Graded Lab 1

Please read the following carefully. There are no completion/participation grades in labs. However, the TAs are there to guide and aid you in your tasks.

Purpose

This lab focuses on implementing, analysing, and optimizing some traditional sorting algorithms. This lab will span two weeks. In the first week you will cover the following (but not limited to):

- Compare the runtimes of Bubble, Selection, and Insertion sorts
- Implement variations of each Bubble and Selection sort and run experiments to determine how much (if any) improvements you observe
- Analyse the performance of these algorithms under specific cases such as short lists and if the list are “near sorted”

During the second week you will cover the following (but not limited to):

- Compare the runtimes of Merge, Quick, and Heap sorts
- Implement variations on Merge and Quick sort and run experiments to determine how much (if any) improvements you observe
- Analyse the performance of these algorithms under specific cases such as short lists and if the list are “near sorted”
- Determine if a “hybrid” search strategy would be beneficial
- Implement your sorting algorithms via a strategy pattern (we’ll see how the workload is)
    - **Update:** Due to the number of experiments this week, this portion has been postponed, potentially to the project.

Part 1

Throughout this lab you will be creating and ultimately submitting a lab report. Your lab report should look professional and complete. It should include the following:

- Title page
- Table of Content
- Table of Figures
- An executive summary highlighting some of the main takeaways of your experiments (this can be presented in bullet form if you wish)
- A clearly marked section for each experiment/exercise outlined in this lab
- An appendix explaining to the TA how to navigate your code (for example, which .py file to find which implementation/experiment in)

For each experiment, include a clear section in your lab report which pertains to that experiment.

Experiment 1

In the file bad_sorts.py posted alongside this document, you will find implementations of Bubble, Insertion, and Selection sort. Run suitable experiments to compare the runtimes of these three algorithms. In bad_sorts there is a create_random_list function which may be useful. In your report this section should include:

- An explicit outline of the experiments you ran. That is, list length values, how many "runs", etc.
- A graph of *list length vs time* displaying the three curves corresponding to the three "bad" sorting algorithms
- A brief discussion and conclusion regarding the results. A few sentences are fine here.

Experiment 2

Implement the following two algorithm variations.

- In lecture we saw that we could improve insertion sort by keeping track of the value we are inserting, finding the location that we want to insert it into, and then appropriately shifting the remaining values, rather than "swapping" it down to where it should go. Review the lecture material if you are unfamiliar with this or ask the TA to explain it. Apply the same general approach to Bubble sort. This will be slightly more complicated since in bubble sort you will need to potentially insert many values and shift things appropriately during a single iteration. Name this implementation of Bubble Sort bubblesort2().
- Instead of having selection sort keep track of the minimum value during a single iteration and positioning it accordingly, have it keep track of the min and max value of a single iteration and position both values accordingly. Note, your loop *boundaries* should be updated accordingly as well. Ask the TA for clarification if things are not clear.

Run experiments where you compare the original Selection and Bubble Sort runtimes to their potential improvements. In your report this section should include:

- An explicit outline of the experiments you ran. That is, list length values, how many “runs”, etc.
- Two separate *list length vs time* graphs:
    - One comparing the original Bubble Sort and its variation
    - One comparing the original Selection Sort and its variation
- A brief discussion and conclusion regarding the results. A few sentences are fine here.

Experiment 3

In bad_sorts.py you will find a create_near_sorted_list(length, max_value, swaps) function. This creates a random list of length *length* of values between 0 and *max_value*. Furthermore, it will make a number of random swaps equal to *swaps.* For example, if swaps = 0, the list will be perfectly sorted. If

swaps = length*log(length) / 2

the list will be statistically indistinguishable from a randomly generated list. Do not worry about understanding the above result – just trust it for now.

Run an experiment where you compare the runtimes of the three bad sorting algorithms (or their improvements, your choice) vs how many random swaps are made on sorted list. For this experiment fix the list length to be constant at 5000 (or another reasonable value – nothing too small). In your report this section should include:

- An explicit outline of the experiments you ran. That is, the fixed list length, number of runs, swaps, etc.
- A graph of *swaps vs time* displaying the three curves corresponding to the three “bad” sorting algorithms. Choose the range of swaps wisely here. If interesting things do not occur for values past a certain threshold, do not feel obligated to include those on the graph.
- A brief discussion and conclusion regarding the results. A few sentences are fine here.

Part 2

Experiment 4

In the file good_sorts.py posted alongside this document, you will find implementations of heap, merge, and quick sort. Run suitable experiments to compare the runtimes of these three algorithms. Hint: it should become clear where Quick sort gets its name. In your report this section should include:

- An explicit outline of the experiments you ran. That is, list length values, how many "runs", etc.
- A graph of *list length vs time* displaying the three curves corresponding to the three "good" sorting algorithms
- A brief discussion and conclusion regarding the results. A few sentences are fine here.

Experiment 5

Okay, so quick sort is quick, big surprise. But what is one of the biggest issues with quick sort? Its worst case runtime is O(n^2). Specifically, if you are choosing the pivot to be the first element in the list (like my implementation does) it will lose to heap and merge sort for lists which are already or reverse sorted. So how "non-sorted" does the list need to be before quick sort begins to win again? Using the create_near_sorted_list() function from part one, devise an experiment to determine the factor of "swaps" needed before quick sort is appealing. In your report this section should include:

- An explicit outline of the experiments you ran. That is, list length values, how many "runs", etc.
- A graph of *number of swaps vs time* displaying the three curves corresponding to the three "good" sorting algorithms. List length should be constant here.
- A brief discussion and conclusion regarding the results. A few sentences are fine here.

Experiment 6

As we have (hopefully) noted, quick sort is quick. But is it as quick as it could be? What potential modifications could be made to speed it up? Consider the following change. Instead of having a single pivot value which we divide the list on (into a left and right portions), why not have two pivots? That is choose two values arbitrarily (the first and second value of the list for example), and split the list into three portions based off the two pivots. Specifically, there will be a left portion of all values less than both pivots, a right portion of all values greater than both pivots, but also a middle portion between the two pivots (inclusive). Then we can recurse down on the three portions. Implement a dual pivot quick sort and name it dual_quicksort(L). Run a suitable experiment to determine if this change is worth doing. In your report this section should include:

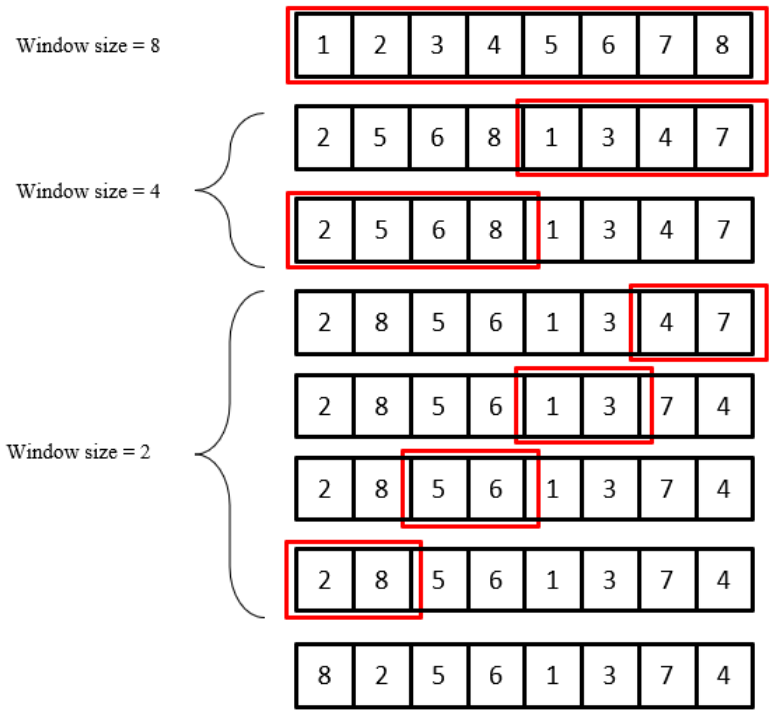
- An explicit outline of the experiments you ran. That is, list length values, how many "runs", etc.
- A graph of *list length vs time* displaying the two curves showing the traditional and dual pivot version of quicksort
- A brief discussion and conclusion regarding the results. A few sentences are fine here.

- **The following is not required, nor will you get bonus grades**, but why stop at two pivots? Why not 3, 4, n? If you are interested, I invite you to try and implement and test these versions. After the due date I will post a 3 and 4 pivot version for the curious few.

Experiment 7

We have noted a general trend that recursion usually loses to iteration. However, Merge sort is fundamentally recursive in nature. Can we implement it iteratively? Yes.

Merge sort seems to embody the divide-and-conquer approach. But do we really need to divide? How about just conquering? What I mean by that is, instead of recursively splitting the list into smaller parts (from the top down), and then rebuild it. Why not instead start off by viewing the array/list as already divided, and simply complete the rebuild/merge portion. This is the essence of the bottom-up implementation. The idea is you iteratively pass through the list and merge portions of the list based off some window size (see the figure below). The size of the window increases after each iteration of the loop.



Implement a bottom up merge sort and name it bottom_up_mergesort(L). Note: in the figure above the length of the list is conveniently a power of 2, in general this may not be the case. You will have to resolve this in some way.

- An explicit outline of the experiments you ran. That is, list length values, how many “runs”, etc.
- A graph of *list length vs time* displaying the two curves showing the traditional and bottom up versions of merge sort
- A brief discussion and conclusion regarding the results. A few sentences are fine here. Hint: I personally saw a slight improvement in runtime for the bottom-up approach. However, not as

much as one might expect after seeing the difference between the iterative and recursive implementations of binary search. Why may this be the case? Merge sort is O(nlogn), therefore, the majority of its runtime is caused by the linear (n) portion of the algorithm. Is this recursive in nature?

Experiment 8

Can a “bad” sort ever be better than a “good” sort? For specific cases, absolutely. In this experiment you will compare insertion sort to merge and quick sort for small lists. Devise an experiment which shows when insertion sort beats merge and quick sort as well as determines when it indeed becomes “bad”.

- An explicit outline of the experiments you ran. That is, list length values, how many “runs”, etc.
- A graph of *list length vs time* displaying the appropriate three curves showing. List lengths should be small here.
- A brief discussion and conclusion regarding the results. A few sentences are fine here.
- A few sentences on why these results are important and practical. Hint: could you make a “hybrid” sort of insertion and merge sort?

Grading and Submission

Your group will submit the following documents to Avenue:

- report.docx (or .pdf, or whatever – as long as a reasonable person can open it)
- code.zip (all your source code pertaining to the lab – including experiment code)

In addition to the grade allocations below, your report may lose up to 20% of the final grade for not looking professional, having formatting/style issues, graphs presented in a messy manner, etc. Moreover, you may lose grades for not including elements explicitly mentioned in the Part 1 section of this document. Find a rough grade breakdown below:

Part 1

Experiment 1	10%
Experiment 2 and corresponding implementations	20%
Experiment 3	10%

Part 2

Experiment 4	10%
Experiment 5	10%
Experiment 6 and corresponding implementation	15%
Experiment 7 and corresponding implementation	15%
Experiment 8	10%