Programming on Purpose (CSC 120)
Winter 2024

Lab 5 – It's Sort of Efficient

2/1/2024

Objectives

- To learn more about runtime analysis (from a pragmatic point of view)

Your Mission

Your goal is to analyze the efficiency of two Python functions. They both do the same thing – sort a list of integers in ascending order but they use different algorithms. Your job is to give each algorithm's ***worst-case*** runtime as a function of the length of the list that is being sorted. Here's the catch: I'm not giving you the source code. You'll have to empirically determine the running times by executing the already (semi-)compiled 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

Create a folder for this lab, download the starter code and save it in the folder. There are two starter files:

- `sorting.txt` contains a (not very human-readable) implementation of the two sorting functions. The functions are called `sort1` and `sort2`. **Do NOT modify this file.**
- `main.py` contains the starter code for running your experiments. You will have to modify the functions `list_generator` and `main`.

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 as a function of the length of the list to be sorted. For example, if n stands for the length of the list, an algorithm that has a running time of $f(n) = 3n$ runs in linear time. That means, if the length of the list doubles, the running time also doubles; if the length of the list triples, the running time also triples, etc.

You can't see the source code for `sort1` and `sort2`. So you'll need to figure out the running times by running each function with different amounts of input and watching what happens with the time the computer needs. (The starter code already is set up to measure and print out the time needed.)

You will end up with a table that maps the length of the input list to the time needed (in seconds). For example, you might call one of the sort function with 100 numbers to sort, then 200, then 500, etc. to see how the time increases as the list gets longer. You could then make a table like this:

list length	time (seconds)
100	0.0007731610094197094
200	0.0034392509842291474
500	0.02085858304053545
1000	0.08206656901165843
2000	0.3325291650253348
5000	2.055459338007495
10000	8.04670136095956
20000	31.67686974199023

You can then analyze the table: When the number of inputs double from 10000 to 20000, the time needed roughly quadruples ($8 \rightarrow 32$). When the list length quadruples ($5000 \rightarrow 20000$), the time increases by a factor of 16 ($2 \rightarrow 32$). This suggests that the running time of this algorithm on the given kind of data is described by the function $f(n) = n^2$.

1 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_experiment`, which takes three parameters. Read the docstring so you know what those are. The starter code gives the following default arguments 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 time taken. This is the kind of data you'll be collecting.

2 Step 2: Complete the Code

The other functions in `main.py` are private helpers to `run_sort_experiment`. 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.

3 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 in a separate text file (copy and paste from the terminal is ok). 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. So if you get an error that the maximum recursion depth was exceeded, also use smaller list sizes.

4 Step 4: Plot the Data

Produce graphs based on your data. For each sorting function, make a graph (Google Sheets is fine) that plots the list length (on the horizontal axis) vs. the time (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 the spreadsheet and chart as needed.

5 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 the appropriate data and plots that you used to determine the worst-case running time.
- Describe the worst-case running time as a function of the size of the input list and explain how you reached that conclusion.

There is very little coding in this lab. So I will primarily evaluate the quality of the data you collected and your report. Make sure to test the functions with a range of input sizes and all different kinds of lists. Make sure your plots are clear, well-labeled, and neat. Make sure your explanation is clear and easy to understand.

6 How to submit

Don't forget to cite who you received help from and include the honor code affirmation in the docstring of each module that you wrote or modified:

```
I affirm that I have carried out my academic endeavors
with full academic honesty. [Your Name]
```

Submit the project by uploading `main.py` and your lab report as a **pdf** file to Gradescope.