EECS 233 Programming Assignment #4

Due April 21, 2016 (11:59pm EST)
100 points

In this assignment, I'd like you to get a taste of experimental system research. When you design or choose an algorithm for a problem at hand, besides theoretical analysis, it is often important to implement and measure the actual performance of the alternatives. There are multiple reasons why the theoretical analysis might not suffice. Sometimes, it focuses on asymptotic bounds while in your practical application the problem size is such that constants and dominated terms matter. Or, as we have seen multiple times in class, the performance depends on the input data characteristics. In these situations, to make a case for a newly invented mechanism, you'd often prototype it and study its performance. The more realistic the conditions under which you perform this study the more compelling case you will build for your invention.

In this assignment, you will implement and study the performance of three sorting algorithms: heap-sort, quick-sort, and merge-sort. Each method below takes as arguments an integer array, which then becomes sorted in the increasing order of elements (each method returns the execution time of the corresponding sorting algorithm):

- long heapSort(int [] arr).
- long quickSort(int [] arr). (Your code in this method must include pivot selection.)
- long mergeSort(int [] arr). **Note**: Our implementation of merge-sort in the lecture notes involved merging pairs of sub-arrays into a temporary array and then copying the merged array of double size back into the original place in the input array at the end of each merging phase – see slide 14 of Lecture 19. Please write your implementation in a way that avoids this back copying (you still need the temporary array though). (**Hint1**: implement the merge-sort without recursion, and take care of the arrays whose size is not a power of two. **Hint2**: You alternate between merging chunks in the input array into chunks in the temporary array, and then merging chunks in the temp array back into the input array. Make sure that in the end your final sorted sequence of numbers resides in the input array and not the temporary array).

Please make these methods static and organize them into a Sorting class (as you would often do with utility functions – we discussed this in class when we started our module on sorting). After implementing the methods and testing them for correctness, your research begins. You need to write a test program to obtain the running times of the three methods, and compare them (see below some tips on measuring running time of your code). Perform your comparison on the following arrays:

1. Sorted arrays (in the increasing order, so the array will be unchanged after the execution of each method)
2. Reverse-sorted arrays
3. Randomly generated input arrays (you can find libraries to generate random integers in Java). Since random arrays may result in some corner cases in terms of algorithm performance, and also because running the same program even on the same input can take different time due to random scheduling and parallel activities, obtain each measurement in this step multiple times (e.g., 10 times) using different randomly generated arrays each time (use different seeds for your random number generators) and use the averages and variances of all your measurements for this step in your report. This is useful to show that your results are not a function of some lucky input data.

The size of input array should be varied in a wide range. To be specific, for this project, measure your executions for arrays of size 1000, 10,000, 100,000, 1,000,000. Please investigate the comparative performance as you vary the size of the input. Note that while repeating your measurements for different random arrays in step 3 will take care of any skew both from corner-case input and from random program scheduling, you should take repeated runs over the same array in steps 1 and 2. In these two steps, just repeating each run 3 times and taking the median running time will eliminate many outlier measurements.

You can see how quickly collecting measurements becomes too tedious to do manually; that's why we normally write special testing scripts and programs that automate these tasks – **as you should in this assignment to get full**

credit. Specifically, you should have, in your deliverables, two classes with "main" functions: Reporting1.java and Reporting2.java. Please also include them in the compiled form: Reporting1.class and Reporting2.class.

The **"Reporting1"** program generates all your arrays for sorting, runs each of the sorting methods as many times as specified in the assignment, giving them the arrays as arguments, records the execution time of these methods, prints out the results in a form suitable for inclusion into your report and/or writes out the results into a file if you need to post-process your results (e.g., by Excel to produce graphs for your report). If you do not need this output file for your report, you do not have to write it.

If you do produce all your results in their final form from the Reporting1 program (rather than writing them out for excel post-processing), you will need to compute a variance within your Reporting1 program. This is actually straightforward. In case someone has not yet taken a statistics class, if you have a sample of n measurements of a random quantity X, $x_1, \ldots, x_n$, then the variance V is computed as follows:

$$V = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2,$$

where $\bar{x}$ is the mean (average) value of the sample $x_1, \ldots, x_n$. So, it's a simple loop to compute this in your java program. Please have two helper methods: double meanVal(int arrayOfSamples) and double varianceVal(int arrayOfSamples, double mean). The meanVal method accepts the array of samples (in your case it will be 10 values) and returns the mean value of the samples. The varianceVal method accepts the array of samples and returns the variance.

The **"Reporting2"** program is to test your implementation of the sorting methods. We will be testing your work by typing

"java Reporting2 <input_file>"

on the command line, for example, "java Reporting2 inFile.txt". This program must accept the name of an input file as an argument (we will create our own file for grading) and, for each of the three runs of each of the three sorting methods, will read input_file into an array, then invoke the needed sorting method. For the first of the runs of each method, your program will also write out the output file with the name that starts with your case ID and append to it the abbreviation of the method name. Thus, since my case ID is rmk4, my program would produce three files: rmk4HS.txt, rmk4QS.txt, and rmk4MS.txt. (Just hard-code these names within your program as string literals.) Finally, your program must print out the results in the form: "HS<caseID> = time; QS<caseID> = time; MS<caseID> = time;" where <caseID> is your case ID. For instance, my program might produce the output line (do not take these numbers as any indication of what your numbers might look like – I just took these from the thin air; also indicate the time unite, e.g., "ns" for nanoseconds)

"HS rmk4 = 3682ns; QS rmk4 = 3456ns; MS rmk4 = 4320ns".

File format: the input file we will be using to test your work will have one integer number per line so you can just read it line by line and populate your array in memory. **Please make sure to use this format in your program.** The three files you will be producing must have the same format - one number per line (but of course in the sorted order - from small numbers to large).

Write a short report to summarize your findings and back them up using tables or graphs.

Deliverables:

- All source codes, including the sorting methods, all main() functions, and any scripts you wrote for measurements. Make sure your programs include the timestamp statements so that we could see what you are measuring. **Hint: make sure your time measurements include as little extraneous processing as possible, especially avoid expensive operations such as I/O. In particular, make sure you are not measuring reading any files from disk into the input array or printing any output.** See below for timestamping details.
- The two harness classes, Reporting1 and Reporting2.

- Report with tables, graphs, etc. as necessary to draw conclusions from your experiments. This should be a formatted MS Word or PDF document. You need to produce a readable report that contrasts the performance of the three methods. I leave it to your discretion the details, but we will grade it based on its readability. In particular, your report should include median execution times for the sorted and reverse-sorted arrays and means and variances for the random arrays (as mentioned in the instructions). Please include explanations of how you produced your results in your Reporting1.java class: did you write your data out on file and used excel for post-processing etc. However, **do not pad your report** – there is no target length as long as the report conveys the needed information.

Grading:

- *Completeness, correctness, and clarity of the code implementing the three sorting methods: 50%*
    - QuickSort: 10%
    - HeapSort: 15%
    - MergeSort: 25% (including 10% for avoiding back-copying on each iteration).
- *Correctness, clarity, and completeness of the testing code (automatic execution of all required runs, proper timestamping for performance measurement): 20%*
- *Completeness and clarity of the report: 30%.*

  Make sure you include enough comments in your code for the TAs to understand it well. Important: your code must compile and your main() functions must run!!

Tips on measuring time in Java:

There are a number of ways to measure time in Java. See this link: http://www.mkyong.com/java/how-do-get-time-in-milliseconds-in-java/ for examples to take a timestamp in milliseconds.

Note that you do need to measure time at least at millisecond granularity if you hope to distinguish the performance of your methods without using humongous input arrays. If you need to measure smaller values of elapsed time, you can use System.nanoTime() call:

```
long startTime = System.nanoTime();
// ... the code being measured ...
long estimatedTime = System.nanoTime() - startTime;
```