CS 0445 Fall 2020 Assignment 4

Online: Sunday, November 1, 2020
Due: All files (see details below) submitted in a single .zip file to the proper directory in the submission site by **11:59PM on Tuesday, November 17, 2020.**
Late Due Date: 11:59PM on Friday, November 20, 2020

Purpose and Goal: Now that we have looked at Mergesort and Quicksort (and discussed several QuickSort 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 4 different sorting algorithms:

1) Simple Quicksort with A[last] as the pivot (as in file Quick.java)
2) Median of 3 Quicksort (as in file TextMergeQuick.java)
3) Random Pivot Quicksort as discussed in lecture
4) MergeSort (as in file TextMergeQuick.java)

The code for algorithms 1,2 and 4 is already written – however, you will need to modify it to handle the base case options as discussed below. 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]. **Be careful to choose the pivot in the correct range** – check this before running your timing program.

Class SortAlgorithms:
For this project you will write a class called SortAlgorithms, with the following specifications:

1) It will contain the 4 sorting methods stated above, modified as follows:
   Rather than going to a base case of 1, each method will stop recursing when the current array size is less than MIN_SIZE. This is already done in the author's Median of Three Quicksort, but you must modify the other sorts to have the same behavior. When the size is less than MIN_SIZE, you should complete the sort using InsertionSort, as is done for the author's Median of Three QuickSort.
2) Rather than a constant, MIN_SIZE will be a static variable that all 4 of your sorting algorithms will access. You must include a method called `setMinSize(int N)` in your SortAlgorithms class which will allow the value of MIN_SIZE to be set to the argument value.

Class Assig4:
Once you have all of your sorts properly implemented in your SortAlgorithms class, you will write a main program that will enable the user to time all 4 of the algorithms under different circumstances in an automated way.

Input and Variable Setup:
Your Assig4 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. Using "the same random data" for all of the different algorithms will make your results more accurate.

3) The name of the file your results will be output to

For each algorithm your program should consider 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.

Executing the Trials:
Your main program execution will have several nested loops in order to calculate several different results in a single run. Specifically:

```
For each MIN_SIZE in { 5, 50, 150 }
      For each of the 3 data setups
            For each of your 4 sorting algorithms
                  Time 10 trials and average and output the results
```

Trials and 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 should already be familiar with using System.nanoTime() from your recitation exercises and Assignment 2. 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 MIN_SIZE values, 3 data setups and 4 sorting algorithms, **each single execution of your main program should produce 36 different results.** Each of the 36 results should look something like the following example:

```
Algorithm: Simple QuickSort
Array Size: 25000
Base Case Less Than: 5
Data Setup: Random
Number of trials: 10
Average Time per trial: 0.0063856 sec.
```

Note: There is no required order that your 36 results must be produced within an execution of the program. You may find it more convenient based on how your loops are organized to produce your output in a particular way. The important issue is that all of your results are produced and are valid.

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
Base case size
Data setup (sorted, reverse sorted, random)
Initial data in array prior to sorting (list the actual values)
Data in array after sorting (list the actual values)

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 and with the different base case values. 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 in the sorted and reverse sorted cases and you will have to increase the stack size of the JRE to accommodate the execution – see below. For the sizes below you will have results only for the last 3 sorting algorithms:

Size = 200000, Filename = test200k.txt
Size = 400000, Filename = test400k.txt
Size = 800000, Filename = test800k.txt
Size = 1600000, Filename = test1600k.txt
Size = 3200000, Filename = test3200k.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
```

Check soon for an example output file.

Results: After you have finished all of your runs, tabulate your results in an Excel spreadsheet. Use a different worksheet for each initial ordering and MIN_SIZE combination ([random, 5], [sorted, 5], [reverse sorted,5], [random, 50], [sorted, 50], [reverse sorted, 50], [random, 150], [sorted, 150], [reverse sorted, 150]). Thus, you should have 9 total worksheets. 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, answer each of the following questions:

1) Which algorithm was best for random data?
2) Which algorithm was best for sorted data?
3) Which algorithm was best for reverse sorted data?
4) Which MIN_SIZE value gave the best performance and which gave the worst?
5) Based on all of your results, if you had to pick one algorithm, which algorithm would you choose?

Your write-up should be well written and justified by your results. Refer to your tables and graphs in your write-up. Your write-up should be 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, your Excel file and Word file**. As usual, put all of these files into **a single .zip file** for submission.

Additional Important 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. Think about why this is the case (consider our analysis for Simple QuickSort in the worst case and what that would require).

Extra Credit:

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