller:\n");
printf("      Type:            Dual Channel (128-bit)\n");
printf("      Active Mode:     Dual Channel (128-bit)\n");
printf("      Bank Interleave: Disabled\n");
printf("\n");
printf("      Front Side Bus Properties:\n");
printf("      Bus Type:        DEC Alpha EV6\n");
printf("      Bus Width:       64-bit\n");
printf("      Real Clock:      167 MHz (DDR)\n");
printf("      Effective Clock: 333 MHz\n");
printf("\n");
printf("      Memory Bus Properties:\n");
printf("      Bus Type:        Dual DDR SDRAM\n");
printf("      Bus Width:       128-bit\n");
printf("      DRAM:FSB Ratio:  6:5\n");
printf("      Real Clock:      200 MHz (DDR)\n");
printf("      Effective Clock: 400 MHz\n");
printf("\n");
printf("      Memory Timings:\n");
printf("      CAS Latency (CL)               3T\n");
printf("      RAS To CAS Delay (tRCD)        5T\n");
printf("      RAS Precharge (tRP)            5T\n");
printf("      RAS Active Time (tRAS)         9T\n");
printf("      Row Refresh Cycle Time (tRFC) 15T\n");
printf("      Command Rate (CR)              2T\n");
printf("      Write Recovery Time (tWR)      3T\n");
printf("      Write To Read Delay (tWTR)     2T\n");
printf("      Refresh Period (tREF)       1632T\n");
printf("\n");
printf("      Memory Benchmark:\n");
printf("      Read       2316 MB/s\n");
printf("      Write      2647 MB/s\n");
printf("      Copy       2445 MB/s\n");
printf("      Latency     128.6 ns\n");
printf("\n");
printf("Me machine results:\n\n");
printf("MinGroup =      16*1byte, gives 8984 clocks.\n");
printf("MinGroup =      64*1byte, gives 8469 clocks.\n");
printf("MinGroup =     256*1byte, gives 7954 clocks.\n");
printf("MinGroup =    1024*1byte, gives 7390 clocks.\n");
printf("MinGroup =    4096*1byte, gives 6937 clocks.\n");
printf("MinGroup =   16384*1byte, gives 6719 clocks.\n");
printf("MinGroup =   65536*1byte, gives 11500 clocks.\n");
printf("MinGroup =  262144*1byte, gives 73781 clocks.\n");
```

```
  printf("MinGroup = 1048576*1byte, gives 1249734 clocks.\n");
  puts("");
  printf("Me machine total time in Windows XP: 1384 seconds.\n");
  printf("Compiled with: cl /Ox Bastumante3.c /FaBastumante3.asm\n");
  printf("Compiler used: Microsoft 32-bit C/C++ Optimizing Compiler Version 13.10.3077\n");
  puts("");
  printf("MinGroup =      16*1byte, gives 8480000 clocks.\n");
  printf("MinGroup =      64*1byte, gives 7920000 clocks.\n");
  printf("MinGroup =     256*1byte, gives 7410000 clocks.\n");
  printf("MinGroup =    1024*1byte, gives 6960000 clocks.\n");
  printf("MinGroup =    4096*1byte, gives 6480000 clocks.\n");
  printf("MinGroup =   16384*1byte, gives 6280000 clocks.\n");
  printf("MinGroup =   65536*1byte, gives 10140000 clocks.\n");
  printf("MinGroup =  262144*1byte, gives 61230000 clocks.\n");
  printf("MinGroup = 1048576*1byte, gives 1020220000 clocks.\n");
  puts("");
  printf("Me machine total time in Linux version 2.6.26.5(RIPLinux 6.8): 1137 seconds.\n");
  printf("Compiled with: gcc -static -O3 Bastumante3.c -o Bastumante3\n");
  printf("Compiler used: gcc version 4.2.3\n");
  puts("");

          // pointerflushD = (char *)malloc( 100000000 );
          // if( pointerflushD == NULL )
          // { puts( "Bastumante: Needed memory allocation denied!\n" ); return( 1 ); }
          // printf( "Allocated memory for data in MB: %lu\n", (100000000>>20) );

// To put data into disk cache ...
   if (!(in_file = fopen(file_name, "rb"))) {
      printf( "Bastumante: Can't open file %s \n", file_name );
      exit(-1);
   }
   fseek(in_file, 0, SEEK_END);
   size = ftell(in_file);
   fseek(in_file, 0, SEEK_SET);
   //printf( "Uploading sort data ...\n" );
   if (fread(&RandNbrs[1], 1, 100000000, in_file) < size) {
      printf( "Bastumante: Can't read file %s \n", file_name );
      exit(-1);
   }
   fclose(in_file);


/*
      printf("\nFlushing unsorted LONGs(32bit-Hexadecimals) ...\n");
      if( ( fp_out = fopen( file_name_unsorted, "wb+" ) ) == NULL )
      { printf( "Bastumante: Can't create file %s \n", file_name_unsorted ); exit( 1 ); }
      for( j = 0+1; j < (Nbr2Sort+1); j++ )
      { fprintf(fp_out, "%lX", RandNbrs[j]);
        fwrite(CRdLFa, 2, 1, fp_out );
      }
      if ( fclose(fp_out) != 0 )
      {
        fprintf(stderr, "Error closing output file.\n");
        return -2;
      }
*/

Mainclocks1 = clock();
for (TestChunk=1; TestChunk<=4*4*4*1024; TestChunk=TestChunk*4) {

   if (!(in_file = fopen(file_name, "rb"))) {
      printf( "Bastumante: Can't open file %s \n", file_name );
      exit(-1);
   }
```

```
   fseek(in_file, 0, SEEK_END);
   size = ftell(in_file);
   fseek(in_file, 0, SEEK_SET);
   printf( "Uploading sort data ...\n" );
   if (fread(&RandNbrs[1], 1, 100000000, in_file) < size) {
      printf( "Bastumante: Can't read file %s \n", file_name );
      exit(-1);
   }
   fclose(in_file);

  MinGroup_GLOBAL = 16*TestChunk;
  printf("MinGroup = %7d*1byte, that is Insertionsort-final-chunk elements.\n", MinGroup_GLOBAL);
  QuickSort();
}
  Mainclocks2 = clock();
  printf("\nTotal sort time was %d clocks.\n", Mainclocks2 - Mainclocks1);


if ( argc == 2 ) // +1 for program name
{
 if (strcmpKAZE(argv[1],"dump") == 0 || strcmpKAZE(argv[1],"DUMP") == 0)
   {

      printf("\nFlushing sorted BYTEs ...\n");
      if( ( fp_out = fopen( file_name_sorted, "wb+" ) ) == NULL )
      { printf( "Bastumante: Can't create file %s \n", file_name_sorted ); exit( 1 ); }
      for( j = 0+1; j < (Nbr2Sort+1); j++ )
      { fprintf(fp_out, "%c", RandNbrs[j]);
        //fwrite(CRdLFa, 2, 1, fp_out );
      }
      if ( fclose(fp_out) != 0 )
      {
        fprintf(stderr, "Error closing output file.\n");
        return -2;
      }
   }
}

  return(0);
}


//*****************************************************************************
//
//                                  QuickSort
//
//  QuickSort has long had a reputation for being the fastest general purpose
//  sort algorithm. It is also perhaps the most difficult to code, and is
//  subject to sharply adverse execution time if the "pivot values" are picked
//  poorly - which can happen if the data to be sorted is already partially
//  sorted.
//
//  The algorithm works by picking one of the elements to be sorted as a "pivot
//  value". The list of items to be sorted is then partitioned so that all
//  elements that have a value less than the pivot end up in the front portion
//  of the array while all elements that are greater than the pivot value end
//  up in the other end. (Elements that are equal to the pivot could end up in
//  either section.) After the first pass, the array of items to be sorted
//  looks like:
//
//                                   Pivot
//            Low elements are here  Item    Higher elements are here
//           ------------------------------------------------------
//  RandNbrs |    |    |    |    |    |    |    |    |    |    |    |
```

```
//           ---------------------------------------------------------
//             1    2    3    4                                    Nbr2Sort
//
//  After round 1, the QuickSort process is applied to both of the 2 subgroups.
//  Whichever subgroup was smaller is processed immediately while the location
//  of the left and right ends of the larger subgroup are placed on a stack
//  for later processing. This processing order will guarantee that the stack
//  will never exceed Log2(Nbr2Sort) items.
//
//  The repetitive processing of subgroups continues until the size of a
//  subgroup falls below a size defined by "MinGroup". Once a subgroup is
//  smaller than this, it is not sorted further by QuickSort. Small groups can
//  be processed faster by Insertion Sort. When Quicksort has reduced all
//  subgroups to < "MinGroup" size, control passes to "Insertion Sort" for a
//  final pass through the entire array.
//
//  In the "old days", the optimal size for "MinGroup" was about 18. The cache
//  memory on current processor chips reduces the time to access anything in
//  the cache - which includes the part of the array that is currently residing
//  in the cache. This greatly increases the efficiency of the final "Insertion
//  Sort" relative to the quicksort portion. Thus, significantly larger values
//  for "MinGroup" work better when a cache is being used. (You can experiment
//  with the value that is assigned to "MinGroup".)
//
//  Selection of the "pivot value" is crucial to the efficiency of Quicksort.
//  If the pivot value is selected so that it evenly partitions a subgroup,
//  then Quicksort is very efficient. On the other hand, if the value of the
//  "Pivot item" is near either the lowest or highest values that are going to
//  be partitioned within any subgroup, that particular round of Quicksort will
//  not do its job of quickly spliting the subgroups into ever smaller sizes.
//
//  The "median of three" portion of the routine is an effort to pick a good
//  "pivot value". If a "pivot value" can be picked so that it exactly splits a
//  subgroup into 2 equal portions, then Quicksort will be as efficient as
//  possible. An effort is made to do this by trying to find a value which is
//  close to the median of the subgroup. This is done by checking the values at
//  the second, last, and middle positions within a subgroup. The middle value
//  of these three is used as the "pivot value" while the two extremes are
//  placed at the two ends of the subgroup.
//
//  The code given here is based on a flyer that Robert Sedgewick (author of
//  "Algorithms") handed out "a few years ago" during a 2-semester sequence of
//  "Analysis of Algorithms". (Professor Sedgewick is 2nd from the left in the
//  center photo at http://groups.yahoo.com/group/CSAtrium/)
//
//*****************************************************************************

void QuickSort(void) {

  unsigned long    i, j, k, StackPtr;
  unsigned long    LeftEnd, RightEnd, LeftPtr, RightPtr, MidPtr, MinGroup;
  unsigned char Pvalue, temp;
  long clocks1, clocks2;

  puts("Quicksort started ...");

  clocks1 = clock();

  RandNbrs[0] = 0;                          //  Sentinel for sort - used
                                            //  by the Insertion Sort
                                            //  portion.

    //  Initialize left end, right end, stack pointer,
    //  and minimum size for subgroups.
```

```
LeftEnd = 1;                              //  For the first round, the 2
RightEnd = Nbr2Sort;                      //  ends will be the whole array
MinGroup = MinGroup_GLOBAL;               //  Years ago this would be ~18

if (Nbr2Sort > MinGroup)                  //  Run quicksort until no
  StackPtr = 1;                           //  subgroup remains larger
else StackPtr = 0;                        //  than "MinGroup" elements.


  //  Start quicksort. First, set the pivot value equal to the median of the
  //  array values at RandNbrs[LeftEnd+1], RandNbrs[(LeftEnd+RightEnd)/2],
  //  and RandNbrs[RightEnd]. The minimum of these 3 is placed at
  //  RandNbrs[LeftEnd+1] while the maximum is placed at RandNbrs[RightEnd].
  //  The value at RandNbrs[LeftEnd] is moved to
  //  RandNbrs[(LeftEnd+RightEnd)/2].

while (StackPtr) {                        //  Loop until all subgroups
                                          //  are partitioned down to
                                          //  <= "MinGroup" size.
  LeftPtr = LeftEnd + 1;                  //  Ptr to left end.
  RightPtr = RightEnd;                    //  Ptr to right end.
  MidPtr = (LeftEnd + RightEnd)/2;        //  Point to middle

                                          //  Start sort of these 3
  if (RandNbrs[LeftPtr] > RandNbrs[RightPtr]) {
    temp = RandNbrs[LeftPtr];             //  elements
    RandNbrs[LeftPtr] = RandNbrs[RightPtr];
    RandNbrs[RightPtr] = temp;
  }

  if (RandNbrs[MidPtr] > RandNbrs[RightPtr]) {
    Pvalue = RandNbrs[RightPtr];
    RandNbrs[RightPtr] = RandNbrs[MidPtr];
  }
  else if (RandNbrs[MidPtr] < RandNbrs[LeftPtr]) {
    Pvalue = RandNbrs[LeftPtr];
    RandNbrs[LeftPtr] = RandNbrs[MidPtr];
  }
  else Pvalue = RandNbrs[MidPtr];
                                          //  The 3 values are sorted and
                                          //  and the median is in Pvalue
  RandNbrs[MidPtr] = RandNbrs[LeftEnd];   //  Fill in hole with LeftEnd

  //  Start the main loop. Move pointers inward until
  //  we find 2 elements that have to be exchanged.

  while (RandNbrs[++LeftPtr] < Pvalue);   //  Set up pointers
  while (RandNbrs[--RightPtr] > Pvalue);  //  for 1st exchange
  while (LeftPtr < RightPtr) {            //  Make these
    temp = RandNbrs[LeftPtr];             //  statements as
    RandNbrs[LeftPtr] = RandNbrs[RightPtr]; //  efficient as
    RandNbrs[RightPtr] = temp;            //  possible.
    while (RandNbrs[++LeftPtr] < Pvalue); //  Continue this loop until
    while (RandNbrs[--RightPtr] > Pvalue); //  the pointers cross.
  }

  RandNbrs[LeftEnd] = RandNbrs[RightPtr]; //  After pointers cross, fill
  RandNbrs[RightPtr] = Pvalue;            //  left end and middle hole.

  //  All values to the left of RandNbrs[RightPtr] are <= Pvalue while all to
  //  the right are >= Pvalue. Next, test the 2 subgroups on either side to
  //  see if they are still larger than the minimum efficient size. If both
  //  are still too large, then place the larger one on the stack and
```

```
    //  partition the smaller. If only one needs partitioning, then partition
    //  it, otherwise get the left and right ends of a subgroup stored on the
    //  stack in an earlier operation.

                                          //  Move RightPtr into
    RightPtr--;                           //  unsorted left subgroup

    if (RightPtr < MidPtr) {              //  If left SubGroup is smaller
      if (RightPtr - LeftEnd > MinGroup) {//  If both are large then put
        QSleft[StackPtr] = LeftPtr;       //  right side on the stack
        QSright[StackPtr] = RightEnd;     //  and sort the left side.
        RightEnd = RightPtr;
        ++StackPtr;                       //  Ready for next subgroup
      }
      else if (RightEnd - LeftPtr > MinGroup) //  Else if just have to
        LeftEnd = LeftPtr;                //  sort the right side
      else {                              //  Else neither gets sorted. Get a
        LeftEnd = QSleft[--StackPtr];     //  prior subgroup from the stack.
        RightEnd = QSright[StackPtr];     //  (Will be garbage if all
      }                                   //  subgroups are sorted)
    }                                     //  End of "if left is smaller"

    else {                                //  Else left side is larger
      if (RightEnd - LeftPtr > MinGroup) {//  If both sides are large
        QSleft[StackPtr] = LeftEnd;       //  then put left side on
        QSright[StackPtr] = RightPtr;     //  the stack
        LeftEnd = LeftPtr;                //  and sort the right side
        ++StackPtr;                       //  Ready for next subgroup
      }
      else if (RightPtr - LeftEnd > MinGroup) //  else if left side is
        RightEnd = RightPtr;              //  too large, then sort it.
      else {                              //  Else neither gets sorted. Get a
        LeftEnd = QSleft[--StackPtr];     //  prior subgroup from the stack
        RightEnd = QSright[StackPtr];     //  (Will be garbage if all
      }                                   //  subgroups are sorted).
    }                                     //  End of "if left is larger"
  }                                       //  Repeat until all subgroups are
                                          //  small.

                                          //  Finish up with "Insertion Sort"

  for (i = 2; i <= Nbr2Sort; i++) {
    k = i;
    j = i - 1;
    temp = RandNbrs[k];
    while (RandNbrs[j] > temp) {
      RandNbrs[k] = RandNbrs[j];
      j--;
      k--;
    }
    RandNbrs[k] = temp;
  }

// Insertionsort(above fragment) follows:
/*
       mov    eax, DWORD PTR _Nbr2Sort
$L1481:
; Line 431
       cmp    eax, 2
       jb     SHORT $L1514
       mov    ecx, OFFSET FLAT:_RandNbrs+1
       mov    ebp, 2
       sub    ebp, ecx
$L1512:
```

```
; Line 434
       mov    dl, BYTE PTR [ecx+1]
; Line 435
       cmp    BYTE PTR [ecx], dl
       lea    edi, DWORD PTR [ecx+ebp]
       jbe    SHORT $L1517
       mov    esi, ecx
       mov    eax, ecx
$L1516:
; Line 436
       mov    bl, BYTE PTR [esi]
       mov    BYTE PTR [eax+1], bl
       mov    bl, BYTE PTR [eax-1]
; Line 438
       dec    edi
       dec    eax
       cmp    bl, dl
       mov    esi, eax
       ja     SHORT $L1516
; Line 435
       mov    eax, DWORD PTR _Nbr2Sort
$L1517:
; Line 434
       inc    ecx
; Line 440
       mov    BYTE PTR _RandNbrs[edi], dl
       lea    edx, DWORD PTR [ecx+ebp]
       cmp    edx, eax
       jbe    SHORT $L1512
$L1514:
*/

/*
  // Bubblesort
        for(i=1;i<((Nbr2Sort+1)-1);i++)
              for(j=1;j<((Nbr2Sort+1)-(i+1));j++)
                           if(RandNbrs[j] > RandNbrs[j+1])
                                    {   temp = RandNbrs[j+1];
                                        RandNbrs[j+1] = RandNbrs[j];
                                        RandNbrs[j] = temp;
                                    }
*/

  clocks2 = clock();
  printf("The time to sort %d items via Quicksort was %d clocks.\n", Nbr2Sort, clocks2 - clocks1);

}
```