COSC 223: PROBABILITY AND COMPUTING
PROJECT 1: QUICKSORT

Due Monday, April 12, 11:59pm ET

1 Find a Partner: Due Thursday, March 25, 11:59pm

This is a partner project: you will work with one other student in the class, and you may choose your partner. Your first task is to find a partner and let me know with whom you will be working. By **Thursday, March 25, 11:59pm** you and your partner must email me to let me know that you're working together (send me one email, copying your partner). If you haven't found someone to work with by Thursday night, email me to let me know and I will find a partner for you.

2 Intellectual Responsibility

You will be working with a partner on this project and all aspects of the project can (and should) be done together. Do not share code or written work with anyone besides your partner, and do not look on the internet for any solutions. You may discuss the concepts involved in this project with other students in the class who are not your partner; if you do, please note at the top of your writeup with whom you consulted, and what you discussed.

3 Submission

There are three different components to submit for this project (see Section 4 for details):

1. All code that you wrote for this project
2. A `README` file explaining how your code works and how I can run it
3. A "research paper" writeup describing your experiments and results

You'll submit all of these files via Gradescope. Please do NOT zip your files before submitting them. This assignment is due Monday, April 12 by 11:59pm ET.

4 Your Tasks

In this project, you will study the Quicksort sorting algorithm. Specifically, your goal is to investigate the number of comparisons between array elements that are made over the course of the algorithm's execution.

While working on this assignment, you will:

- Implement both a deterministic and a randomized version of the Quicksort algorithm

- Observe the differences between randomization over inputs for a deterministic algorithm, and randomization over the algorithm's choices.
- Compare the empirical performance of Quicksort to the analysis that we'll do in class.
- Design and evaluate new ways to improve the performance of Quicksort.

4.1 Implementation and Experiments

You aren't required to use any particular programming language for your simulator, with the caveat that the experiments will require you to initialize the array using some Java code that I've provided. If you choose to work in a language other than Java, you'll need to do a little extra work to integrate my Java code with your code.

You will need to implement two versions of quicksort. For **randomized quicksort**, your partition element should be chosen randomly within the range `lo` to `hi`, with each element equally likely to be chosen. For **deterministic quicksort**, you should always use the element at position `hi` (i.e., the last element in the range you're sorting) as the partition element. I recommend implementing the version of the `partition` method that we discussed in class, because this will make your next job a little easier.

Your goal in this project is to assess the runtime of both deterministic and randomized quicksort on different kinds of inputs. For our purposes, we'll define the "runtime" to be the number of times two elements in the array are compared to each other. Add some code to your quicksort implementation to count the number of such comparisons that occur during a single run of the quicksort algorithm. **Be sure to reset your counter** every time you run quicksort on a new array; if you do not reset your counter, your results will not turn out the way they should.

You'll consider three different kinds of inputs in this project: arrays that are "uniformly" randomized (i.e., any ordering of the array elements is equally likely), arrays that are "partially sorted," and arrays that are "mostly sorted." The `Generate.java` program, which you can download at `http://bit.ly/cosc-223-s21-project1`, contains three methods called `generateRandomInput(n)`, `generatePartiallySortedInput(n)`, and `generateMostlySortedInput(n)` that take an array size, `n`, as input and return an array of the specified size containing elements with the corresponding ordering. If you're working in Java you can call these methods directly from your code; if you're working in another language you may want to adapt `Generate.java` to, e.g., write the generated array contents to a file so that you can read it into your program.

4.2 Research Questions

At a high level, your goal in this assignment is to answer four fairly broad questions about designing, analyzing, and understanding randomized algorithms. These questions are:

1. What's the difference between randomization over the input and randomization over the algorithm's behavior?

2. What do we learn from analyzing an algorithm's expected performance, and what is missing?

3. What are the potential advantages of using a randomized algorithm rather than a deterministic algorithm?

4. How can we use the insights from the comparison of deterministic and randomized quicksort to come up with further improvements to the algorithm's performance?

To answer these questions, you'll conduct a series of experiments to evaluate the performance of the quicksort algorithm on different types of inputs.

1. First, study the runtime of deterministic quicksort by creating a graph that plots the average runtime as a function of n (the number of elements in the array). Your n's should get fairly high (up to $n = 10^6$ is reasonable, but you may find that you need to stop at a smaller value for some of the input types). To compute the average runtime for a particular value of n, you'll need to: (1) initialize a new array of size n, using one of the three array-generation methods I've provided; (2) run quicksort on the array, recording the number of comparisons executed during the run; (3) repeat 1 and 2 many times, and take the average. Do this for each of the three array-generation methods, and plot all three lines on the same graph. You should also include a line for the runtime bound for randomized quicksort that we'll derive in class.

2. Do the same thing, this time for randomized quicksort.

3. The average runtime gives us some information, but it's not always the full story. Next, we'll explore the variance of runtime under each configuration (both deterministic and randomized quicksort, and all three array generators). You might want to start by plotting a histogram of runtimes, for a few values of n. (Think about how many times you need to repeat the experiment, for a given n, in order for the histogram to give you meaningful information). For each configuration, pick a (reasonably large) n and compute the *empirical variance* as follows:
$$\text{Var} = \frac{\sum_{i=1}^{m} x_i - \bar{x}}{m-1},$$
where m is the number of times you ran the experiment, x_i is the runtime of the ith run, and $\bar{x}$ is the average runtime across the m runs.

4. One disadvantage of the variance as a performance metric is that it is sensitive to the mean: if we have two distributions that have the same "shape," but one is shifted to have a higher mean, the version with the higher mean will also have a higher variance. This is sometimes undesirable, so we might also consider a "normalized" version of variance, called the *squared coefficient of variation*, or C^2. The squared coefficient of a random variable X is
$$C_X^2 = \frac{Var(X)}{E[X]^2}.$$

Compute the squared coefficient for the same n's for which you computed the variance.

5. Do you think you can further improve on the process by which quicksort selects the partition element? Come up with an idea, implement it, design and run some experiments to evaluate your idea (on both randomized and determinisic quicksort, and all three input configuations). It's ok if your idea turns out to not work so well after all, once you've tested it!

5 Deliverables

You'll submit three things for this assignment:

1. All of the code that you've written for this project
2. A `README` file explaining how your code works and how I can run it. The main purpose of this document is to enable me to use your code. You should explain what command to type to run your code (and what, if any, command line arguments are needed) and explain the input, output, and purpose of each method. You do not need to describe line-by-line what your code is doing.
3. A writeup of your experiments, results, and what you've learned about quicksort from your results (see Section 4.1)

I will also ask you, after the assignment is complete, to submit a short reflection on your collaborative process while working with your partner on this project.

5.1 The Writeup

The goal of your writeup is to provide a complete description of what you did in your project, why you did it, and what you found. This will be structured similarly to a research paper; after all, your project was a mini research study.

Your writeup should include the following sections:

1. **Introduction.** Explain the problem that you are studying, why it's important, and the goal of your project.
2. **Experimental setup.** Describe the experiments that you ran. What size arrays did you use? How were the inputs generated? The purpose of this section is to give your readers enough information to enable them to replicate the exact same experiments you ran. In this project, I told you in Section 4.2 how to set up many of your experiments. But if you had designed the experiment yourself, you would need to explain to your audience exactly what you did. We're including this section in the writeup for this project to give you practice describing your experiments, even though I already know what you did. For your new idea for how to further improve quicksort, you'll need to give a detailed description of your new idea and how you evaluated it.

3. **Results and Discussion.** What happened in your experiments? Include graphs showing the data you collected in your experiments, and explain what patterns you see in the graphs. What insights into the runtime of quicksort do your results reveal? This section is where you provide answers to your research questions and provide some explanation of why the results look the way they do. We'll analyze the expected runtime of quicksort in class—do your experimental results match up well with what the analysis predicted would happen, or were you surprised by what happened in practice? If the analysis and the experiments differed, why might that have occurred? Was anything particularly surprising or particularly interesting about your results?
   Note: You may want to have one pair of "Experimental setup" and "Results and Discussion" sections for the initial experments, and then a second pair of sections for your new quicksort idea and experiments. This organization may make it easier for you to describe how the results of your intial experiments informed your ideas for the second part.

4. **Conclusion.** In this section you'll summarize the main findings of your experiments and discuss any limitations of your experiments or remaining questions that you might want to study.

There's no length requirement for the writeup; your goal should be for your writeup to convey all of the above information as clearly and thoroughly as possible, rather than to target a particular number of pages.