EECS 2500 – Fall 2015

Programming Project #3 – Due December 1, 2015 23:59:59

For this assignment, you will code and compare the performance of various sorting algorithms. The goal of this assignment is not so much to create working code (I've given you the pseudocode and/or Java code for 90% of what you'll need); it's to USE the code to gather data to analyze and draw conclusions about algorithms.

Your program will create an array of varying size, fill it with random values, and then sort that array with different sorting algorithms, noting the number of comparisons, data movements, and elapsed time.

You will use `math.random()` to fill your array with integers in the range of 0 - 999,999.

Sort the array with bubble sort, counting the number of comparisons and the number of swaps (exchanges) made, and the elapsed time (for small values of *n*, 100 in this case, the elapsed time may appear as zero; that's OK).

Repeat this process for Insertion Sort, Selection Sort, Quick Sort, and Shell Sort:

Implement Shell Sort three different ways: Some implementations of Shell Sort use sequences that can be built simply and efficiently in your code; others require pre-computed interval sequences stored in an array.

Shell Sort 1: Use Hibbard's sequence. This one can be coded in-line. Hibbard's sequence uses values that are always one less than a power of two. This can easily be generated within your code, rather than via a list. Start with $d = 1$. Double d until $d > n$. Subtract 1 from d. Divide that by 2 (integer division, which will throw away the remainder). That's your starting value (d_0). Example: If $n = 1000$, start with d at 1. Double it (2). Double it again (4). Double it again (8)…(256), (512), (1024). Once $d > n$, subtract 1 (1023), and divide by 2, discarding the remainder (511). That's your initial interval's value (d_0). At the end of each pass of Shell Sort, divide the previous distance by 2 (discarding the remainder), so 511 becomes 255 for the next pass (then 127, 63, 31, 15, 7, 3, and 1). The sort completes when d becomes 0 (the last pass will use a distance of 1, so when we attempt to divide that in half, we get zero, and exit, rather than starting another pass).

Shell Sort 2: Use Knuth's sequence: This one can also be coded in-line. Knuth's sequence can be built much like Hibbard's. Start with $d = 1$. Triple it and add 1. Repeat until $d > n$. Then divide by 3 (discarding the remainder). At the end of each sorting pass, divide the interval by 3. For $n = 1000$, you would start with $d = 1$, $3d+1 \rightarrow 4$, $3d+1 \rightarrow 13$, $3d+1 \rightarrow 40$, $3d+1 \rightarrow 121 \rightarrow 364 \rightarrow 1093$. Then, divide by 3 (364), and use that as your d_0. After each pass, divide d by 3, discarding the remainder (364, 121, 40, 13, 4, 1). When you complete the last pass with a gap of 1, dividing it by 3 will result in zero, so the sort terminates, rather than starting another pass.

Shell Sort 3: Use Pratt's sequence. This one will require coding the list in an array, and then just using the appropriate values. Pratt's sequence is all numbers of the form 2^p3^q for all values of p and q such that $2^p3^q < n$. The numbers in Pratt's sequence, therefore, are 1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 27, 32, 36, … Every number less than n whose only factors are 2 and/or 3. I will leave it up to you do build this list. Be careful here. There lots of values on this list; don't miss any. Once you have these in an array, just figure out how

far up the list you can go for the last value less than *n*, and start there. Build your list up to 995328 (this will allow you to sort a list of up to 1 million items).

Hint: Consider using Excel to build this list as a 2-D table, and then convert it to a one-dimensional list. Put powers of 3 vertically in column A, and powers of 2 horizontally on row 1. Then fill in the remainder of the grid with the various products of all of the powers, keep the ones you need (i.e., the ones less than or equal to 995328), and sort them into a list, which you can use as an array initializer.

For each sort, your approach will be:

```
For SIZE = 100 to 20,000 in steps of 100
{
        Initialize statistics
        Create a SIZE-sized array
        For REP = 1 to RepCount
        {
                Fill the array with random data
                Sort the array (within the sort, count every comparison and every swap)
                <During debugging, make sure the array really IS sorted!>
        }
        Output the average statistics for this SIZE
}
Output the final run time for this algorithm
```

Then repeat the process for each algorithm.

During development, I strongly recommend RepCounts of 1 for all algorithms. On my desktop PC, that resulted in a total run time of about 90 seconds. When your code is running correctly, you will raise the RepCounts to produce more reliable (i.e., less noisy) results: For the three $O(n^2)$ sorts (Bubble, Selection, and Insesrtion), use a RepCount of 10. For the remainder (Quick, Shell1, Shell2, and Shell3), use a RepCount of 1000. This combination took my machine approximately 35 minutes to complete. Of course your mileage will vary (perhaps significantly). If you're working on a laptop, make sure it is plugged in the entire time. Most laptops switch CPU speeds depending on whether they're running on battery or AC power, so (a) to have it run faster, and (b) to make sure the battery doesn't die before it finishes, plug it in before starting. Under no circumstances should you plug or unplug your laptop WHILE this is running – it will ruin the results. Make sure your PC isn't being used for anything else while this test is running (no web surfing, game playing, music or video streaming, etc.). Ideally, you should disable your screen saver, or at least move the mouse a little every few minutes to prevent the screen saver from running.

If you would like to have your data be even cleaner, feel free to raise the rep counts from 10 & 1,000 to 100 & 10,000, but realize that the "35-minute" runtime will become "overnight". Make sure your system's power-saving features will not kick in and put the PC to sleep while this runs!

Assemble your results into an Excel spreadsheet, using the "Data / Text To Columns" approach covered in class, and graph the six sorts' run times for values of *n* up to 20,000. Use a straight-line (no marker) X-Y scatter plot. Label the six series "Bubble", "Selection", "Insertion", "Quick", "Shell1", "Shell2", and "Shell3". Create the new plot as a new sheet called "Run Time", and set its title to "Run Time by Algorithm". Set the X-axis to run from 0 to 20000, with major units of 2000. Title the X-axis "Number of Items to Sort", and title the Y-axis "Sort time (Seconds)". Format the Y-axis to have one decimal place.

You will create a report with your findings (embedding the graphs in your report as you discuss them is a good idea). For this graph, address the following in your report:

1) Which of the $O(n^2)$ algorithms has the best and worst overall performance? What is it about the one with the longest run time that makes it take so long? Why does the one that runs the fastest do so?

Produce a second graph, using the number of comparisons (rather than the run times). This graph should also be its own sheet ("Comparisons") within the Excel workbook, with a title of "Comparisons Made by Algorithm". Name the data series the same way as in the run-time graph, and use a Y-axis title of "Comparisons Made (millions)". Have Excel display the Y-axis in millions with no decimal places

For this graph, address in your report

2) Why do Bubble and Selection sort have exactly the same number of comparisons, and insertion makes (approximately) half as many comparisons as the other two?

Produce a third graph, just like the second, except using the number of swaps (rather than comparisons). Address this in your report:

3) Why do Bubble and Insertion do basically the same number of swaps, but Selection does so few?

Create new columns in your spreadsheet, containing the sum of the comparisons and swaps for each algorithm. Title this column "Work", with a Y-axis label of "Total Work Done (Operations, in Millions)". Address this in your report:

4) If the total work done in a sorting algorithm is captured in the comparisons and swaps (the loop overhead is negligible, and is a non-issue here), why do the "Run Time" and "Work" graphs look so different?

At this point, we have pretty well exhausted what there is to analyze with respect to the $O(n^2)$ sorts, but we have not examined the others. Have Excel make *copies* of the four graph sheets you already have, calling them "Run Time (2)", "Comparisons (2)", "Swaps (2)", and "Work (2)". In these sheets, remove the data series for the $O(n^2)$ sorts, so that we may concentrate on the others.

From these graphs, address these questions in your report:

5) We said that Quick Sort is the fastest general-purpose sorting algorithm. Do your graphs (swaps, comparisons, total work, and run time) confirm or refute this assertion?

6) Shell Sort is still an excellent non-recursive, in-place sort. Which of the three Shell sequences produced the fewest comparisons? Did this sequence produce the lowest run time?

7) In lecture, I made the assertion that Pratt's sequence produced the fewest swaps. Do your graphs confirm or refute this claim?

Re-run your program with RepCounts of 0 for the $O(n^2)$ sorts, 32 for the other four, and let SIZE range from 10,000 to 1,000,000 (one million) in steps of 10,000 [Note: This took my desktop 26.5 minutes to run]. Make a new sheet (Sheet2) within your Excel workbook to hold this new data.

8) Create a new Run Time graph ("Run Time (3)"), with labeled X- and Y-axes. Format the Y-axis with 2 decimal places, and scale the X-axis from 0 to 1000000. Is QuickSort still the fastest?

9) Create a new "Work" column, whose contents are the sum of the comparisons and swaps. Graph this (similar formatting as before). Compare the Shell Sort algorithm that does the least work with the one that takes the longest to run. How do you explain the apparent discrepancy?

Finally, re-run your program with RepCounts of 0 for the O(n^2) sorts and for QuickSort, 100,000 for the three Shell Sorts, and let Size run from 10 to 150 in steps of 1 [about 5 minutes of run time]. Because the lists are so small, elapsed run time will be meaningless (non-measurable). Create graphs for the three Shell Sort sequences for Comparisons ("Comarares (Small N)"), Swaps ("Swaps (Small N)"), and total work ("Work (Small N)")

10) Which Shell Sort sequence makes most comparisons?
11) Which Shell Sort sequence makes the fewest swaps?
12) Which Shell Sort sequence does the least overall work?

Your code should be well-documented. All output should be to the console (no GUI output).

Because this is simply an algorithmic investigation, it doesn't lend itself very well to an object-oriented approach; you can code everything in a single class – there are no nodes, no lists, or any other objects (other than arrays).

As always, your work is to be yours and yours alone. Don't swap code (I've already provided most of it), and don't discuss your findings or "what the answers should be" – this should be your own investigation into how the algorithms behave on your machine.

Some of you might not have much experience using Excel for graphing. To the extent that you need to get help from a classmate on the ***mechanics*** of creating a graph, setting / formatting the axes, creating titles, naming the series (for the legend), etc., feel free. When it comes to the *content* of your graphs, though, don't share.

Submit a 7-zip archive of your ENTIRE Java Workspace directory (and its subfolders) to Blackboard. The name of the folder containing your Eclipse Workspace (and hence the 7-zip file) should be <LastName><comma><Space><FirstName>. I should be able to unzip your workspace, point Eclipse to it, and run your code without modification. You should be using the 32-bit versions of Eclipse Standard (Mars.1), with the JDK 8u60 (even though 8u65/66 has been released). If you need them, they are available at:

http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/mars/1/eclipse-java-mars-1-win32.zip

http://www.oracle.com/technetwork/java/javase/downloads/java-archive-javase8-2177648.html#jdk-8u65-oth-JPR

Also submit your Excel Workbook and Analysis Report (in MS Word .doc/docx format or as a PDF) (***not*** in the 7-zip) to Blackboard

If you have questions, please let me know ASAP.