Computer Laboratory 5

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

1 Essential information

- This assignment is due Tuesday October 18 at noon and will be turned in on gradescope
- The assignment is intended to be done in pairs, although you're allowed to work on your own if you want.
- You can start the assignment before your lab period, although I recommend not getting too far – the lab TAs often have useful hints, and I don't want you wasting too much time struggling with something that the TAs might announce in-class.
- For more rules see the lab rules document on canvas.
- **This lab has a substantial non-coding part. DO NOT FORGET TO DO THIS PART. DO NOT FORGET TO TURN THIS PART IN**

2 Introduction

In this lab we will get a little more hands-on experience with sorting functions, big-O runtime of these algorithms, and the trade-offs triggered by sorting different types of lists. We will do this by modifying sorting algorithms to count how many times they compare two elements in the input list. Comparing elements is often the most expensive operation in sorting data (especially when data is of a user-defined type, rather than a simple data type like numbers) therefore, by counting the number of times we perform a comparison, we are estimating the time cost.

IMPORTANT NOTE. DO NOT MISS This assignment has a drawn/plotted/graphical and short-answer component on-top of the python component. We are not looking for anything too crazy for these (you can hand draw the image component, we will not judge you on your artistic skills) Do not forget these components in your submission. We cannot grade what you forget to turn in, even if you know the answers to these questions.

This lab will:

- Give you practical experience with a way of practically determining runtime efficiency

- Let you practice thinking analytically about code.
- Strengthen your understanding of the sorting algorithms, especially merge sort.

While we have not gotten to the sorting algorithms in-class, we have read about them – and that should be enough for these purposes. Remember, we're not asking for you to write these algorithms – you will be given complete and working code. Instead of writing these algorithms you will simply be modifying them to count certain operations. The focus here is more the "counting" than the sorting itself, and you may even find this process leads to a deeper understanding of the algorithms than other study approaches you've taken.

3 Files

This lab will involve the following files

- `sorts_template.py` - this contains the sorting algorithm code from zybooks. You will be modifying this code (and renaming it)
- `sorts.py` - this is the name of your modified version of the sorts code.
- `make_sorts_data.py` this contains a main program which will run the sorting algorithms for a variety of problem sizes, and four different types of lists.
- `sort_count_test` this runs the sorting algorithms with a few example lists validating that the comparisons are being counted correctly.
- several csv files generated by `make_sorts_data.py` these will be generated when you run the code and contain the results of the test
- `timing_util.py` This file is optional, and is provided if you wish to do the "more fun" at the end of this lab.
- several image files you will make (in the software of your choice, or by hand) that shows the behavior of the different sorting algorithms under the different lists

Of these files, you only need to turn in: `sorts.py`, and your 4 image files. The autograder will confirm you have `sorts.py`, but it cannot confirm the image files for you (because we have not required specific filenames.)

4 Python Component of Lab

This section details the python coding part of the lab, remember, this is not the only part of the lab.

For this part, you will need to modify all functions for the three algorithms found there (insertion_sort, selection_sort and merge_sort). Currently, these three algorithms are (almost)exactly as found on zybooks – that is, they represent an in-place sort over a list of values. The functions take a list as input, and modify the list returning no value. (Note, I made one small change to zybook's merge sort algorithm to provide a starting function that doesn't take two ints as parameters. This function simply calls the recursive merge_sort function. Merge sort, therefore is implemented over 3 functions.)

Your job is to modify each of these functions to return the number of times they compare two list elements.

A few things to know about how we want these methods to work and counting comparisons:

- Comparisons can be both equality (`==` or `!=`) or inequalities `<, >, <=, >=`) count both.
- We do not care about all comparisons, only comparisons between elements of the list. So if two indices are being compared, you should not count those
- The counting should happen programmatically, I don't want you to do this by hand, you will need to modify the code to have a count variable which increases as needed
- The merge sort algorithm is implemented recursively and over multiple functions, **ALL COMPARISONS** that result from a call to the merge_sort function should be counted.
- I've left at least one comment in the code itself about a hard-to-count case and how to deal with it.

Specifically, we will be testing, the following:

- is insertion_sort modified to count comparisons?
- is selection_sort modified to count comparisons?
- is merge modified to count comparisons?
  - Merge here being the *helper function* for merge sort.
  - Since this has all of the actual comparisons of merge_sort in it You will need to modify it for counting along with the other functions
  - Like the sort functions, this should return the count of list-element comparisons.
- is merge_sort modified to count comparisons?
  - Pay special attention to make sure this counting is *Repeatable* – you should be able to call this function multiple times and get correct answers each time.

    * Hint – merge sort is a recursive algorithm – you will likely need to think about counting recursively as well. This might feel harder than less elegant solutions, but it's worth the struggle. Figuring this out will not only help you build your recursive thinking skills, but it will also help you understand merge sort in general.
    * Don't forget that the TAs can help you with this particular recursive task, and other students can certainly give you general-purpose pointers on recursion. It might even be worth stepping back from this specific task to review recursion in general if you're struggling.
    * When modifying the `merge_sort_recursive` function to return a count of operations remember that the comparisons done while recursively sorting the left- and right-sublists is just as important as the comparisons done to merge the sublists.
    * You may find it useful when making this modification to **temporarily assume that the recursive calls** to `merge_sort_recursive` and `merge` **return the correct counts** of each respective step's comparisons
  - To verify this, all tests for merge_sort run the test twice in a row – if you give a different answer your code is wrong – this is normally caused by inappropriate use of a global variable.

5 Analysis

Provided is a program `make_sorts_data.py` once you've updated the sorting functions you can run this program. Note, this program may take a moment to run depending on your computer. This program will produce four files on your computer.

- `backwards.csv` (sort lists that are currently sorted backwards)
- `near_sorted.csv` (sort lists where some elements have been swapped out-of-order)
- `random.csv` (sort un-sorted lists)
- `sorted_list.csv` (sort already-sorted lists)

Each file will have a header row and four columns of information, the first is the size of the list, and the remaining columns are the different sorting algorithms.

Run this program and then, for each of the four scenarios, generate a plot of the data. Your plots should indicate which line is associated with which algorithm, and be based on all data in the CSV file. Your plot should be in some standard image format (png, jpg, svg, pdf) (NOT a document format, so no docx or excel files). Your plots can be generated by any software you are familiar, or by hand and photographed. If you want a recommendation, excel or google sheets are pretty good at plotting data in CSV formats. If you have prior experience with matplotlib for python, there is a plotting utility in that package as well.

Important note It's common and expected for two lines to mostly overlap, or for slow-growing curves to be hard to see at the bottom of an image. If this happens it's often best to simply note this in your answers and move-on. Some drawing tools make it easier to tell this is happening, others don't, and you may need to look at the raw data to know what's going on.

Academic integrity note Since "plotting output curves" is not a learning objective of this class, it is allowed and encouraged to help other students plot their output CSV files. When doing this make sure that you are helping the other students plot their own outputs, NOT giving them your output files, or giving them your plots. Each group should have their own plots based on the output of their own programs.

6 Report

Finally, at the top of the sorts template file is a series of questions. You should answer these questions by updating the strings in the sorts.py file. Taken together, these questions provide a brief analysis of the plots generated in the previous steps, and guides you through considering how the practical realities of these algorithms line up with theoretical values.

1. If the filenames are not obvious – which image file covers each case.

2. How did each algorithm behave (answer separately for each algorithm). Describe what sort of behavior you see in the images – include any noticeable properties such as the shape of the curve, if it looks straight or curves upwards etc. If the answer for a given algorithm varies based on the case make sure to describe this as well.

3. What is the theoretical runtimes for each algorithm (answer separately for each algorithm). If the algorithm's behavior should, in theory, change with the different cases, explain that too. You should be able to find this information in the textbook.

4. How do your observations compare to the theoretical runtimes for these algorithms (answer separately for each algorithm) If your answer varies case-by-case explain that as well.

5. Merge sort is theoretically the fastest algorithm, are there cases where another algorithm might be faster?

6. If you didn't know the order of data you might want to sort, what algorithm might you use to sort it, and why?

These answers do not need to be in essay format – you can simply have a list of bullet-point answers to each question. The answers need to be contained in the correct python multi-line string. Remember – python multi-line strings can have newlines so feel free to give answers like:

```
# Question 1: Which image file you submitted covers which analysis case?
Answer1 = '''
 * img1.png is random order.
 * img2.png is sorted order.
 * img3.png is near-sorted order.
 * img4.png is reverse-sorted order.
'''
```

So long as your answers are reasonably readable and the spelling / grammar do not prevent us from understanding you we will not judge you, however, if your spelling or grammar is bad enough that we cannot correctly understand your meaning, you WILL lose points for it. The same logic applies to your plots, they do not need to be works of art, and hand-drawn is acceptable, but if an image is unclear or hard to actually see (really bad lighting, weird zoom, too small etc.) you may lose points.

7 Submission details

You will be expected to submit the following files:

- `sorts.py` (modified to perform counting)
- four image files in a standard image format (jpg, png, svg, pdf) representing the four cases tested by out test code. (2 points each)

8 Grading Rubric

The general grading rubric is as follows:

- 26 points from autograder
  - 6 points (as normal) on submission, filename checks, and runability of the submitted file
  - 4 points for insertion_sort based on the examples below
  - 4 points for selection_sort based on the examples below
  - 4 points for merge based on the examples below
  - 4 points for merge_sort based on the examples below. Note, all merge_sort tests are ran twice to ensure your approach to handling the count recursively is repeatable.
  - 4 points for updating all the answers
- 16 points manually grading the code modifications for correctness.
- 8 points for the 4 required image files (2 points per image file)

  - 0 points for image files we cannot find, open, or understand
  - 1 points for image files with a noteworthy/obvious flaw effecting our ability to read/understand it.
  - 2 points for image files that are good enough to understand what is being plotted

- 50 points for the contents of the question answers. See above for the exact questions asked
  - Q1 (filenames for image files) 4 points
  - Q2, (what did you see) (once for each algorithm) 12 points
  - Q3, (what does the theory say) (once for each algorithm) 12 points
  - Q4, (did practice match theory) (once for each algorithm) 12 points
  - Q5 (Can anything beat mergesort?) 5 points
  - Q6 (Which is the best general purpose algorithm and why?) 5 points

9 Extra Fun

This step is NOT REQUIRED and NOT FOR CREDIT.

I've also included a python file timing_utils which has functions to help you time your code. Modify the make_sorts_data program to output timing information. Regenerate your plots and answer the one following question:

- Q7: How does the specific timing of these functions relate to the operation count? Do the results generally match? or no? If not, why not?