CSC 120: Lab 6 – It's Sort of Efficient
Due Wednesday, October 15, 2025 at 3pm

Objectives

- To learn more about Big-O notation from a pragmatic point of view

Your Mission

Your goal is to figure out the worst-case running time efficiency of two functions. They both do the same thing -- sort an array of integers in non-decreasing order. Here's the catch: you won't be able to read the source code. You'll have to empirically determine the running times by executing the code under different conditions and then analyzing the results.

This lab is more about doing thorough experiments and careful analysis than programming (though there is some coding you need to do).

Preliminary Setup

1. Download the starter code from Nexus and open it in PyCharm. Rename the project with your last name and the lab number.

2. There are two files: *main.py* and *sorts.py*. The *sorts.py* file has had its contents **obscured**, meaning the text of the file has been encoded to make it difficult to read. It actually contains two functions named `sort1` and `sort2`. Don't change anything in this file. You'll be working with *main.py* instead. **PITFALL ALERT**: PyCharm might complain about importing sorts or calls to `sort1` and `sort2` by putting a red line under them in *main.py*. Ignore it. It'll still run.

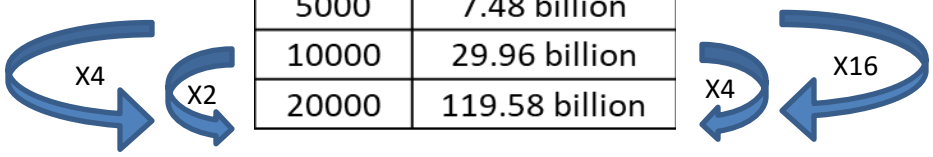
3. Fill in the *:author* tag at the top of *main.py* with your name.

The Big Idea

Here's the gist of what you're doing in this lab: you need to find the worst-case running times of `sort1` and `sort2` and express each in Big-O notation, even though you can't see the code for them. So you'll need to figure it out by running each function with different amounts of input and watching what happens with the number of basic operations (in this case, the number of times that the list of input is accessed). That number will print each time you run a sort function. You can then make a table similar to the one on the next page that maps the amount of input (i.e. the length of the list of numbers you're trying to sort) vs. the number of basic operations (i.e. the times that the list was accessed).

For example, you might call a sort function with 100 numbers to sort, then 200, then 500, etc. to see how the number of list accesses increases each time. Your table would look like this:

list size	# of list accesses
100	2,856,408
200	11,802,978
500	73,672,194
1000	298,925,730
2000	1.19 billion
2250	1.50 billion
3000	2.70 billion
5000	7.48 billion
10000	29.96 billion
20000	119.58 billion



You can then analyze the table in the following way. When the number of inputs doubles from 10000 to 20000, the number of accesses quadruples (~30B → ~120B). So 2*the input results in 4*the list accesses. Similarly, when the list size quadruples (5000 → 20000), the number of accesses increases by a factor of 16 (~7.5B → ~120B). So 4* the input results in 16* the list accesses. This is evidence of an $O(n^2)$ algorithm because in such an algorithm, when n becomes 2n, the amount of executed lines increases by a factor of 4:

$n^2 \rightarrow (2n)^2 = 4n^2$

and a 4-fold input increase results in a 16-fold increase in accesses:

$n^2 \rightarrow (4n)^2 = 16n^2$

and that's exactly the behavior we see in the chart above.

Step 1: Understand the Starter Code

The code in *main.py* is designed to make it easy for you to collect this data. The primary function is `run_sort`, which takes three parameters. Read the docstring so you know what those are. The default arguments are given in `main()`:

```
sort = sort1
kinds=["random","identical"]
lengths=[10,20,30]
```

This means that sort1 will be run with a list of 10 random numbers, then 20, and then 30 random numbers. Then it will be run with 10 identical numbers, then 20, and then 30 identical numbers. Why the different kinds of data? Because you don't know when the worst case is going to happen! It might come when sorting random numbers. Or already-sorted numbers.

Or reverse-sorted once you reach a list 100,000 long. The point is you don't know when the worst case happens, so you need to test under a lot of different conditions – various lengths and various kinds of data. You'll be changing these three lines to run your own tests, but not yet. First, run the code as-is so you can see the output. Each time a sort is run, it prints the sort function being used, the data kind, the list size, and the number of list accesses that occurred during the sort. This is the kind of data you'll be collecting.

Step 2: Improve the Starter Code

The other functions in *main.py* are private helpers to `run_sort`. The function `list_generator`, for example, is the one that fills the list with numbers to be sorted. But notice that only two kinds of data are currently available – random and identical. Fix that by adding code to create a sorted list and a reverse-sorted list of the given length. Test and debug until it works.

Step 3: Collect the Data

It's time. Adjust the three lines in `main()` to change the sort function, input size, and kind of data as you see fit. Record all output as tables in a separate text file. Have one table for random data, one for sorted data, etc. **You are required to try at least 10 different list sizes for each kind of data** in order to be reasonably sure that you can see the pattern. **PITFALL ALERT**: if you start waiting too long for output (more than a few minutes), try smaller list sizes. And one of the sorts uses recursion, which can be a memory hog. So if you get an out-of-memory error, then again, use smaller list sizes.

Step 4: Plot the Data

Produce graphs based on your data. For each sorting function, make a line graph (Google Sheets is fine) that plots the list length (on the horizontal axis) vs. the number of list accesses (on the vertical axis). Make one graph for each sort function, with all of the data variations on the same plot. Hints for using Google Sheets:

- After pasting the data, use Data menu → "Split text to columns" to auto-format.
- Highlight the numerical data and use Insert menu → "Chart" to get a chart. Smooth Line charts worked well for me. Customize as needed.

Step 5: Analyze the Data

Finally, analyze your raw data and graphs using the technique described in **The Big Idea** section above. Write a short lab report that shows your findings. For each sorting function, your report should

- Show which kind of data and plots led you to determine the worst-case running time.
- Explain what the worst-case time complexity is in Big-O notation (and explain how you reached that conclusion).

You should treat this lab like a physics or chemistry experiment in which you are taking measurements and making conjectures based on your measurements. Therefore a significant number of points will be taken off if you have not sufficiently tested the functions or analyzed your results. Make sure your plots are clear, well-labeled, and neat. The harder it is for me to understand what you did, the fewer points you will receive.

How to turn in this lab
Make sure your lab report has:

- the code you wrote for `list_generator`
- the tables you made of the raw data
- screenshots of your two graphs
- your analysis and conclusions in Big-O notation

Convert it all to a **single** pdf file and upload it to Nexus. You do not need to turn in the project folder this time.

But you should still backup your work.