CS/SE 2XB3 — Lab 3

Due Friday, Feb 5th, 11:59pm

Submit the lab on Avenue. Note, Avenue must receive your lab by the due date. **Do not leave your submission to the last minute! Late submissions, even by seconds, will not be graded.** You have been warned.

This lab is worth (59/7)% of your final grade in the course. Read this document carefully and completely. **Whenever I ask you to discuss something, I implicitly mean to include that discussion in your lab report.** Include all timing experiments in your `code.py` file.

Purpose

The goals of this lab are as follows:

1. Implement Quicksort and variations of it.
2. Empirically analyze the performance of your implementations.
3. Based off your experiments design an "optimized" sorting algorithm

Submission

You will submit your lab via Avenue. You will submit your lab as three separate files:

- code.py
- sorts.py
- report.pdf (or docx, etc.)

Note, this is different from the previous labs where you submitted it as a .zip. The TAs have requested labs be submitted this way to make grading easier for them. Only one member of your lab group will submit to Avenue – see the section below for more details on that. Your report .pdf will contain all information regarding your group members, i.e. name, students number, McMaster email, and enrolled lab section. This will be given on the title page of the report. For the remainder of this document, read carefully to gauge what other material is required in the report. Your report should be professional and free from egregious grammar, spelling, and formatting errors. Moreover, all graphs/figures in your report should be professional and clear. You may lose grades if this is not the case. Organize the report in a such a way to make things easy for the TA to grade. You are not doing yourself any favors by making your report difficult to mark!

Quicksort [100%]

I am assuming you have seen quicksort in your other courses, having talked to the two 2C03 professors, this should be the case. Otherwise, do some independent research on what quicksort is. Understanding how it works is relatively straight forward.

In-Place Version [20%]

See the lab3.py file given on Avenue alongside this document. Here I have written a reasonable/traditional implementation of quicksort, called `my_quicksort()`. Write an "in-place" version of quicksort. Name this function `quicksort_inplace()` and include it in your sorts.py file. Your implementation should take in a list as input, and after terminating the list should be sorted. By in place it is meant that you only use the input list to store and move values. You should not be creating copies of the list, or creating other temporary lists. If you do, your implementation is not in-place and therefore will not receive grades.

Before testing the performance of your implementation, discuss any advantages it has over the implementation you are given. Now design and run some timing experiments (focus on average runtime here). You may wish to use the `create_random_list()` function here. Which implementation is better? By how much? Which would you use in practice?

Multi-Pivot [40%]

As you know, the traditional quicksort algorithm chooses a single pivot and "splits" the list/array into two components. Why can't we choose two pivots and split the list/array into three components? Well, we can. So let's do it. Implement a `dual_pivot_quicksort()` function in your sort.py file. Implement it in the style of `my_quicksort()` (so we can compare them later). Now implement a quicksort with three pivots, and (yes, you guessed it) one with four pivots. In your sort.py file name these functions `tri_pivot_quicksort()` and `quad_pivot_quicksort()`.

Design experiments to test the performance of all four quicksort variants (1, 2, 3 and 4 pivots). Focus on general average case performance. Which of the four variants do you recommend? For the remainder of the lab, use your recommendation for your default quicksort,

Worstcase Performance [20%]

What is the worst-case performance of quicksort? Run an experiment which graphs the average case performance vs the worst case performance vs n. We saw in lecture (and I have included it in lab3.py), a function which creates near-sorted-list based off some input *factor*. In lecture we also saw implementations/optimizations for three elementary sorts: bubble, selections, and insertion. When this factor is low, i.e. the list is close to sorted, what elementary sorts/optimization (if any) would you expect to out perform your quicksort implementation?

Design some experiment to explore this further. Focus your experiments on lists of length 1000. Graphs the runtime of your quicksort as well as a few of the best performing elementary sorts vs the near-sorted factor. At what value(s) of this factor does quicksort begin to out perform the other sorting algorithms (if it does at all)?

Small Lists [20%]

Revisit the performance of your quicksort implementation, but focus on small lists, i.e. low values of n. Compare this to the other (elementary) sorting algorithms. What observations do you make? Implement a version of quicksort which is optimized using the lessons learned in this lab. Your implementation may be a hybrid of several sorting algorithms, but it should fundamentally have elements of quicksort present. Name this sort `final_sort()` in your sort.py file. Discuss all design decisions you made and why. What goals were you trying to achieve? Minimized average runtime? Recursion depth? Worstcase? Some combination thereof? There is no *perfect* sorting algorithm, but your reasoning and justification must be valid.