605.202 Data Structures Lab 4

This lab assignment requires you to compare the performance of two distinct sorting algorithms to obtain some appreciation for the parameters to be considered in selecting an appropriate sort. Write a **HeapSort** and a **Shell Sort**. They should both be recursive or both be iterative, so that the overhead of recursion will not be a factor in your comparisons. In this case, iteration is recommended. **Be sure to justify your choice. Also, consider how your code would have differed if you had made the other choice.**

The strategy behind a **Shell Sort** is to create a more nearly optimal environment for a simple, relatively inefficient sort technique, namely Simple Insertion Sort. This optimal environment allows the simple strategy to be efficient. Use the following sets of increments

1, 4, 13, 40, 121, 364, 1093, 3280, 9841, 29524 (Knuth's sequence)

1, 5, 17, 53, 149, 373, 1123, 3371, 10111, 30341

1, 10, 30, 60, 120, 360, 1080, 3240, 9720, 29160

One or more sets of increments of your choice.

Please note that the increment sets will need to be supplemented if you use data sets larger than approximately 29000. Use the sequences increment sets like you use Knuth's increments. Find the first value larger than the file. Move back two increments to find the starting increment. So in the third sequence, for a file of size 1200, you would use increments of 360, 120, 60, 30, 10, 1, in that order. You will have four different Shell sorts to run.

Heap Sort is a practical sort to know and is based on the concept of a heap. It has two phases: Build the heap and extract the elements in sorted order from the heap. Altogether, you will have five sorts: 1 Heap and 4 Shell.

Create input files of four sizes: 50, 500, 1000, 2000, and 5000 integers. For each size file, make three versions. On the first, use a randomly ordered data set. On the second, use the integers in reverse order. On the third, use the integers in normal ascending order. (You may use a random number generator or shuffle function to create the randomly ordered file. It is important to avoid too many duplicates. Keep them to about 1%). **This means you have an input set of 15 files plus whatever you deem necessary and reasonable.** Files are available in the course site if you want to copy them. Your data should be formatted so that each number is on a separate line with no leading blanks. There should be no blank lines in the file.

Each sort must be run against all the input files. This will give you at least 75 runs. **For grading purposes, for each sort, turn in only output from the files of size 50. You will have 15 sets of output to turn in for the size 50 files. Your code needs to print out the sorted values and the times for each of the Shell Sorts and the Heap Sort for each of the three orders for size 50.**

Your program should access the system clock to get some time values for the different runs. The call to the clock should be placed as close as possible to the beginning and the end of each sort. If other code is included, it may have a large, fixed, cost, which would tend to drown out the differences between the runs, if any. Why take a chance! **If you get too many zero time data values or any negative time values then you must fix the problem.** One way to do this is to use larger files than those specified. Another solution is to perform the sorting in a loop, N times, and calculates an average value. You would need to be careful to start over with unsorted data, each time through the loop.

Turn in a analysis comparing the two sorts and their performance. Be sure to comment on the relative runtimes of the various runs, the effect of the order of the data, the effect of different size files, and the effect of different increment sizes for the Shell Sort. Which factor has the most effect on the efficiency? Be sure to consider both time and space efficiency. Be sure to justify your data structures. As time permits consider implementing a Straight Insertion Sort to compare with Shell Sort. Also, consider files of size 10,000 or additional random files - perhaps with 15-20% duplicates. Your write-up must include a table of the times obtained.

The source code you turn in needs to print out the sorted values. For grading purposes, it does not need to print the times, but the times should be printed in the sample runs you turn in.

It is to your advantage to turn the lab in on time to avoid competition with studying for the final. **This lab will not be accepted after the due date, except by prior arrangement.**