CS/COE 0445 Spring 2015 Assignment 4

Online: Monday, March 2, 2015
Due: All files (see details below) submitted in a single .zip file to the proper directory in the submission site by **11:59PM on Monday, March 23, 2015.**
Late Due Date: 11:59PM on Wednesday, March 25, 2015

Purpose and Goal: Now that we have looked at Quicksort and discussed several variations, we'd like to empirically verify what we have discussed about their relative efficiencies. We will do this by timing each sort in different situations on different size arrays. We will then tabulate our results and compare the algorithms' performances. We hope to see differences in the relationships between the run-times and array sizes for the different algorithms, and possibly come to some conclusions about which versions are best in given situations and overall.

Details: You will test 5 versions of Quicksort:
1) Simple Quicksort with A[last] as the pivot (in file Quick.java)
2) Median of 3 Quicksort as given in TextMergeQuick.java (base case array size < 5)
3) Median of 3 Quicksort as given in TextMergeQuick.java but with base case array size < 10
4) Median of 3 Quicksort as given in TextMergeQuick.java, but with base case array size < 20
5) Random Pivot Quicksort with base case array size < 5

The code for versions 1-4 is already completely written – you only have to change the base case value in the Median of 3 sort from a constant to a variable so that you can give it different values during your program execution. You must write the Random Pivot Quicksort so that it works correctly. This is actually very similar to the simple Quicksort, except that you choose the pivot index as a random integer between first and last (inclusive) rather than choosing it as A[last].
Your primary task in this assignment is to write a main program that will enable the user to time all 5 of the algorithms under different circumstances, and then to tabulate and analyze the resulting data.

Input and Variable Setup: Your program should allow the following to be input from the user:
1) The size of the arrays to be tested
2) The number of trials for each test. The overall time for the test will be the average of the times for each of the trials. For random data, each trial should have different numbers, but the data for a given trial should be **the same random data for each algorithm**. In other words, consider, for example, an array called A1, algorithms QS1 and QS2, and trials T1 and T2. If A1 is filled with random numbers for QS1 in trial T1, then those same numbers (in the same initial positions) should be used for QS2 in trial T1. However, different random numbers should be generated for trial T2, again using the same numbers for both QS1 and QS2.
3) The name of the file your results will be output to

For each algorithm your program should iterate through 3 initial setups of the data:
a) Random – in this case you will fill the arrays with random integers. To make your assessments more accurate, each of your algorithms should utilize the same random data, as mentioned above. This can be accomplished in several ways but you will lose credit if the data is not the same.
b) Already sorted (low to high) – in this case simply fill the arrays with successive integers starting at 1.
c) Already reverse sorted (high to low) – in this case simply fill the arrays with decreasing integers starting at the array length.

Timing: You will time your algorithms using the predefined method
```
System.nanoTime()
```
This method returns the time elapsed on the system timer in nanoseconds. You will time one trial in the following

fashion:

```
long start = System.nanoTime();
// Execute the sorting method here (array should ALREADY be filled before timing starts)
long finish = System.nanoTime();
long delta = finish – start;
```

Since you are performing multiple trials, for a given algorithm you will add the times for the trials together, then divide by the number of trials to get the average time per trial. You may also want to divide by 1 billion to get your final results in seconds rather than nanoseconds.

Output: For each of the variations in the run, your program must output its results to the file named by the user. Note that since you have 3 data setups and 5 algorithms, **each overall execution of your program should produce 15 different results.** Each result should look something like the following example:

```
Algorithm: Simple QuickSort
Array Size: 25000
Order: Random
Number of trials: 10
Average Time per trial: 0.0063856 sec.
```

Trace Output Mode: In order for your TA to be able to test the correctness of your sorting algorithms and main program logic, you are required to have a Trace Output Mode for your program. This mode should be automatically set when the Array Size is <= 20. In Trace Output Mode, your program should output all of the following to standard output (i.e. the display) for EACH trial of EACH algorithm:

Algorithm being used
Array Size
Data configuration (sorted, reverse sorted, random)
Initial data in array prior to sorting
Data in array after sorting
Time (in nanoseconds) required for the sort

The evaluation of the correctness of your algorithms and data processing will be heavily based on the Trace Output Mode for your program. If you do not implement this or it does not work correctly, you will likely lose a lot of credit.
Note: Be sure that Trace Output Mode is OFF for arrays larger than 20.

Runs: The goal is to see how the run-times of the algorithms change as the size of the arrays increases. However, actual run-times will vary based on the speed of your machine. Follow the guidelines below for the array sizes. Use 10 trials for all of your runs.

Size = 25000, Filename = test25k.txt
Size = 50000, Filename = test50k.txt
Size = 100000, Filename = test100k.txt

Note: Only do the first 3 sizes above for the Simple QuickSort. Even with these it may take a while for the simple QuickSort algorithm and you will have to increase the stack size of the JRE to accommodate the execution – see below. For the sizes below you will only have 12 results.

Size = 200000, Filename = test200k.txt
Size = 400000, Filename = test400k.txt
Size = 800000, Filename = test800k.txt
Size = 1600000, Filename = teset1600k.txt

An example run may appear as shown below:

```
assig4> java -Xss10m Assig4
Enter array size: 25000
Enter number of trials: 10
Enter file name: test25k.txt
```

Here is an example output file: test25k.txt

Results: After you have finished all of your runs, tabulate your results in an Excel spreadsheet. Use a different worksheet for each initial ordering (random, sorted, reverse sorted). In each worksheet, make a table of your results with the array sizes as the columns and the algorithms as the rows. Also make a graph for each of your tables so that you can visualize the growth of the run-time for each algorithm. You must also **write a brief summary / discussion of your results**. Based on your tables, indicate the best algorithm for each of the initial data orderings. Based on your overall results (for all data orderings), speculate on what you think the best of the 5 algorithms is for general purpose use. Your write-up should be well written and justified by your results. Your write-up can be embedded in your spreadsheet or submitted as a separate document (ex: a Word document).

Submission: Submit all of your Java source files, as usual, but **also submit all output files and your Excel file (and Word file)**. As usual, put all of these files into **a single .zip file** for submission.

Additional Requirements, Hints and Help:

- For help with generating random integers, see the Random class in the Java API and specifically the nextInt() method.
- To make your results more accurate, **do not run anything else on your machine while you are doing your runs.** Don't worry about system processes that are running – just make sure you don't run any other applications.
- To make your results consistent, **do all of your runs on the same machine under the same (if possible) circumstances.**
- Note that for smaller arrays and in some cases even for larger arrays the time for a given trial may be very small – perhaps even negligible.
- Be sure to time only the actual sorting procedure – do not time loading the data into the array or any I/O (especially not I/O – this is very slow and will skew the timing greatly).
- As we discussed with some class handouts, in some cases a recursive algorithm makes so many calls that it uses up all of the memory in the run-time stack, causing the JRE to crash. To prevent this problem we can invoke the Java interpreter with a flag to indicate the size of the run-time stack. This can be done in the following way:

```
prompt> java -Xss<size> MainClassName
```

- You may have to experiment with the value for <size> to avoid getting a StackOverflowError, but in my runs using 10M worked with all of my runs. If you think about how these algorithms execute, you will see that we only have to worry about the stack size for the Simple QuickSort algorithm in the cases of the sorted and reverse sorted data.

Extra Credit:

- For comparison purposes, add some other sorting algorithms to your tests to see how they compare to Quicksort. For example, you could include Mergesort, Shellsort or Insertionsort.