Durango Bill’s

“C” Program to implement Sort Algorithms
(Source Code)

Return to the main Sort page

Web page generated via KompoZer

```
//****************************************************************************
//
//                                Sort Algorithms
//                                      by
//                                  Bill Butler
//
//  This program can execute various sort algorithms to test how fast they run.
//
//  Sorting algorithms include:
//  Bubble sort
//  Insertion sort
//  Median-of-three quicksort
//  Multiple link list sort
//  Shell sort
//
//  For each of the above, the user can generate up to 10 million random
//  integers to sort. (Uses a pseudo random number generator so that the list
//  to sort can be exactly regenerated.)
//  The program also times how long each sort takes.
//
//*************************************************************************


#include <stdheaders.h>                      //  The usual stdio.h, etc.

void GenRandom(void);                        //  Generate a list of random nbrs.
void BubbleSort(void);                       //  Bubble Sort
void InsertionSort(void);                    //  Insertion Sort
void MLLsort(void);                          //  Multiple Link List Sort
void QuickSort(void);                        //  Median-of-three Quicksort
```

```
void ShellSort(void);                           //  Choose from 3 gap sequences

void PauseMsg(void);                            //  Pause the screen display


unsigned RandNbrs[10000004];                    //  The list of random numbers
                                                //  will be generated here. Note:
                                                //  RandNbrs[0] is used as a sentinel
int MLLlinks[26777220];                         //  Used by MLLsort. Will use up to
                                                //  10,000,000 + 2^24 + 1 of this
int QSleft[30];                                 //  Stack arrays for the left/right
int QSright[30];                                //  ends of subgroups within
                                                //  QuickSort()
int Gaps13[24] = {0, 1, 3, 7, 21, 48, 112,      //  Three possible gap sequences
    336, 861, 1968, 4592, 13776, 33936,         //  may be used for experiments
    86961, 198768, 463792, 1391376,             //  with Shell Sort.
    3402672, 8382192, 21479367, 49095696,       //  See Sedgewick & ATT database
    114556624, 343669872};                      //  of integers for more info

int Gaps18[24] = {0, 1, 8, 23, 77, 281,         //  Gaps[i+2] = 4^(i+1) + 3*(2^i) + 1
    1073, 4193, 16577, 65921, 262913,
    1050113, 4197377, 16783361, 67121153,
    268460033, 1073790977};

int Gaps14[24] = {0, 1, 4, 13, 40, 121, 364, //  Gaps[i+1] = 3*Gaps[i] + 1
    1093, 3280, 9841, 29524, 88573, 265720,
    797161, 2391484, 7174453, 21523360,
    64570081, 193710244, 581130733, 1743392200};

int Gaps[24];                                   //  Will copy one of the above into
                                                //  here.

int Nbr2Sort;                                   //  Number of items to sort.

char Databuff[100];                             //  Buffer for user input (Assumes
                                                //  user does not abuse size)



int main(void) {

  int choice;

  puts("\nThis program demonstrates the performance of various sort algorithms.");
  puts("You may run time trials to compare results.\n");
```

```
  puts("Note: If you want to use identical inputs for the time trials,");
  puts("just use the same seed number for the random number generator.\n");
  PauseMsg();

  while(1) {                                //  Loop until user ends program
    puts("\n\n     *****     Enter number for menu choice     *****\n");
    puts("1)  Bubble sort");
    puts("2)  Insertion sort");
    puts("3)  Median-of-three Quicksort");
    puts("4)  Multiple link list sort");
    puts("5)  Shellsort (Choice of 3 different shell definitions)");
    puts("6)  End the program\n");

    gets(Databuff);
    choice = atoi(Databuff);

    if (choice == 6)
      break;

    GenRandom();                            //  Generate a list of random nbrs

    if (choice == 1) BubbleSort();
    else if (choice == 2) InsertionSort();
    else if (choice == 3) QuickSort();
    else if (choice == 4) MLLsort();
    else if (choice == 5) ShellSort();
  }
  return 0;
}




//******************************************************************************
//
//                                GenRandom
//
//  This routine generates a list of unsigned random integers in the range 0 to
//  4,294,967,295. An exact repetion of any list can be regenerated by using
//  the same seed number and the same quantity of numbers. Up to 10 million
//  numbers can be generated in the array RandNbrs[].
//
//  The random numbers will occupy positions RandNbrs[1] to RandNbrs[Nbr2Sort].
//  Nothing is placed in RandNbrs[0], but RandNbrs[0] will be used by some of
//  the sort routines to optimize execution speed.
//
```

```
//*****************************************************************************

void GenRandom(void) {

  int i;
  unsigned seed;
  unsigned Multiplier;

  do {
    puts("\nEnter number of elements to sort (3 - 10,000,000)");
    gets(Databuff);
    Nbr2Sort = atoi(Databuff);
  } while ((Nbr2Sort < 3) || (Nbr2Sort > 10000000));

  puts("\nEnter a seed number for the random number generator");
  gets(Databuff);
  seed = atoi(Databuff);

  Multiplier = 3141592621;
  for (i = Nbr2Sort; i; i--) {                //  Fill array with random
    seed *= Multiplier;                       //  numbers.
    seed++;
    RandNbrs[i] = seed;
  }

  puts("\nDo you want to see the random numbers before they are sorted (Y or N)?");
  gets(Databuff);
  if (tolower(Databuff[0]) == 'y') {
    puts("");
    for (i = 1; i <= Nbr2Sort; i++) {
      printf("%4d) %'16u\n",i , RandNbrs[i]);
      if (!(i%20))                            //  Keeps list from running
        PauseMsg();                           //  off top of screen.
    }
    PauseMsg();
  }
}




//***************************************************************************
//
//                                Bubble Sort
//
//  Bubble sort is not exactly the world's fastest sorting algorithm, but for
```

```
//  some reason, everyone seems to like it. Maybe it's related to:
//
//  "Tiny Bubbles"
//  "In the wine"
//  "Make you feel happy"
//  "Make you feel fine"
//
//  This version of Bubble Sort will sort the array RandNbrs[] in ascending
//  order. When the routine is finished, the lowest number in the RandNbrs[]
//  array will be found in position RandNbrs[1] while the largest number will
//  be in position RandNbrs[Nbr2Sort].
//
//  The algorithm starts by looking at positions 1 and 2 in the array. If the
//  number at position 1 is larger than the number at position 2, then the two
//  numbers are exchanged. The end result will leave the larger of the two
//  numbers at position 2 and the smaller at position 1.
//
//  After the above step, the algorithm looks at the numbers at positions 2 and
//  3. If the number at position 2 is larger, then it is exchanged so that the
//  larger number will now be in position 3.
//
//  The "j" loop continues this process until it reaches the bottom (highest
//  index location) of the array. At this point the largest number in the array
//  has been moved to the bottom of the array, and everything else has "bubbled"
//  up one level. Then, the "i" loop exerts control and the whole process
//  repeats. However, this time through, the 2nd largest number in the array
//  will be shifted down to the 2nd lowest position in the array. (Again this
//  stopping point is controled by the "i" loop.)
//
//  By the time "i" decreases to 1, the whole array is sorted.
//
//*****************************************************************************

void BubbleSort(void) {

  int i, j, k;
  unsigned temp;
  double Time1, Time2;

  puts("\nStarting Bubble Sort");

  Time1 = (double)clock();                        //  Save time to 0.001 sec.
  Time1 = Time1/(double)CLOCKS_PER_SEC;

  for (i = Nbr2Sort - 1; i; i--) {
    for (j = 1, k = 2; j <= i; j++, k++) {
```

```
      if (RandNbrs[j] > RandNbrs[k]) {
        temp = RandNbrs[j];
        RandNbrs[j] = RandNbrs[k];
        RandNbrs[k] = temp;
      }
    }
  }

  Time2 = (double)clock();
  Time2 = Time2/(double)CLOCKS_PER_SEC;

  printf("\nThe time to sort %'d items via Bubble Sort was %g seconds.\n",
    Nbr2Sort, Time2 - Time1);

  PauseMsg();

  puts("\nDo you want to see the random numbers after they were sorted (Y or N)?");
  gets(Databuff);
  if (tolower(Databuff[0]) == 'y') {
    puts("");
    for (i = 1; i <= Nbr2Sort; i++) {
      printf("%4d) %'16u\n", i, RandNbrs[i]);
      if (!(i%20))
        PauseMsg();
    }
    PauseMsg();
  }
}




//****************************************************************************
//
//                                Insertion Sort
//
//  Insertion sort is simple to code and difficult to beat if you are sorting
//  a short list of elements. It is also very good at sorting a much larger
//  list that is nearly in sorted order. It is thus used as the basis of Shell
//  Sort and the final stage of "median-of-three Quicksort".
//
//  Computer processors usually have "on board" cache memories that provide an
//  added boost to Insertion Sort. Portions of the array to be sorted will be
//  stored in the (faster access) cache memory. If an array is nearly sorted
//  when Insertion Sort is called, then most of Insertion Sort's operations
//  will take place inside this high speed cache memory.
```

```
//
//  This version of Insertion Sort will sort the array RandNbrs[] in ascending
//  order. When the routine is finished the lowest number in the RandNbrs[]
//  array will be found in position RandNbrs[1] while the largest number will
//  be in position RandNbrs[Nbr2Sort].
//
//  The algorithm starts with some number in place at RandNbrs[1]. Then it
//  moves down to array position 2. The number at position 2 is copied to a
//  temporary holding position in variable "temp". Then all numbers that are in
//  lower index positions in the array and are also larger than what is in
//  "temp" are moved down one position. When a smaller number is encountered,
//  then the number in "temp" is inserted back into the array.
//
//  Next the algorithm works on the number in array position 3. Again all
//  numbers that are larger than what is in "temp" are moved down one position.
//  When a smaller (or equal) number is encounter, the number in "temp" is
//  inserted back into the empty space.
//
//  The algorithm continues in this fashion until it reaches the bottom of
//  the list to be sorted.
//
//*****************************************************************************

void InsertionSort(void) {

  int i, j, k;
  unsigned temp;
  double Time1, Time2;

  puts("\nStarting Insertion Sort");

  Time1 = (double)clock();                      //  Save time to 0.001 sec.
  Time1 = Time1/(double)CLOCKS_PER_SEC;

  RandNbrs[0] = 0;            //  Sentinel for sort. Must be <= the lowest
                              //  value that will be sorted. Using a sentinel
                              //  speeds up the algorithm since an additional
                              //  run-off-the-"0"-end-of-the-array test will
                              //  not be needed.

  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;
  }

  Time2 = (double)clock();
  Time2 = Time2/(double)CLOCKS_PER_SEC;

  printf("\nThe time to sort %'d items via Insertion Sort was %g seconds.\n",
    Nbr2Sort, Time2 - Time1);

  PauseMsg();

  puts("\nDo you want to see the random numbers after they were sorted (Y or N)?");
  gets(Databuff);
  if (tolower(Databuff[0]) == 'y') {
    for (i = 1; i <= Nbr2Sort; i++) {
      printf("%4d) %'16u\n", i, RandNbrs[i]);
      if (!(i%20))
        PauseMsg();
    }
    PauseMsg();
  }
}




//**************************************************************************
//
//                                MLLsort
//
//  If the numbers to be sorted are within a known range, and if on average
//  they are distributed approximately evenly, and if you have lots of extra
//  random access memory, then a multiple link list sort may be faster than
//  all other sorting algorithms. In this application, the numbers to be sorted
//  are in the array RandNbrs[]. The sorting algorithm never moves them.
//
//  Instead, for each element to be sorted, its proper sort location is
//  calculated as an index into the MLLlinks[] array. This initial calculated
//  link head address is equal to the sum of the Nbr2Sort plus a calculated
//  distance beyond this index number. The calculated distance can be varied to
//  see what produces the best result.
//
```

```
//  The correlation between the RandNbrs[] array and the MLLlinks[] array
//  looks like:
//
//             ------------------------
//  RandNbrs |    |    |    |    |    |
//             ------------------------
//               0    1    2    Nbr2Sort
//
//             ---------------------------------------------------------------
//  MLLlinks |Link lists for each grp |       Link head for each group      |
//             ---------------------------------------------------------------
//              0     1   2     Nbr2Sort  Calculated pos. for each random nbr.
//
//  Once a calculated address is known, a link list is started using this
//  number's calculated addess. If any subsequent RandNbrs[] element ends up
//  using this same calculated address, it is added to the link list for this
//  address. The links in any link list are adjusted for these additions such
//  that the access order will be in sorted order.
//
//***************************************************************************

void MLLsort(void) {

  int HeadBase, i, count;
  unsigned NbrHeads, ShiftAmt, value;
  unsigned ptr1, ptr2;
  double Time1, Time2;

  puts("\nStarting Multiple Link List Sort");

  Time1 = (double)clock();                      //  Save time to 0.001 sec.
  Time1 = Time1/(double)CLOCKS_PER_SEC;

  RandNbrs[0] = 4294967295;                     //  Sentinel for sort.
                                                //  Must be >= the largest
                                                //  number to be sorted.
  HeadBase = Nbr2Sort + 1;
  NbrHeads = 4;                                 //  Calculate how many list
  ShiftAmt = 30;                                //  heads and how many bit
  while (NbrHeads < Nbr2Sort) {                 //  positions to shift for
    ShiftAmt--;                                 //  indexing.
    NbrHeads <<= 1;
  }
                                                //  Optional debug info
//  printf("With %'d numbers to sort the calculated value for NbrHeads is %'d\n",
//        Nbr2Sort, NbrHeads);
```

```
for (i = Nbr2Sort + NbrHeads; i >= HeadBase; i--)
  MLLlinks[i] = 0;                          //  Zero out the link heads. Note:
                                            //  If you are only going to run
                                            //  this routine once, you can take
                                            //  advantage of the built in
                                            //  initialization routines in C
                                            //  and ignore this step.

for (i = Nbr2Sort; i; i--) {                //  For all input numbers.
  value = RandNbrs[i];                      //  Will calculate where it should
  ptr1 = value;                             //  go. Construct index.
  ptr1 >>= ShiftAmt;
  ptr1 += HeadBase;
                                            //  Search link list to see where
                                            //  to insert this element. Most of
                                            //  the time the new value will be
                                            //  the 1st in the list.
  for (ptr2 = MLLlinks[ptr1]; RandNbrs[ptr2] < value;
      ptr1 = ptr2, ptr2 = MLLlinks[ptr2]);

      //  Note: The average length of these link lists does not increase as
      //  "Nbr2Sort" increases. The processing time per sort element is
      //  a constant that is independent of "Nbr2Sort". Thus the algorithm
      //  runs in pure linear time and not something slower such as
      //  N*log(log(N)) time.

  MLLlinks[ptr1] = i;                       //  Insert location of new
  MLLlinks[i] = ptr2;                       //  value in link list.
}

Time2 = (double)clock();
Time2 = Time2/(double)CLOCKS_PER_SEC;

printf("\nThe time to sort %'d items via MLL Sort was %g seconds.\n",
  Nbr2Sort, Time2 - Time1);

PauseMsg();

puts("\nDo you want to see the random numbers after they were sorted (Y or N)?");
gets(Databuff);
if (tolower(Databuff[0]) == 'y') {
  count = 0;
  ptr2 = Nbr2Sort + NbrHeads;
  for (i = HeadBase; i <= ptr2; i++) {
    for (ptr1 = MLLlinks[i]; ptr1; ptr1 = MLLlinks[ptr1]) {
```

```
        count++;
        printf("%4d) %'16u\n", count,  RandNbrs[ptr1]);
        if (!(count%20))
          PauseMsg();
      }
    }
    PauseMsg();
  }
}




//*****************************************************************************
//
//                                    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 flier 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) {

  int    i, j, k, StackPtr;
  int    LeftEnd, RightEnd, LeftPtr, RightPtr, MidPtr, MinGroup;
  unsigned Pvalue, temp;
  double Time1, Time2;

  puts("\nStarting QuickSort");

  Time1 = (double)clock();                    //  Save time to 0.001 sec.
```

```
Time1 = Time1/(double)CLOCKS_PER_SEC;

RandNbrs[0] = 0;                            //  Sentinel for sort - used by
                                            //  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 = 65;                              //  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;
  }

  Time2 = (double)clock();
  Time2 = Time2/(double)CLOCKS_PER_SEC;

  printf("\nThe time to sort %'d items via QuickSort was %g seconds.\n",
    Nbr2Sort, Time2 - Time1);


  PauseMsg();

  puts("\nDo you want to see the random numbers after they were sorted (Y or N)?");
  gets(Databuff);
  if (tolower(Databuff[0]) == 'y') {
    for (i = 1; i <= Nbr2Sort; i++) {
      printf("%4d %'16u\n", i, RandNbrs[i]);
```

```
      if (!(i%20))
        PauseMsg();
    }
    PauseMsg();
  }
}




//*******************************************************************************
//
//                                    ShellSort
//
//  If you are only going to learn how one sorting algorithm works, concentrate
//  on Shell Sort. On most arrays that you are ever going to work with it is
//  nearly as fast as Quicksort, and it is much easier to code
//  (and understand).
//
//  Shell Sort is based on Insertion Sort. In fact, if you are working on a
//  very small group of numbers, it is exactly the same as Insertion Sort. If
//  you are sorting a larger group, then it is just an expansion of Insertion
//  Sort. Shell Sort breaks down large arrays into a series of small sections
//  which it then treats as though they were "Insertion Sort".
//
//
//            ---------------------------------------------------------------
//  RandNbrs | A | B | C | D | A | B | C | D | A | B | C | D | A | B | C | D |
//            ---------------------------------------------------------------
//             1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
//
//  For example, let's assume you are going to sort 16 elements and are using
//  the 1, 4, 13, etc. sequence for "gap" sizes. The algorithm will first use a
//  "Gap" size of 4. It will run 4 separate simultaneous "Insertion Sorts". One
//  of these will be an "Insertion Sort" on the numbers in the "A" positions.
//  The other 3 "Insertion Sorts" will use the "B", "C", and "D" groups. When
//  the "Gap = 4" process is finished, the array will be roughly sorted with
//  most of the small value numbers concentrated near the low end of the array.
//  Similarly, most of large value numbers will be concentrated near the high
//  end of the array. The algorithm then continues with a "Gap" size of 1. This
//  is essentially an ordinary "Insertion Sort", and "Insertion Sort" is very
//  efficient on arrays that are nearly sorted.
//
//  If the number of elements to be sorted is still larger, then the elements
//  are initially sorted using a "gap" size of 13. This is followed by a "gap"
//  of 4 (as above) and finally a "gap" of 1. Still larger lists that are to be
```

```
//  sorted will use larger initial "Gap" sizes, and then gradually decrease the
//  "Gap" size as progress is made.
//
//  The efficiency of Shell Sort can be fine tuned by changing the sequence of
//  "gap" sizes. The 1, 4, 13, 40... series was originally sugested by Knuth.
//  Two other series are better candidates for the gap sizes. The user can
//  experiment with a choice of:
//  1)  1, 3, 7, 21, 48...
//  2)  1, 4, 13, 40...
//  3)  1, 8, 23, 77, 281, 1073...
//
//*****************************************************************************

void ShellSort(void) {

  int choice, i, j, k, GapPtr, Gap;
  unsigned temp;
  double Time1, Time2;

  puts("\nYou may experiment with the gap sizes that are used in Shell Sort");
  puts("\nPick one of the following sequences for the gap size.");
  puts("(Any other number cancels Shell Sort)\n");

  puts("1)  1, 3, 7, 21, 48, 112,...");
  puts("2)  1, 4, 13, 40, 121, 364,...");
  puts("3)  1, 8, 23, 77, 281, 1073,...");

  gets(Databuff);
  choice = atoi(Databuff);

  if (choice == 1) {                    //  Copy one of the three gap sequences
    for (i = 1; i <= 20; i++)
      Gaps[i] = Gaps13[i];
  }
  else if (choice == 2) {
    for (i = 1; i <= 20; i++)
      Gaps[i] = Gaps14[i];
  }
  else if (choice == 3) {
    for (i = 1; i <= 20; i++)
      Gaps[i] = Gaps18[i];
  }
  else return;

  puts("\nStarting Shell Sort");
```

```
  Time1 = (double)clock();                              //  Save time to 0.001 sec.
  Time1 = Time1/(double)CLOCKS_PER_SEC;

  temp = Nbr2Sort/3;                                    //  Set up GapPtr
  for (GapPtr = 1; Gaps[GapPtr] < temp; GapPtr++);
  GapPtr--;
  Gap = Gaps[GapPtr];

  do {
    for (i = Gap + 1; i <= Nbr2Sort; i++) {
      temp = RandNbrs[i];
      k = i;
      for (j = i - Gap; j > 0; j -= Gap) {
        if (RandNbrs[j] <= temp)
          break;
        RandNbrs[k] = RandNbrs[j];
        k = j;
      }
      RandNbrs[k] = temp;
    }
  } while (Gap = Gaps[--GapPtr]);

  Time2 = (double)clock();
  Time2 = Time2/(double)CLOCKS_PER_SEC;

  printf("\nThe time to sort %'d items via Shell Sort was %g seconds.\n",
    Nbr2Sort, Time2 - Time1);

  PauseMsg();

  puts("\nDo you want to see the random numbers after they were sorted (Y or N)?");
  gets(Databuff);
  if (tolower(Databuff[0]) == 'y') {
    for (i = 1; i <= Nbr2Sort; i++) {
      printf("%4d) %'16u\n", i, RandNbrs[i]);
      if (!(i%20))
        PauseMsg();
    }
    PauseMsg();
  }
}



//*******************************************************************
```

```
//
//                    Misc routines
//
//******************************************************************

void PauseMsg(void) {

  puts("\nPress RETURN to continue.");
  gets(Databuff);
}
```