3XB3 Lab 1:

Experimenting with Sorting Algorithms

Muhammad Umar Khan

Tim Pokanai

Omar El Aref

Table of Contents

Table of Figures

Executive Summary

Throughout the experiments ran there were many different observations that we made. Some were general observations while some were specific to different experiments. One of the main take away of these experiments though was that not all the sorting algorithms that we learned in 2C03 were as fully optimized as they could be. Some required to be tweaked and changed a bit to run faster than their traditional parts. This could prove to be crucial when dealing with huge sets of data as the faster sorting algorithms run the better. Another main takeaway was that the superiority of different algorithms may depend on the circumstances that they are used. For example, in experiment 8 we saw that 'Insertion Sort' ran faster than 'Merge Sort' for certain list lengths. We also saw how 'Quick Sort' was beaten by two of the good sorts because of the degree of randomness in the list. There were also other key findings that we found for each experiment.

Experiment 1:

- 'Selection Sort' is the fastest bad sort on average.

Experiment 2:

- Having less write operations leads to faster performances.

Experiment 3:

- 'Selection Sort' had a consistent performance, while 'Insertion Sort' was efficient for nearly sorted lists (few swaps).
- 'Bubble Sort' was consistently the slowest.

Experiment 4:

- 'Quick Sort' generally outperformed 'Merge Sort' and 'Heap Sort'

Experiment 5:

- 'Quick Sort' was faster when the list was more random.
- 'Merge Sort' and 'Heap Sort' were faster when the list was less random.

Experiment 6:

- 'Dual Quick Sort', utilizing dual pivots, notably outperformed traditional 'Quick Sort', especially in larger lists.

Experiment 7:

- 'Bottom-Up Merge Sort' consistently outperformed traditional 'Merge Sort' across all tested list lengths.
- Traditional divide and conquer might not always be the best idea.

Experiment 8:

- ‘Insertion Sort’ can be better than ‘Merge Sort’ when the lists are small in length.

Experiment 1

In this Experiment we decided to experiment with five different list lengths, 10, 100, 500, 1000 and 5000. To ensure fairness between the three algorithms we made lists using the 'create_random_list()' and then copied them to make sure each algorithm got the exact same list. To make sure our results were accurate we ran each algorithm five times and took the average time between the 5 runs.

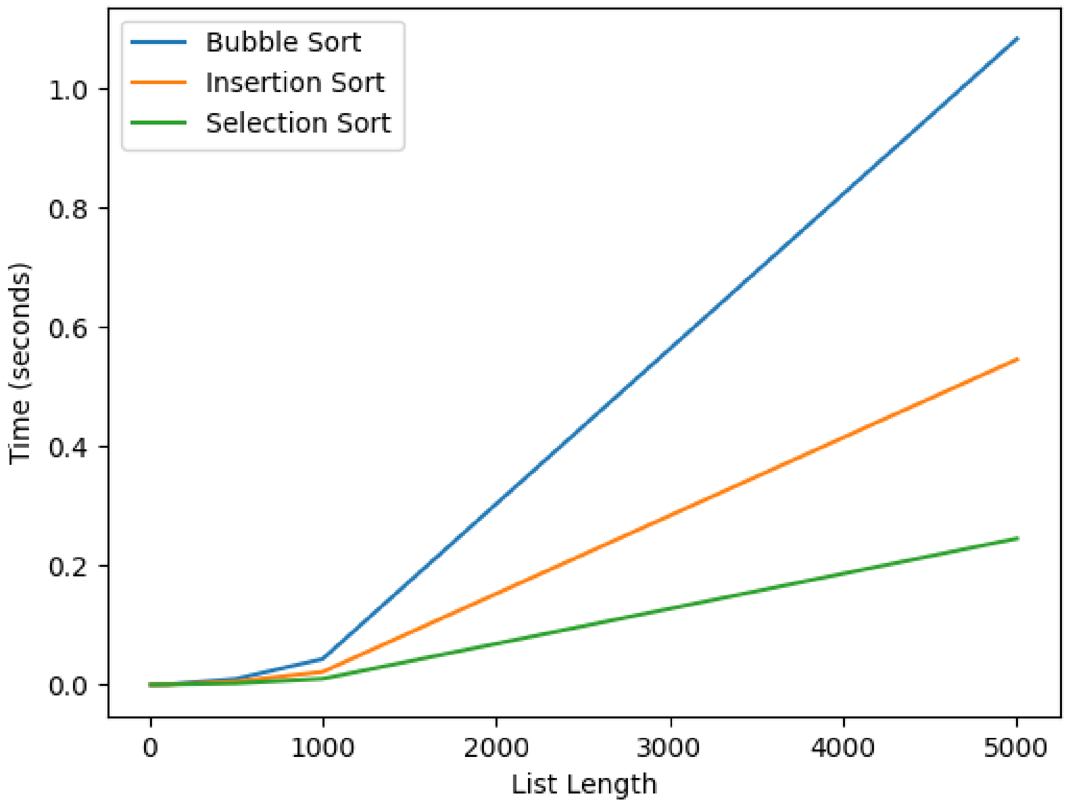

Figure 1:Three bad sorts comparison

As shown in the results in *Figure 1* all three algorithms preformed around the same when the list size was around 500 or less. Though for list lengths larger than 500, there was a major difference between the three algorithms. Bubble sort became much slower than Insertion and Selection sort. Both, Insertion and Selection Sort, got significantly slower as the list length increased but Selection Sort seemed to have been affected the least between the three algorithms meaning out of the three bad sorts Selection Sort was the best.

Experiment 2

For experiment 2 we ran each test 50 times. For each test we increased the list from 10 to 1,000 with each step being 10. That way we got to test each algorithm with different list lengths. The maximum number in each list was 1,000.

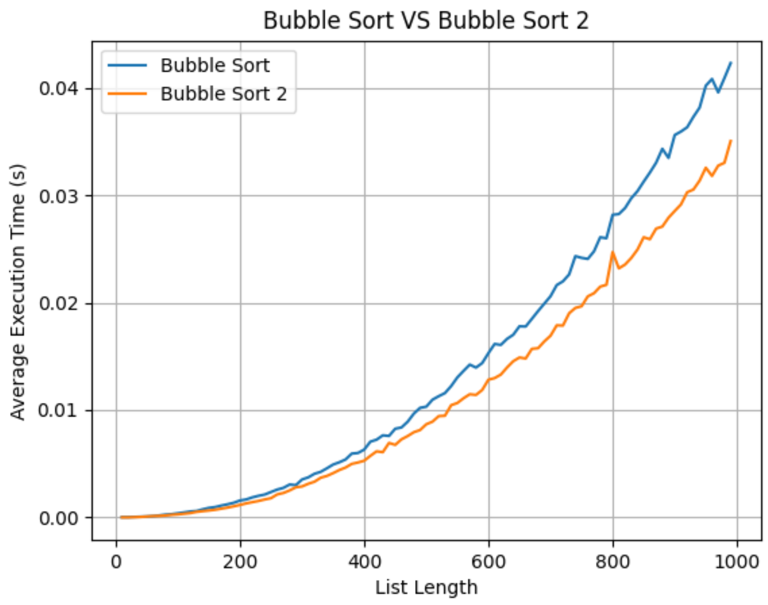

Figure 2: Bubble Sort VS Bubble Sort 2

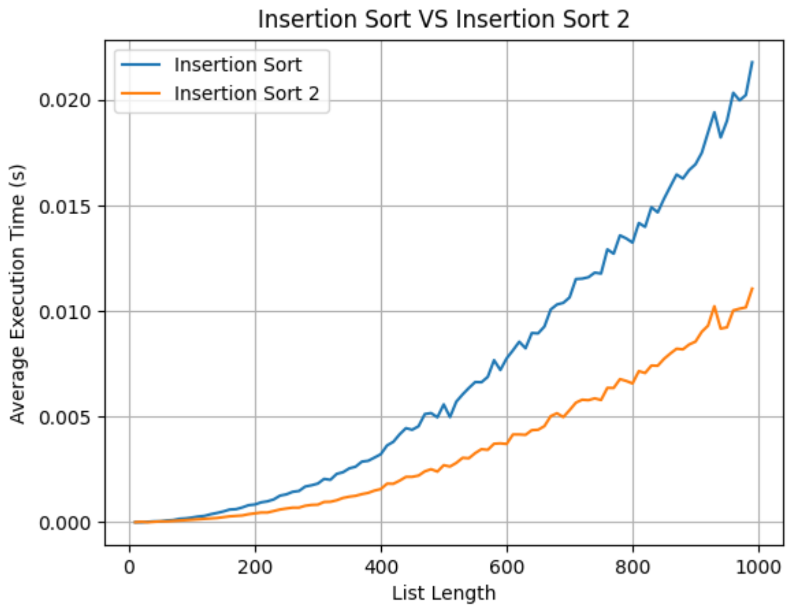

Figure 3: Insertion Sort VS Insertion Sort 2

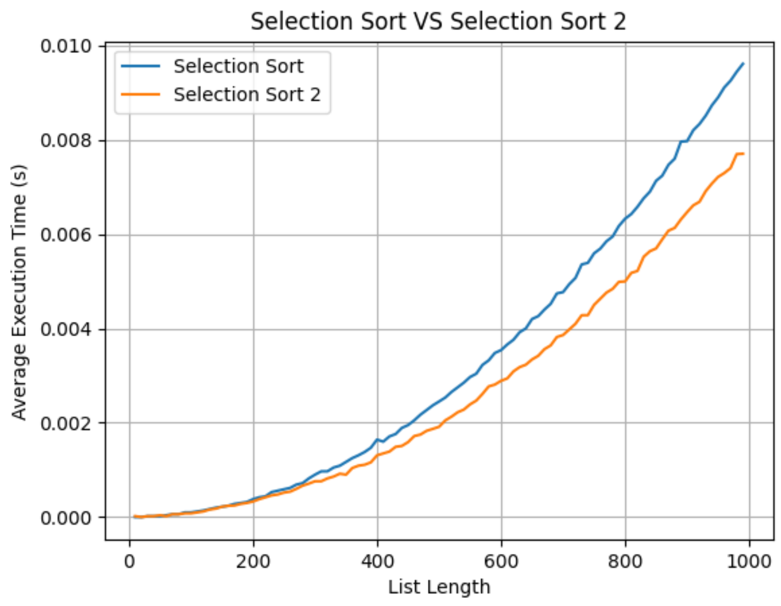

Figure 4: Selection Sort VS Selection Sort 2

As shown by all three graphs, the difference between the lists when the list lengths are 150 or less the algorithms seem to work at basically the same speed. Though as the list length increases there seems to be a larger difference between the speeds. This experiment proved that having less 'writes' is better for these algorithms as it takes too much time to write down and update the list constantly.

Experiment 3

Experiment 3 consisted of testing the performance of the three original bad sorting algorithms on a list of fixed length with a variable number of random swaps performed on it. To conduct this experiment, the fixed length of the list was 1000. The range of random swaps performed throughout the experiment was between 0 and 1000 inclusive, while using 10 step swap increments. At a given number of swaps every algorithm was given the same copy of a list to sort to ensure consistency in results. Every number of swaps was run 10 times, and the aggregated time for each sorting algorithm was averaged out over the number of runs for a smoothed result.

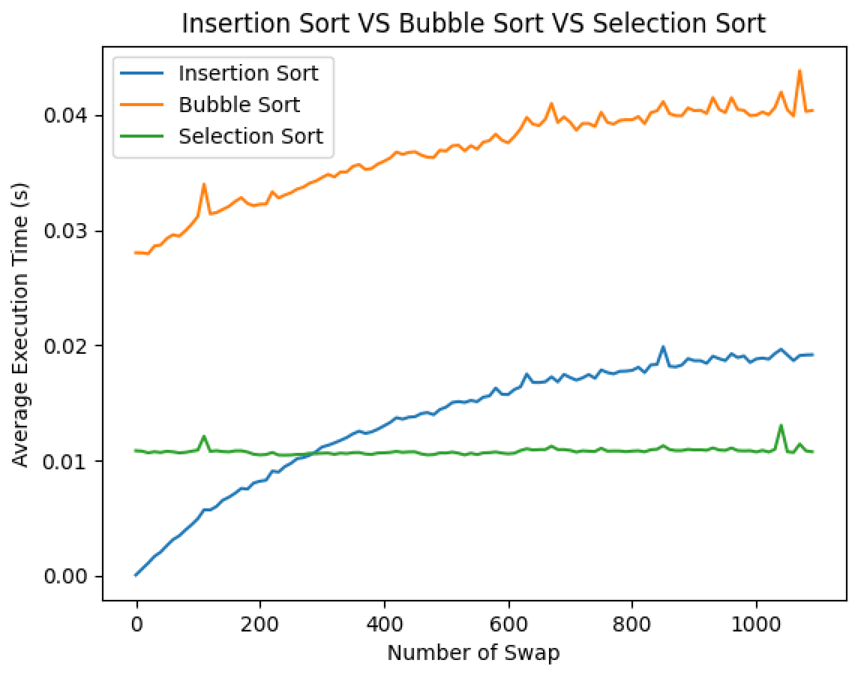

Figure 5: Insertion Sort VS Bubble Sort VS Selection Sort

As shown in *Figure 5*, we can immediately identify that the runtime of the Selection Sort algorithm remains relatively constant for a list of fixed length, no matter how many random swaps are performed on the list. Additionally, Insertion Sort outperforms Selection Sort until roughly 300 random swaps are performed. We also notice that Bubble Sort is always slower than both Insertion Sort and Selection Sort for any number of random swaps performed on the list. From this experiment we can conclude that out of the three bad sorts Selection Sort should be preferred when sorting lists of the same length with many unsorted elements, whereas Insertion Sort should be preferred when the list is almost unsorted.

Experiment 4

Experiment 4 ran the Quick Sort, Merge Sort, and Heap Sort algorithms on different lengths of randomized lists, generated with create_random_list() to compare their runtimes. The list lengths used throughout the experiment were between 0 and 30000 inclusive, with step sizes of 1000. When running each algorithm at a given list length, every algorithm was given the same copy of a list to sort to ensure consistency in runtime results. Lastly, each algorithm was given 10 runs for each list length, and the total time was averaged out over the number of runs for a smoothed result.

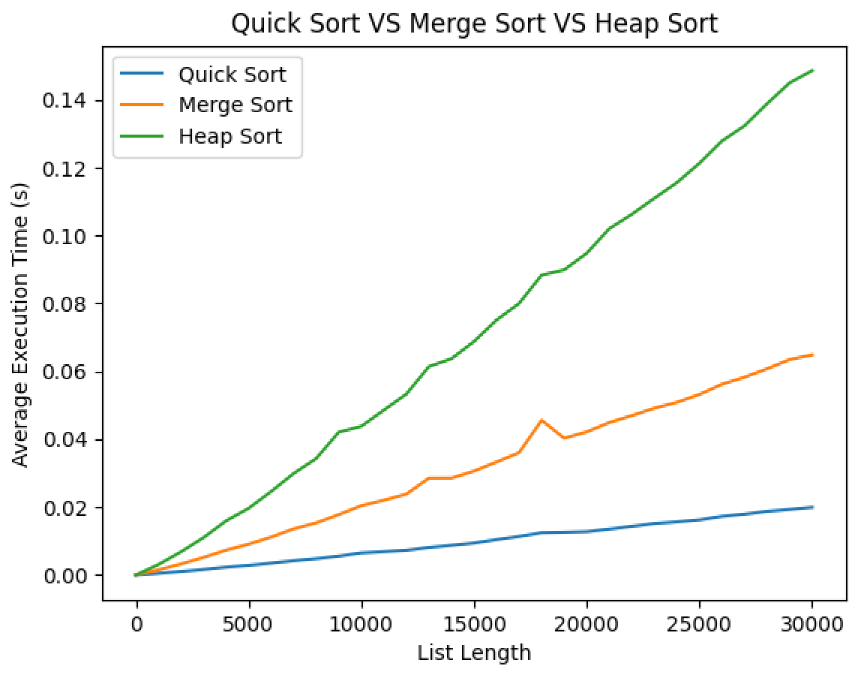

Figure 6: Quick Sort VS Merge Sort VS Heap Sort

From *Figure 6* we can observe that all three of the good sort algorithms scale well in execution time with considerably larger list lengths, especially when compared with the bad sorts. With these results Quick Sort remains the fastest in runtime complexity while sorting random unsorted lists of varying lengths, followed by Merge Sort staying slightly slower and then Heap Sort being the slowest of the three good sorts. Heap Sort's significant increase in runtime complexity is likely because of its use of the Heap data structure adding overhead. Through this experiment we can conclude that Quick Sort's runtime dominates Merge Sort and Heap Sort when sorting randomized lists of varying length, though these three good sorts are preferred over any of the bad sorts when sorting lists of large lengths for their efficiency and scalability.

Experiment 5

In experiment 5 we ran the three good sorts, 'Quick Sort', 'Merge Sort', and 'Heap Sort'. The goal of this experiment was to see how unordered the list must be for 'Quick Sort' to be faster than the other two sorting algorithms. To test this, we kept the list length the same, 1,000, as that would show us the best representation of each of the algorithms. We tested each list 50 times and got the average pf the lists so that way there were no results that were skewed. For the number of swaps, we first started with 0 swaps and incremented each round with 10 swaps until we had reached 500 swaps.

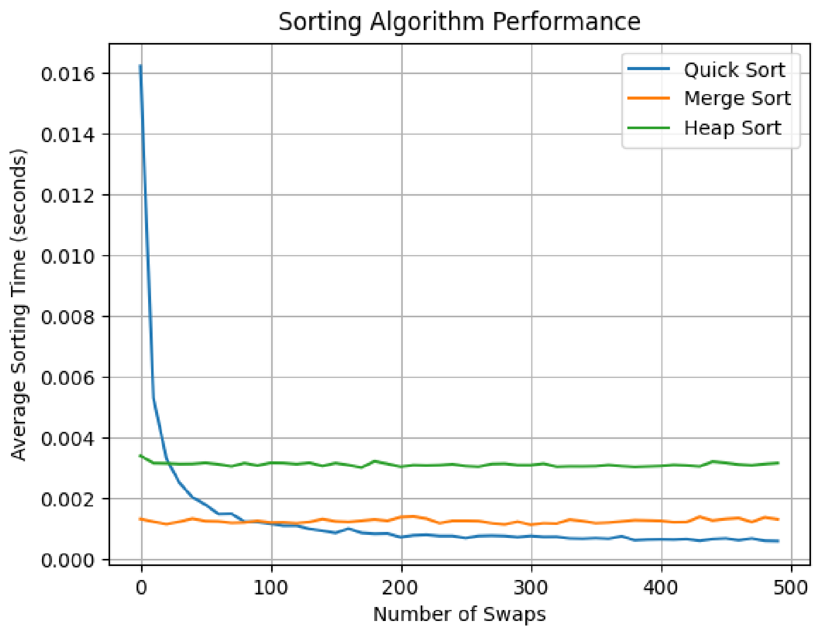

Figure 7: Merge Vs Heap Vs Quick with different number of swaps

As shown in the results in figure 7, 'Quick Sort' was worse than both sorts when the number of swaps was less than around 30. Though from around 30 – 80 swaps 'Quick Sort' was faster than 'Heap Sort' yet still slower than 'Merge Sort'. After the number of swaps exceeded 80 'Quick Sort' became the fastest sort out of the three. The conclusion that we can draw from this experiment was that although 'Quick Sort' is known to be the fastest sorting algorithm, sometimes its worst case scenario can be beat by the two other good sorts, 'Heap Sort' and 'Merge Sort'.

Experiment 6

The way we tested the difference between the two algorithms in this experiment was by also testing how long both algorithms took to sort the same lists at different lengths. The range of list lengths was between 100 and 100,000. At each list length we ran 5 different lists. The increments between list sizes were 1,000.

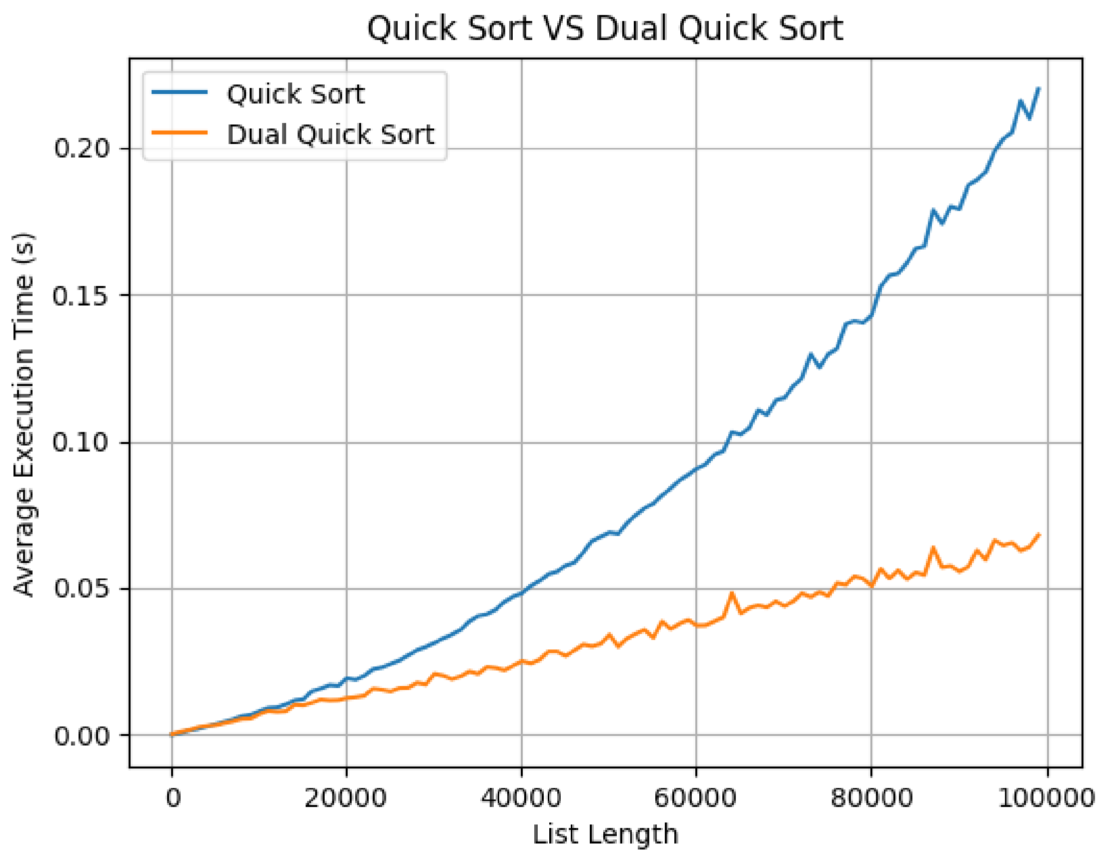

Figure 8: Quick Sort VS Dual Quick Sort

As shown by the results in figure 5 'Dual Quick Sort' runs much faster than 'Quick Sort' as the lists get larger. Though when the list is short the difference in time is almost the same. Based of the resulting graph it is obvious that having more than one pivot greatly decreses the amount of time 'Quick Sort' takes which is impressive considering 'Quick Sort' is arguably the fastest sorting algorithm.

Experiment 7

In this experiment, we tested the difference between the two algorithms by comparing the time they took to sort lists of varying lengths, ranging from 100 to 100,000. For each list length, we tested five different lists. The list lengths increased by increments of 1,000.

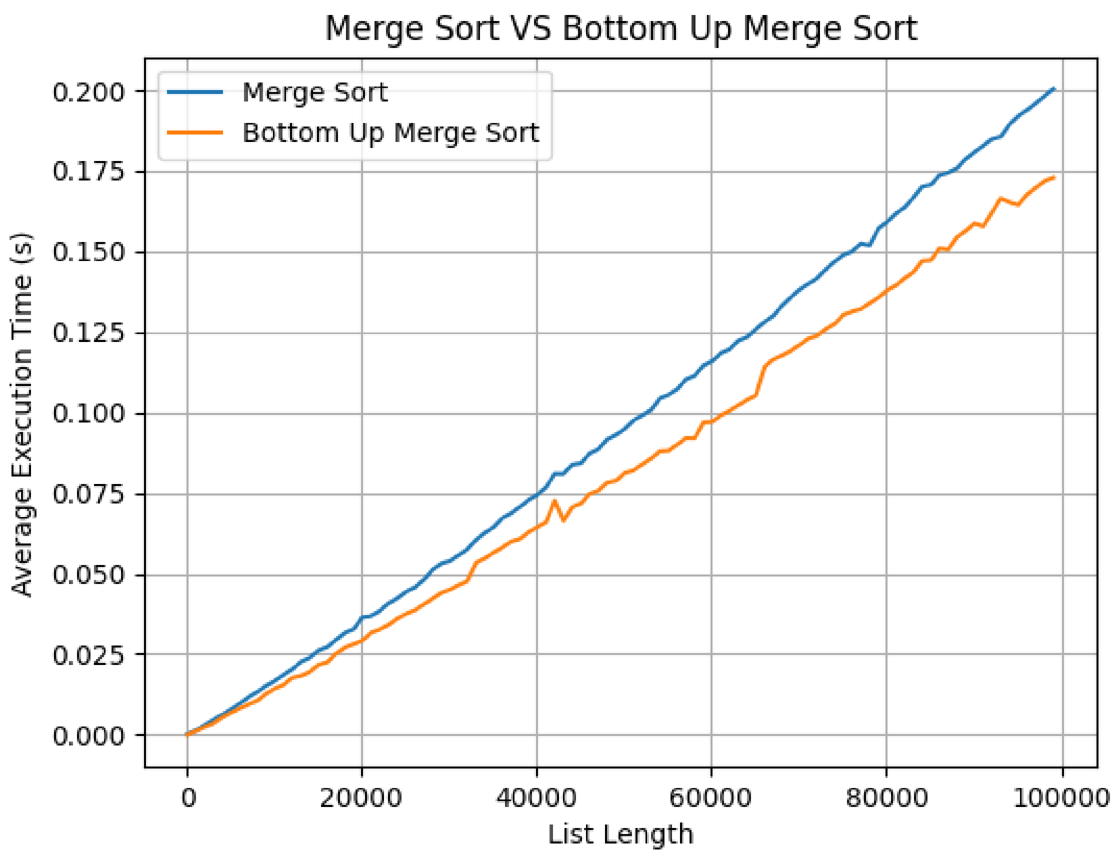

Figure 9: Merge Sort VS Bottom Up Merge Sort

As shown by figure 6 'Bottom Up Merge Sort' seems to be the faster sorting algorithm regardless of the list length. This Implementation also didn't use recursion which may or may not have helped the results. Though it is clear that 'Bottom Up Merge Sort' is faster which proves that 'conquering' the list is more efficient than 'divide and conquer'.

Experiment 8

The final experiment consisted of testing Insertion Sort with Quick Sort and Merge Sort against lists of varying length to determine the circumstances of when a bad sort such as Insertion Sort can outperform good sorting algorithms. To test these circumstances, we used list lengths ranging from 10 to 100 inclusive, with 10 step increments. When running each algorithm at a given list length, every algorithm was given the same copy of a list to sort to ensure consistency in runtime results. Lastly, each algorithm was given 1000 runs for each list length, and the total time was averaged out over the number of runs for a smoothed result.

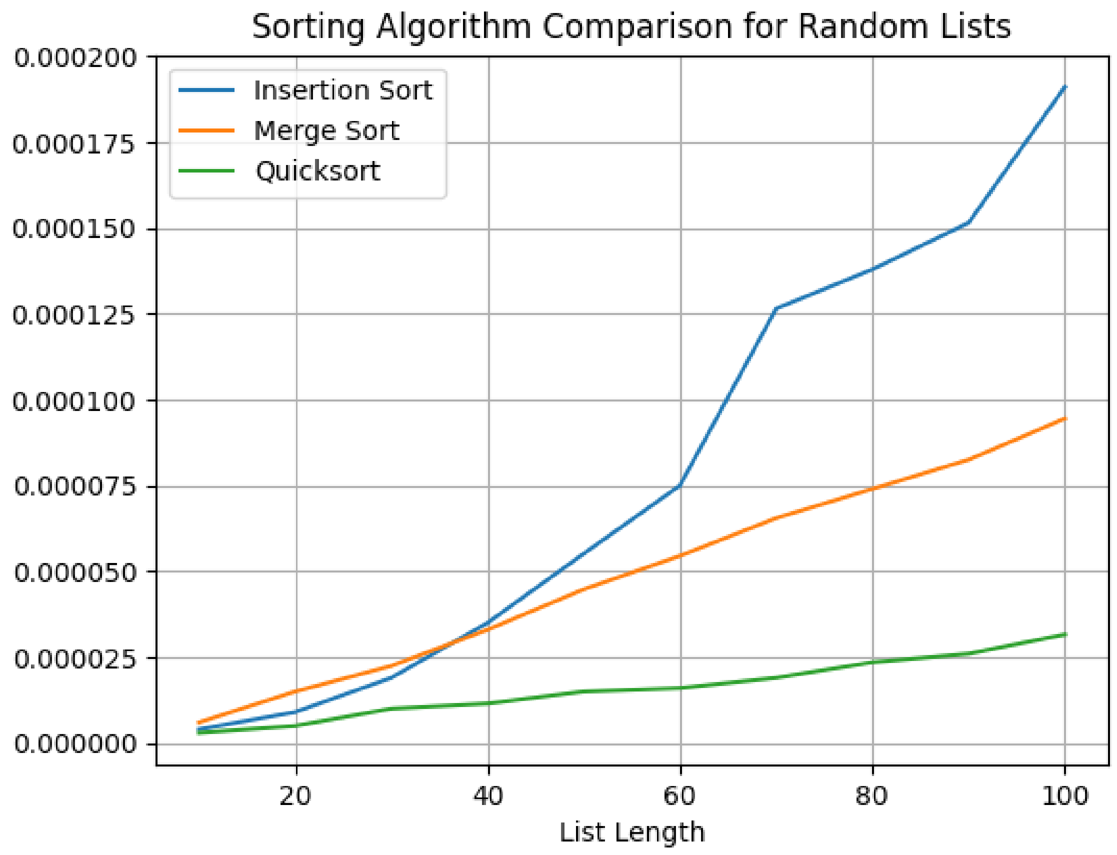

Figure 10: Comparing the Good and Bad Sorts with random list lengths.

From the results shown in *Figure 10,* we can observe that Insertion Sort doesn't surpass Quick Sort with the varying list lengths that were tested, however with list lengths in the range of 10 to 30 both runtimes are very similar. We can also see that Insertion Sort does perform better than Merge Sort with list lengths in the range of 10 to 35 inclusive. In that range we can see the point where a "bad sort" can be better than a "good sort". From this experiment, we can conclude that for lists of particularly smaller lengths, it can be more or equally as efficient to use a "bad sort", such as Insertion Sort, instead of a "good sort". This conclusion can be applied in a very practical sense. One can create a general sorting algorithm that decides which specific sorting algorithm should be used based on the length of the input list. In scenarios where efficiency is important, sorting data can be optimized instead of just consistently using one algorithm for everything.

Appendix

Each experiment is in its own .py file and they are named after the experiments. The only difference is with experiment 1 which is in the bad_sorts.py file as for the rest they are in their corresponding .py file, for example experiment3 is in experiment3.py file. As for experiment 2 it has its own folder since it had 3 experiments in 1 so you should be able to find all the stuff for experiment 2 folder. We have submitted the code in a zip folder that contains all the files needed. We set up a private GitHub for this lab as well which we will make public at the time of delivery: https://github.com/umarkhan135/3XB3-Lab