Benchmarking Sorting Algorithms

Summary: In this homework, you will be implementing code to generate test data and benchmark sorting.

1 Background

Learning Objectives:

- LO1: Gathers data to support and refute ideas (EM@FSE C)
- LO2: Propose a tentative model for an algorithm's performance.
- LO3: Develop code to produce sample inputs to evaluate a sorting algorithm.
- LO4: Develop code to evaluate a sorting algorithm and empirically estimate a Big-Oh run-time.

A typical problem with using an existing solution to an algorithmic problem is needing to understand its behavior. To make matters more difficult, behavior may depend on the input's structure (e.g., insertion sort's linear time performance on nearly sorted data). Algorithms may be very complicated, or we may not have access to the source code, in either case, the algorithm may need to be treated as a black box, whose behavior is unknown. To understand it's behavior, we need to perform an empirical evaluation (benchmarking). This problem often occurs in the context of a software development team. For example, a team may receive a piece of legacy software that is not documented properly (e.g., this algorithm performs in $O(n^2)$ on data that looks like...), or they may be asked to work with a blackbox solution from another team or a 3rd party. In these cases, a software developer on the team would be asked to analyze/quantify the performance of the algorithm. The team can be thought of as a customer, who the developer is trying to serve by providing knowledge on the internal solutions that the team can leverage to achieve their larger goal.

For this homework, you will start by manually predicting a performance of two sorting algorithms on three types of input (by considering the algorithms and the input data format). This produces an initial "model" for how the algorithm will behave. Note that rather than producing a model from initial sample data like in empirical analysis, we are "seeding" the understanding of these novel algorithm/data scenarios with our existing understanding for other inputs (e.g., we understand that insertion sort on nearly ordered data is linear time). The next step will be to evaluate the algorithm on the input data, and see what is the result in practice. This produces two things: 1) We are able to evaluate and improve our own intuition (was the way we thought it would work correct?). 2) It produces a model for how the algorithm behaves. The first aspect is the the reason why we want to make our own prediction instead of jumping directly to benchmarking. (Consider: the more accurate our mental models are, the faster we can find solutions to problems. A side effect of solving a problem is becoming better at problem-solving in general.)

To perform the benchmarking to validate our initial "guess" for performance, we will create three different types of input data, that may give different results when sorted. The two sorting algorithms will then be benchmarked on these three types of data. Each algorithm will be run twice, for different dataset sizes, in order to get times that we can use to apply the doubling formula. (see slide 23: Modeling Small Datasets) in the Analysis of Algorithms slide deck for details on the doubling formula.) The doubling formula is $lg\frac{T(2N)}{T(N)} = b$ where lg indicates a base2 log. If we compute the formula, then we will be able to figure out the algorithm's Big-Oh for a particular type of input data, since they will be $O(n^b)$. b is simply the power.

This document is separated into four sections: Background, Requirements, Testing, and Submission. You have almost finished reading the Background section already. In Requirements, we will discuss what is expected of you in this homework. In Testing, we suggest some basic tests you use to start to verify your implementation. Lastly, Submission discusses how your source code should be submitted on Canvas/Gradescope.

2 Requirements [36 points total]

For this assignment you will be writing code to generate test data and benchmark sorting algorithms on it (edited from Sedgewick and Wayne: 2.1.36). Already provided for you is CompletedBenchmarkTool.java which contains the sorting algorithms: insertion sort and shellsort. As usual, it's not complete just yet. This class extends an interface called BenchmarkTool which defines all of the methods that you need to create. You may create additional private helper methods. Before writing any code, you should write up your hypotheses (3 per algorithm) to describe what you think the running time will look like on each algorithm/input combination (see Writeup below). Your first step to benchmarking the algorithms will be to write a series of methods to generate test data that is "non-uniform":

- Implement the method *public Integer[] generateTestDataBinary(int size).* Half the data is 0s, half 1s. For example, an input of length 8 might look like [0, 0, 0, 0, 1, 1, 1, 1]. See the interface. [4 points]
- Implement the method *public Integer[] generateTestDataHalfs(int size).* Half the data is 0s, half the remainder is 1s, half the reminder is 2s, half the reminder is 3s, and so forth. For example, an input of length 8 might look like [0, 0, 0, 0, 1, 1, 2, 3]. See the interface. [6 points]
- Implement the method *public Integer[] generateTestDataHalfRandom(int size).* Half the data is 0s, half random int values (use nextInt() from Java's Random package, and use Integer.MAX_VALUE as its argument to ensure the values are positive). For example, an input of length 8 might look like [0, 0, 0, 0, 54119567, 4968, 650736346, 138617093]. See the interface. [4 points]

Each of these three techniques should be implemented as a method that takes an int representing the size of a dataset, and returns an Integer array containing that number of elements generated with the corresponding rule. Do not randomize (shuffle) the contents of the generated arrays. You may assume that only arrays with a power 2 length will need to be created.

For each of the sorting algorithms, your program should run them on the three types of test data. Test them with datasets size of 4096 and 4096*2. (If your system is so fast that you don't get good results, you may increase the dataset size. If your system continues to generate strange values even with a large dataset size, try turning off compiler optimizations.) Time each of the tests with the Stopwatch class discussed in class. The program needs to compute the result of the doubling formula on the run times from the sample data generated to get the power ("b") for that algorithm on that type of input, and then display it. Six different values should be shown if you have properly implemented all of the benchmarks.

- Implement the method *public double computeDoublingFormula(double t1, double t2).* See the interface for more information. [2 points]
- Implement the method *public double benchmarkInsertionSort(Integer[] small, Integer[] large).* See the interface for more information. [2 points]
- Implement the method *public double benchmarkShellsort(Integer[] small, Integer[] large).* See the interface for more information. [2 points]
- Implement the method *public void runBenchmarks(int size).* Whitespace is flexible (any number of tabs or spaces) but you must show three decimal places. See the interface for more information. Hint: should call the two benchmark methods above. The output should look like below. [4 points]

```
          Insertion     Shellsort
Bin         #.###         #.###
Half        #.###         #.###
RanInt      #.###         #.###
```

2.1 Packages

Do not import any packages other than java.text.DecimalFormat or java.util.Random. (Do not use any star imports.)

Required Files	Optional Files
CompletedBenchmarkTool.java	(none)

Table 1: Submission ZIP file contents.

2.2 Writeup [12 points]

Using the code you implemented, test and evaluate hypotheses about the effect of input on the performance of insertion sort and shellsort. In a separate document (to be submitted as a PDF; typically two pages double spaced), discuss:

- **Your hypothesis:** Write up your hypotheses (3 per algorithm): describe what you think the running time will look like ($O(n)$? $O(n^2)$? $O(n^3)$?) on each data set, and explain briefly why you think that. As long as your ideas make sense, **and you do the analysis prior to benchmarking**, you will receive full credit on the hypotheses. [4 points]
- **Benchmark Results:** Include the data (these are really the power on n) that your program outputs. If your results are strange, please explain possible issues (see disadvantages of empirical analysis). [1 points]
- **Evaluation/Validation:** Compare and contrast your hypothesis for the run-time with the data that you generated. What was supported? What is still unclear? (Your hypothesis and your result will probably be different!) [4 points]
- **Scenario:** Suppose that you are on a team that was given a legacy codebase that uses shellsort sort to process an array with half zeros, half random data. The legacy documentation indicates that shellsort is the best choice but you are asked to investigate and confirm. Based on your findings above, would you recommend, for better performance, that the team to stay with shellsort or switch to insertion sort? Multiple answers are possible but must be logical. (If you do not feel you have enough information to make a decision, state that, and justify.) [3 points]

3 Testing

The main functionality to test is the methods that generate test data. You will want to run them multiple times, on different sizes, and display their output. Check that the output matches the patterns required above. There isn't much else to test for this homework. The algorithms you are benchmarking have already been tested for correctness. Optionally, you may want to try giving an input that is pure random numbers to each of the algorithms and checking if your doubling formula code gives the algorithms expected Big-Oh.

Please note that some Gradescope grading tests may give "strange" output. Remember that different computers will give different for this time of assignment, so even if you see expected values, they may not make any sense to you when working locally.

4 Submission

The submission for this assignment has two parts: a written hypothesis document, and a source code submission.

Writeup: Submit a PDF that discusses your hypotheses. Include a header that contains your name, the class, and the assignment. We not are expecting anything longer than two pages. (If it makes your writeup more clear, you may use additional pages.)

Source Code: The source file must be named as "CompletedBenchmarkTool.java", and then added to a ZIP file (which can be called anything). The class must be in the "edu.ser222.m02_01" package, as already done in the provided base file (do not change it!). You will submit the ZIP on Gradescope.

4.1 Gradescope

GRADESCOPE IS EXPERIMENTAL FOR FALL 2021, ITS USE IS SUBJECT TO CHANGE. IF IT BREAKS, TELL US.

This assignment will be graded using the Gradescope platform. Gradescope enables cloud-based assessment of your programming assignments. Our implementation of Gradescope works by downloading your assignment to a virtual machine in the cloud, running a suite of test cases, and then computing a tentative grade for your assignment. A few key points:

- **Grades computed after uploading a submission to Gradescope are NOT FINAL.** We have final say over the grade, and may adjust it upwards or downwards.
- **Additional information on the test cases used for grading is not available, all the information you need (plus some commonsense and attention to detail) is provided in the assignment specification.** Note that each test case will show a small hint in its title about what it is testing that can help you to target what needs to be investigated.

If you have a hard time passing a test case in Gradescope, you should consider if you have made any additional assumptions during development that were not listed in the assignment, and then try to make your submission more general to remove those assumptions.

Protip: the Gradescope tests should be seen as a way to get immediate feedback on your program. This helps you both to make sure you are meeting the assignments requirements, and to check that you are applying your knowledge correctly. Food for thought: if you start on the assignment early, check against our suite often, and use its feedback, there's no reason why you can't both get full credit and know that you'll get full credit even before the deadline.

4.1.1 Standard Programming Deductions

Due to the nature of cloud grading, we use a different policy for the standard deductions:

- **If your submission does not compile, or is missing the file mentioned above, you will receive a zero grade.**
- **Following the file submission standards (e.g., the submission contains project files, lacks a proper header) is optional, and will not be enforced.** (We would appreciate if you at least included the header though!)