CS161: Introduction to Computer Science
Lab Assignment 11

Today you and your partner will be running empirical experiments to determine the best, worst, and average time complexity of three different algorithms: linear search, binary search, and a median finding algorithm.

Recall that for a searching or sorting algorithm, the *time complexity* is the number of comparisons performed as a function of the array size. In class, we determined that the best, average, and worst case running times for linear search are,

Best	**Average**	**Worst**
1 comparison	$\frac{n}{2}$ comparisons	n comparisons

where n is the size of the array. The best case scenario is if the value we are looking for is the first element in the array. The worst case scenario is if the value we are looking for does not occur in the array. The average case is the number of comparisons made averaged over lots of different scenarios: on average, we have to do about $n/2$ comparisons.

This lab is unusual because you will be generating plots and providing a **lab report** (see the Word document in the starter code)! At the end of this lab, please submit whatever you and your partner were able to accomplish.

Step 0: Familiarizing Yourself

Do not begin the steps below until you have read through each of the methods in the two Java classes provided!

- Read through the methods in the `ListSearcher` class.
- Read through the methods in the `PerformanceTester` class.

Make sure you understand what each class is doing, what the methods do, and what the instance variables represent. If you have any questions, call me or the lab assistant over.

Step 1: Counting Comparisons

You will notice that there is no code for actually counting the number of comparisons in the `ListSearcher` class! This is your first task.

1. Add an instance variable to the `ListSearch` class to keep track of the number of comparisons made.
2. Modify the `linearSearch()`, `binarySearch()`, and `findMedian()` methods so that they count how many comparisons are made
3. Finally, add the following accessor/mutator methods:
    - `public int getComparisons()` - This method should return the number of comparisons made
    - `public void resetComparisons()` - This method should reset the number of comparisons to 0.

You are now able to reset, count, and get the number of comparisons made.

Step 2: Running Experiments

Now that you can count comparisons, let's test to see if our analysis of the time complexity of linear and binary search are actually correct.

The `PerformanceTester` class has three static methods: `testLinearSearch`, `testBinarySearch`, and `testMedian`. This section will step you through experimenting with linear search. Once you learn how to do this, you can go back and do the same for binary search. (Ignore the median method for now.)

Look inside the `main` method. You'll see that we're calling `testLinearSearch` with an array of size N=1000, number of trials equal to 1, and unordered data. If you run the `main` method, you should see something similar to the following print out:

```
N=1000, best=333 (expected 1), worst=333 (expected 1000), avg=333 (expected 500)
```

Let's actually spend a little bit of time understanding the report we just generated above. Recall our hypothesis that the best case, worst case, and average case comparisons for linear search are 1, n, and $n/2$, respectively.

In the report printed out, the best, worst, and average number of comparisons are all the same because `linearSearch()` was only run once. As a result, the quality of these results is not very good, and as computer scientists, we know that results from a single experiment cannot be taken at face value! A good scientist would repeat the experiment multiple times to get a good understanding of what's really happening.

- Modify the input values to `testLinearSearch` to repeat the experiment 10 times. The report printed out should show values closer to the expected number of comparisons.
- Open up the Word document included in the starter code and answer question **Q1**.

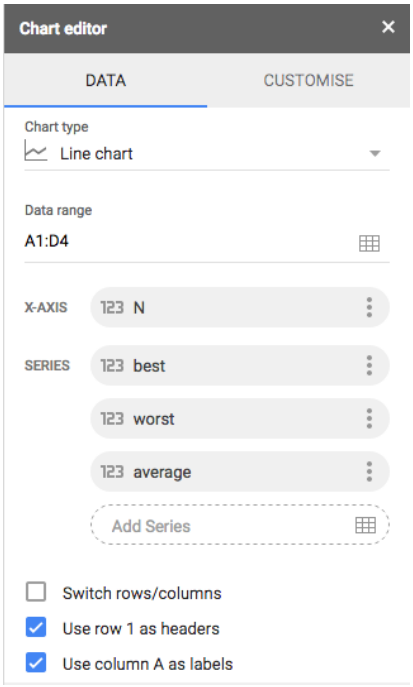
Generating Plots

It would be great to generate plots to better visualize our results. Open up your favorite spreadsheet tool, like Sheets on Google Docs or Excel. The following example shows how to generate charts using Sheets on Google Docs:

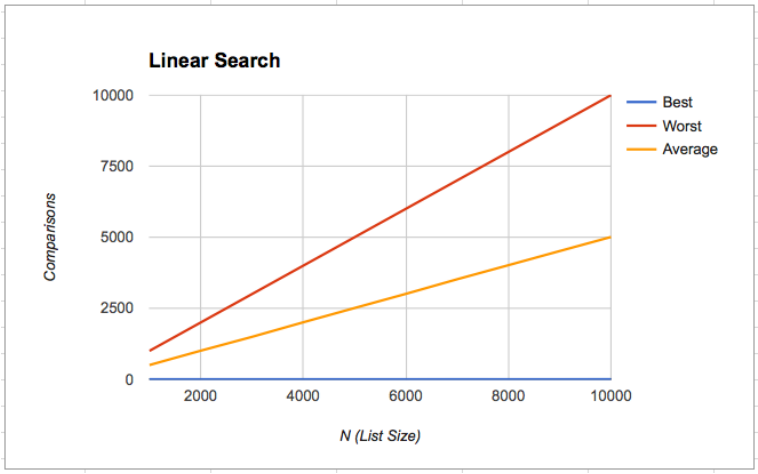
- In `main`, call `testLinearSearch` for $N = 1000, 2000, 3000, \ldots, 10000$ using the number of trials you found worked best in question **Q1**.
- Head over to Google Docs. One of you will need a Google ID to sign in. Once in, click on the striped-lines button on the top-left corner, and select Sheets. Next, click on the + button and it should generate a new blank spreadsheet. Let us know if you have troubles getting this far.
- After you've got a blank sheet in front of you, copy the numbers for N, best case, worst case, and average case as follows:

	A	B	C	D
1	**N**	**Best**	**Worst**	**Average**
2	1000	1	1000	502
3	2000	1	2000	1011
4	3000	1	3000	1507
5	4000	1	4000	2007
6	5000	2	5000	2509
7	6000	2	6000	2957
8	7000	2	6999	3550
9	8000	1	7997	4004
10	9000	2	9000	4510
11	10000	6	10000	5037

- Double-check that all of your data is entered correctly. Then highlight your data (including the first row containing labels). From the menu bar, click "Insert > Chart..." Click on the "Chart Types" tab at the top and select the first "Line" box to generate a line plot. Make sure there is a checkmark in "Use Row 1 as Headers" and "Use Column A as Labels".



- Click on the "Customise" tab at the top. Click on the "Chart & axis titles" drop down menu. Give your chart a title. Label your horizontal axis "N (List Size)" and your vertical axis "Comparisons". And with that, you're all done!
- Click on the chart, and in the top-right corner you should see three dots. Click on this and click "Save Image". This will download the chart as a PNG file. Open it and drag this chart into the appropriate place in your lab report under **Q2**.
- That's all there is! Go ahead and answer **Q3** in your lab report. This question asks you to experiment



with linear search using ordered (i.e. sorted) data.

Step 3: Binary Search

Alright, now do this all over again but this time with binary search. No, really...do it all over again with binary search.

Notice that `testBinarySearch` has no boolean flag to indicate whether it should generate ordered or unordered data because binary search requires an ordered array to work correctly. Run this method for $N = 1000, 2000, 3000, \ldots, 10000$. Generate a plot and add it to your report. Answer question **Q4**.

Analyzing a New Algorithm

This last section asks you to analyze a brand new algorithm: an algorithm to find the median value in an array. The *median* of a set of numbers is the number that is bigger than half the values and smaller than half the values. If the array was sorted, the median would be in the middle spot. Note: If the array has even length, there is no unique median. In this case, either of the middle values would work.

1. Open up the ListSearcher class and read the findMedian() code. This is a very naive algorithm for finding the median!
2. Make a guess for the best, worst, and average time complexity for this method. Record your guess in your lab report.
3. Run experiments by calling the `testMedian` method in the `PerformanceTester` class using $N = 100, 200, 300, \ldots, 1000$. If you run bigger values, it will take a long time to complete. Try to run as many trials as possible.
4. Open your spreadsheet program, and generate another plot, with the best, worst, and average-case. Insert the plot in your Lab Report, and answer the final question.

Submitting your lab assignment

Submit your `lab11` folder with both Java classes and your Word lab report inside. Remember to put both of your names on the folder *before* you zip it. Submit via Moodle.