McMaster University

Department of Computing and Science

# COMPSCI 2XC3
# Lab-02 Report

Suhaas Parcha (Mac-ID : Parchas)
Yu Chang (Mac-ID : changy72)
Yuvraj Singh Sandhu (Mac-ID : sandhy1)
Group 20-L01

*Professor:* Dr. Vincent Maccio

A report on the performance of various sorting algorithms under different circumstances

February 9, 2023

# Contents

**Part - 1** **vi**

0.1 **Experiment 1** . . . . . vi

0.2 **Experiment 2** . . . . . xii

0.3 **Experiment 3** . . . . . xv

**Part - 2** **xviii**

0.4 **Experiment 4** . . . . . xviii

0.5 **Experiment 5** . . . . . xxiv

0.6 **Experiment 6** . . . . . xxvi

0.7 **Experiment 7** . . . . . xxxi

0.8 **Experiment 8** . . . . . xxxiii

**Appendix** **xxxvi**

0.9 **How to navigate the code:** . . . . . xxxvi

# List of plots


1 Experiment 1-1 . . . . . vii

2 Experiment 1-2 . . . . . viii

3 Experiment 1-3 . . . . . ix

4 Experiment 1-4 . . . . . x

5 Experiment 1-5 . . . . . xi

6 Experiment 2 - 1 . . . . . xii

7 Experiment 2 - 2 . . . . . xiii

8 Experiment 3 . . . . . xvi

9 Experiment 4-1 . . . . . xix

10 Experiment 4-2 . . . . . xx

11 Experiment 4-3 . . . . . xxi

12 Experiment 4-4 . . . . . xxii

13 Experiment 4-5 . . . . . xxiii

14 Experiment 5 . . . . . xxv

15 Experiment 6-1 . . . . . xxvii

16 Experiment 6-2 . . . . . xxviii

17 Experiment 6-3 . . . . . xxix

18 Experiment 6-4 . . . . . xxx

19 Experiment7Code . . . . . xxxi

20 Experiment7 . . . . . xxxii

21 Experiment 8_1 . . . . . xxxiv

22 Experiment 8_2 . . . . . xxxv

## Abstract

In this Lab we compared the run time of six different sorting algorithms with thousands of experiments to test their performances on various situations, such as the length and the degree of disorder of the list that we put into the sorting algorithms. We also discovered how to make potential improvements on them. The six algorithms all have their disadvantages or advantages at some points, so the goal of this lab is getting to know and consider which sorting algorithm to use when we encounter problems.

# Part - 1

## 0.1 Experiment 1

**Description:**

Compare the run times of bubble, insertion and selection sort with different parameters and determine which is the best and give a conclusion and analysis as to why.

**Procedure:**

In figure 1, the size of the list ranges from 0 to 100. The maximum value in the list is 100. We ran each iteration 5000 times and then took their average to reduce any potential errors.

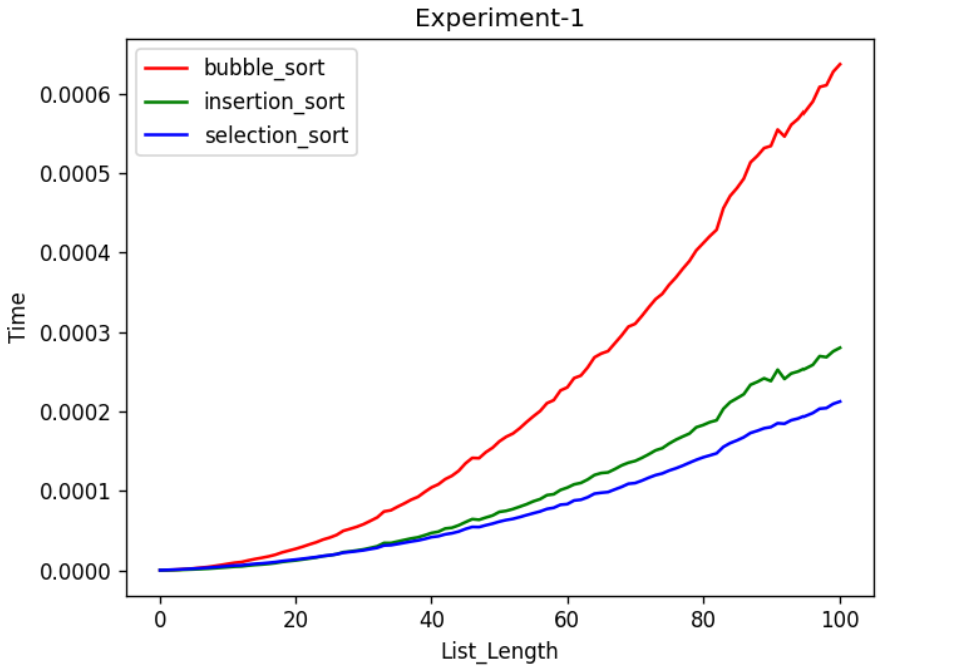



Figure 1: Experiment 1-1

In figure 2, the length of the list starts from 0 and goes to 10,000, the maximum value in the list is 10,000. We ran each iteration 10 times and then took their average. The goal of this test is to analyse the performance of algorithms for large lists.

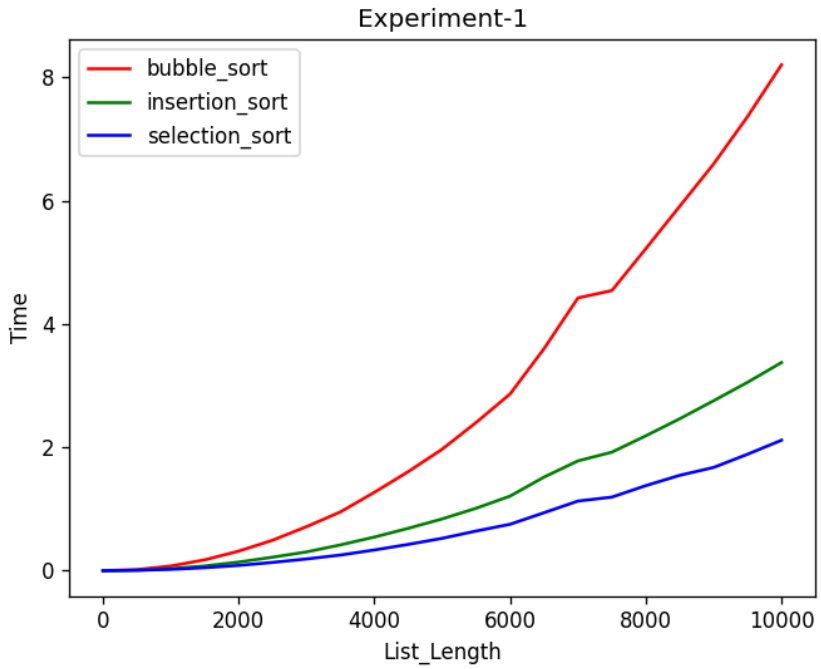



Figure 2: Experiment 1-2

For the next figure, the length of the list starts from 0 and goes to 20,000. But the maximum value in the list is only 100. We ran each iteration 5 times and then took their average. The goal of this test is to analyse the performance of algorithms when there are a lot of duplicate values.

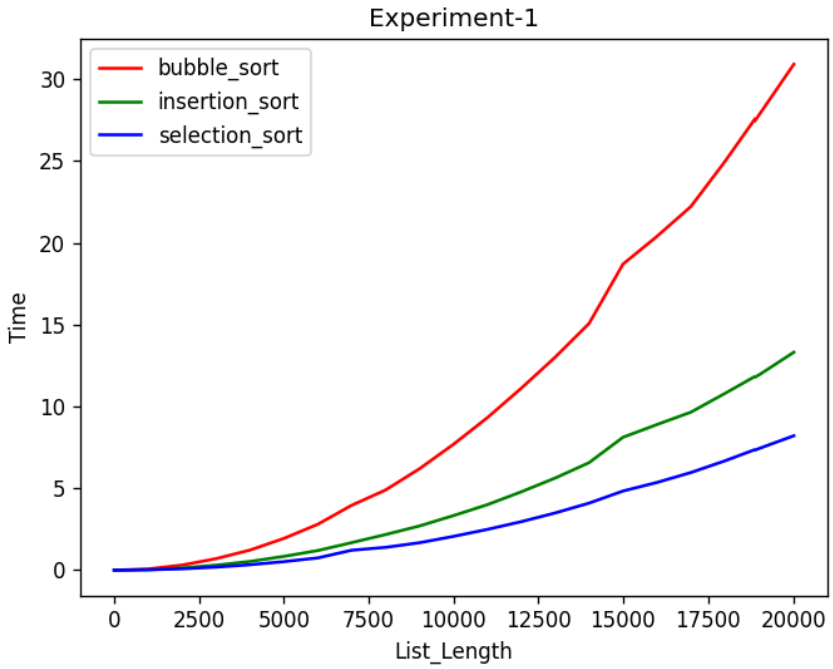



Figure 3: Experiment 1-3

For the graph below, the length goes to 1000, and the maximum value in the list is 100. We ran each iteration 50 times before taking their average. Instead of using a random list for this test, we first sorted a random list and then swapped 10 random values in the list. The goal of the test is to analyse the performance of algorithms when we have to sort a nearly sorted list.

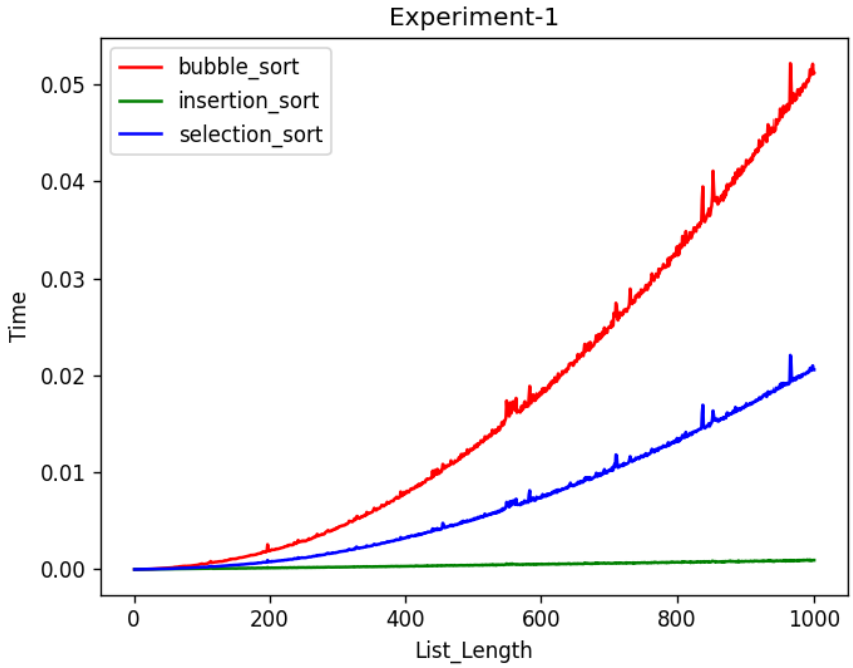



Figure 4: Experiment 1-4

For the figure 5, the length goes to 1000, and the maximum value in the list is 1000. We ran each iteration 100 times before taking their average. But this time we only swapped once in our already sorted list. The goal was to analyse the performance of our algorithms when the list is sorted except for only 1 swap.

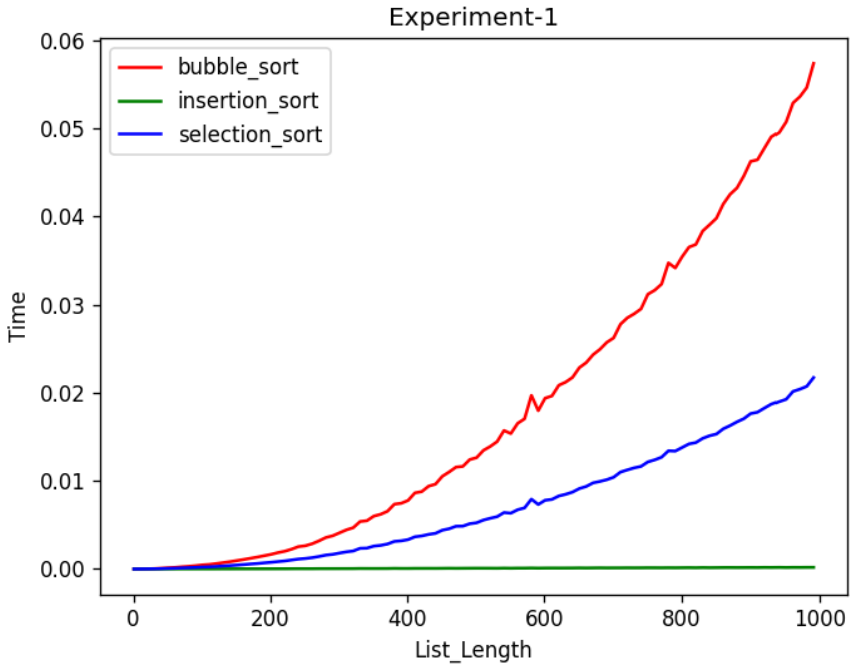



Figure 5: Experiment 1-5

**Conclusion:**

Bubble sort performs the worst among those three sorting algorithms because it has many swapping operations and it contains many "if statements" in the code, which will greatly increase the run time. In comparison with the other sorting algorithms, even if the list is already sorted, it still needs to iterate $Length * Length$ times. Insertion sort performs better than bubble sort and worse than selection sort in most cases. But if the list is nearly sorted, then insertion sort is the best-performing sorting algorithm because if the list is sorted it only takes linear runtime complexity. Selection sort is the best-performing sorting algorithm among those three sorting algorithms if the list is random and not nearly sorted.

## 0.2 Experiment 2

**Description:**

The objective is to create versions of both insertion sort and bubble sort which are better than the existing ones. By a series of tests, we can confirm that the newly written code for both the algorithms is better by comparing both with varying list length values while setting the runs at a constant which is 200 * avg(20) in this case.

**Procedure:**

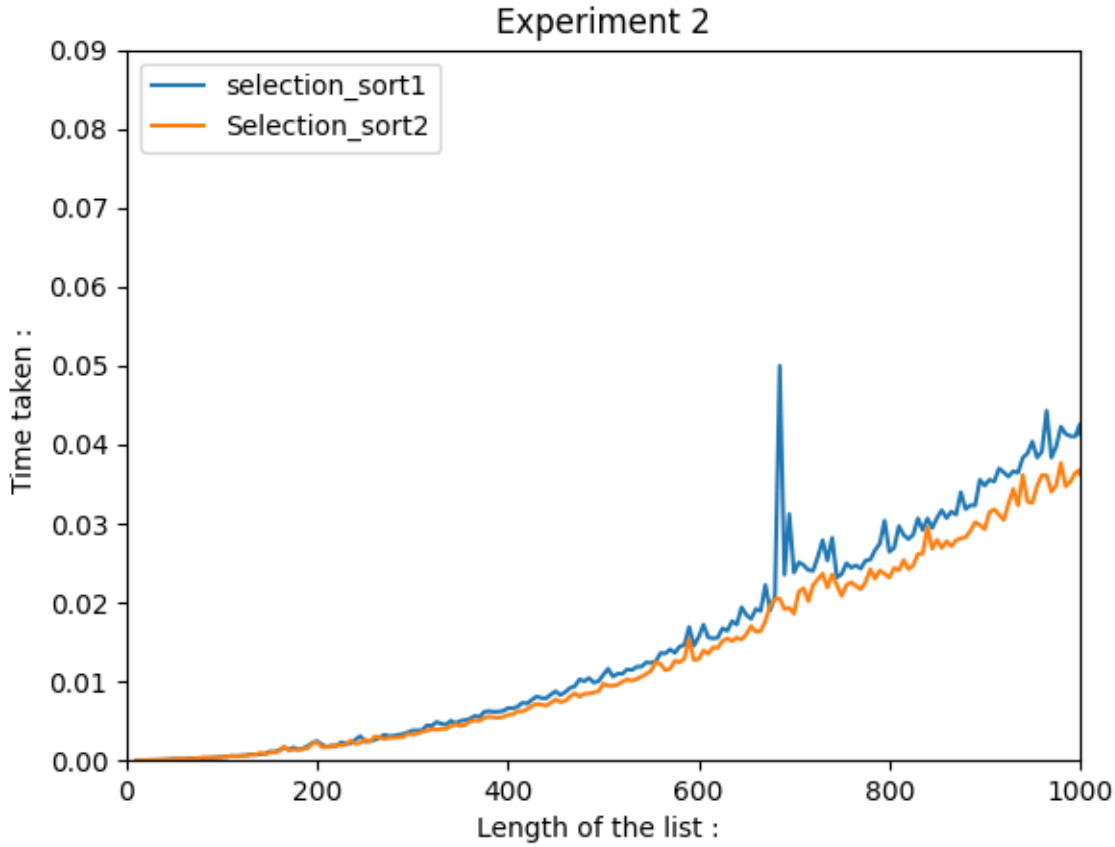



Figure 6: Experiment 2 - 1

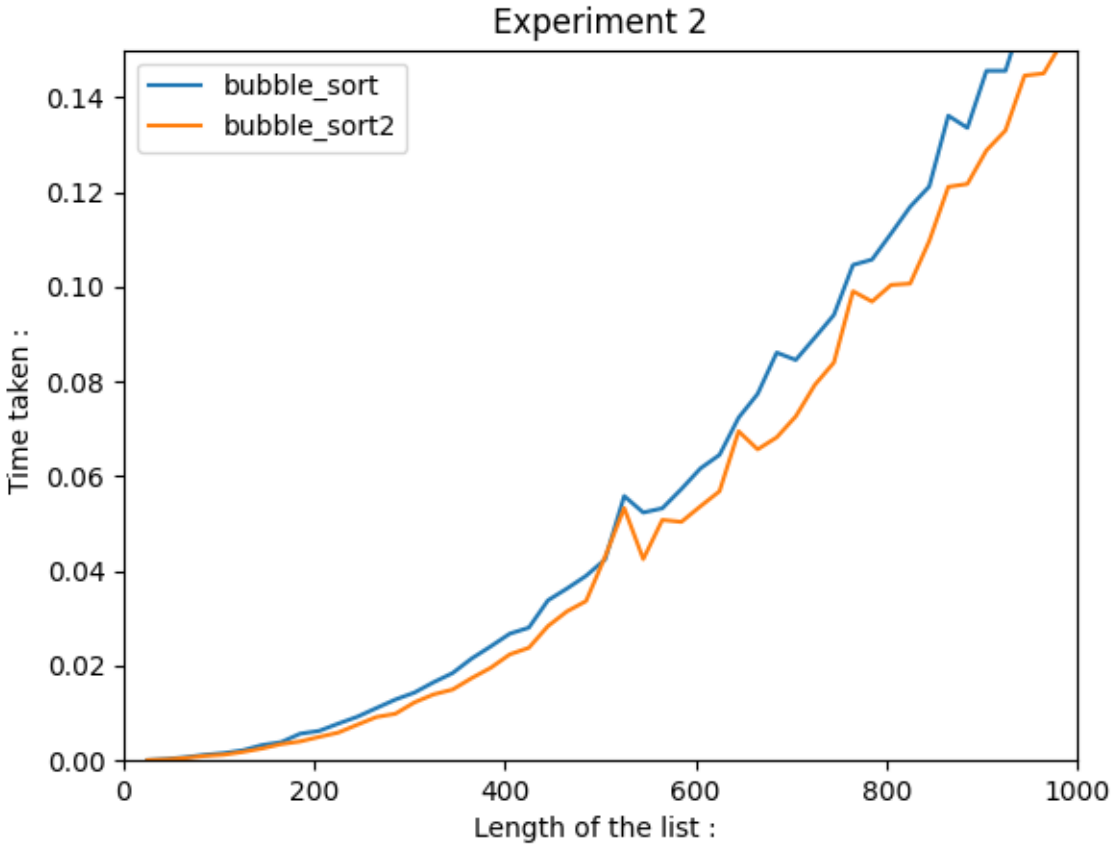



Figure 7: Experiment 2 - 2

Starting with the bubble sort, we can make it better by keeping track of values per iteration and shifting them rather than using the swap function, the idea is to maintain a variable called counter which acts as a flag, rather than going through the whole list again, the algorithm checks for L[i+1], the flag checks if its bigger and then assigns the value to a temp variable var and switches the value with L[i]. This saves a lot of time as the algorithm doesn't have to run another potential O($n^2$) and can just swap multiple values in just one iteration. If the value of L[i+1] is smaller than that of L[i] it sets the flag back to 0 which breaks the loop and the algorithm runs once again until the list is sorted.

Going on to the selection sort, we can make it better by finding both the largest and the smallest value in list. Doing so, while sorting the list, we can reduce the size of the list by decrement of 1 when the largest value is placed at the back of the list , then sort it both ways by putting

the largest value at the back and the smallest in the front. In an ideal case, then the sorting algorithm has to only sort half the list. Once the iterable size of the list reaches the midpoint of the original size of the list, we can stop sorting it from the back and just do normal selection sort. Explaining the code, the function find_min_index2 returns us the max and the min values in the list, the main function keeps swapping the lowest and the largest values until it crosses the iterator crosses the midpoint.

**Conclusion:** From the above bubble sort graph, we can see that for lists of smaller size ($\leq 200$) the difference really isn't much, but when the size increases, the gap between the 2 sorts becomes very noticeable and the average test case for the new sort is much better than the previous one. The number of runs used were 200 with length increments of 5 where the maximum length was 1000. Moreover, 10 tests were run for each sort per iteration to find the average.

From the above insertion graph, Similar to the bubble graph, the runtime is much faster as the size of the list increases. For length is ($\leq 200$) the difference in runtime is much more visible. This is essentially because the average case of the new algorithm is far better than the previous one. The number of runs used were 200 with length increments of 5 where the maximum length was 1000. Moreover, 20 tests were run for each sort per iteration to find the average.

## 0.3 Experiment 3

**Description:**

1. In this experiment we are testing the trend between the number of swaps of a randomly generated list (the graph is based on the number of swaps) and the runtime of three sorting algorithms based on the list. And for the number of swaps, the minimum number of swapping are 0 and the maximum swapping time would be based on the length of the list(length_of_list) and the equation: swaps = length_of_list*log(length)

2. (The question suggests using 5000 as the length of the list. I set the increment to be 1 but it took me 30 mins and still didn't get the result, so I made the length = 500.)

**Procedure:**

We made three lists "bubble" "insertion" and "selection" to store the result of runtime of different swap times. Notably the maximum swap times would be the rounding up of the equation: swaps = length*log(length) / 2 because it sometimes gives us value in type double. Based on the length and swap times, we generated three identical lists for the three sorting algorithms at each iteration(0 - round(length*log(length) / 2) ). Then append the runtime to the three lists respectively.

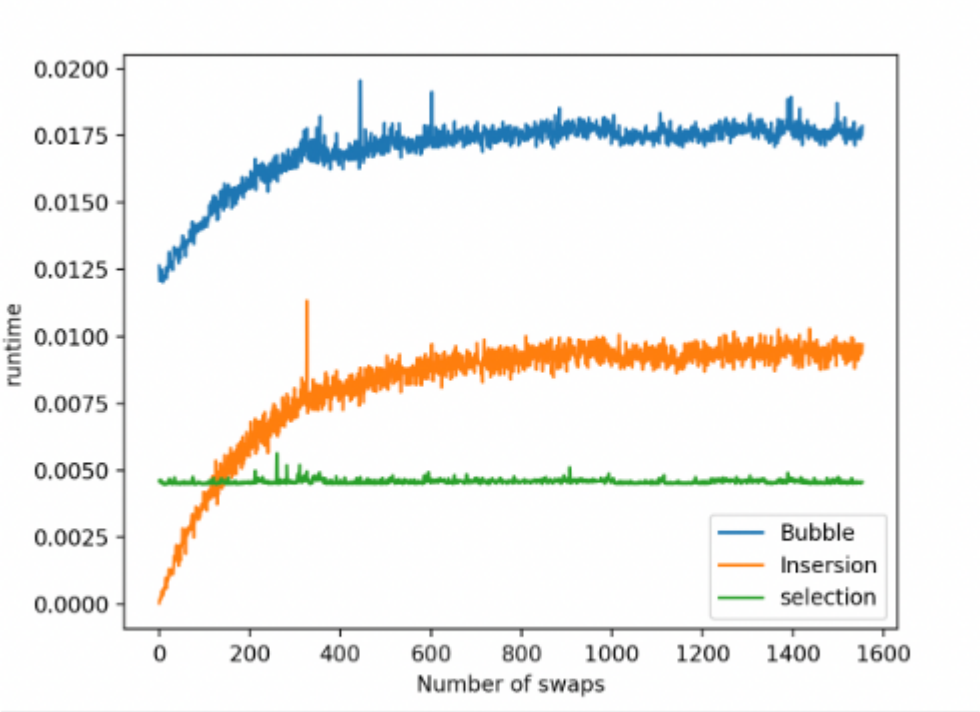



Figure 8: Experiment 3

Graph1: length = 500, maximum number of swap is 1554, which is the rounding value of 500*log(500)/2. We did 3 runs on each swap of different numbers to minimize the error to make sure the result is not an exception. Note: we used the original insertion sort, not the improved version.

**Conclusion:**

From the above graph, we can clearly see that if the number of swaps is equal to 0, which means the list is completely sorted, insertion sort performs really well. When the list became more and more unsorted, the performance went down. And although when the list is completely sorted, the performance of selection sort is worse than insertion sort, the runtime of selection sort isn't influenced by the degree of disorder of the list. The bubble sort is the worst one among those three sorting algorithms.

For selection sort, the runtime complexity of it is guaranteed to be $n^2$ because this algorithm disregards the relative sort-ness of the list and always processes the whole array. For insertion sort, this sorting algorithm

performs pretty well when the list is already sorted to some degrees. If it's a completely sorted list, the runtime complexity for insertion sort would just be "n" because it will just make one pass of the array and does nothing, no swapping, no re-assigning values. A thing we found from the graphs is: the runtime of insertion sort keeps increasing with the number of swapping reaching 500, after that threshold, the runtime of it becomes stable. The reason is: since insertion sort is good at sorted list, with the number of swapping increases and the degree of disorder increase, the performance of insertion sort went down. After that threshold, the list becomes very unsorted, the number of swapping and re-assigning required by insertion sort increases. For bubble sort, it is a pretty bad sorting algorithm because this algorithm needs to examine the comparison (if statement) and swapping of values many times, which are really time-consuming. The computer needs to examine the if statement at each iteration. We can see from the graph that when the number of swaps is 0, it still costs much more time than insertion sort although their time complexity is all $n^2$.

# Part - 2

## 0.4 Experiment 4

**Description:**

Compare the run times of quick, merge and heap sort with different parameters and determine which is the best sorting algorithm and give a conclusion as to why.

**Procedure:**

For figure 9, the size of the list ranges from 0 to 100. The maximum value in the list is 100. We ran each iteration 10,000 times and then took their average.

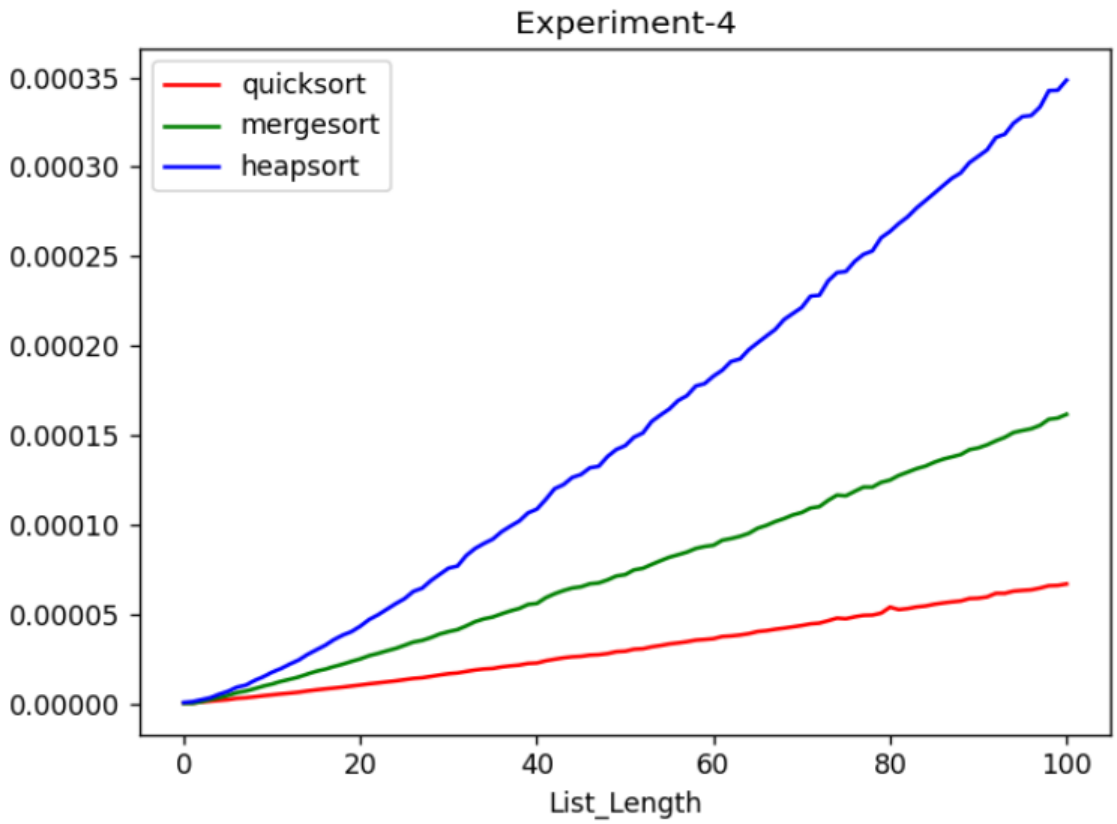



Figure 9: Experiment 4-1

In the graph below, the length of the list starts from 0 and goes to 10,000, the maximum value in the list is 10,000. We ran each iteration 100 times and then took their average. The goal of this test is to analyse the performance of algorithms for large lists.

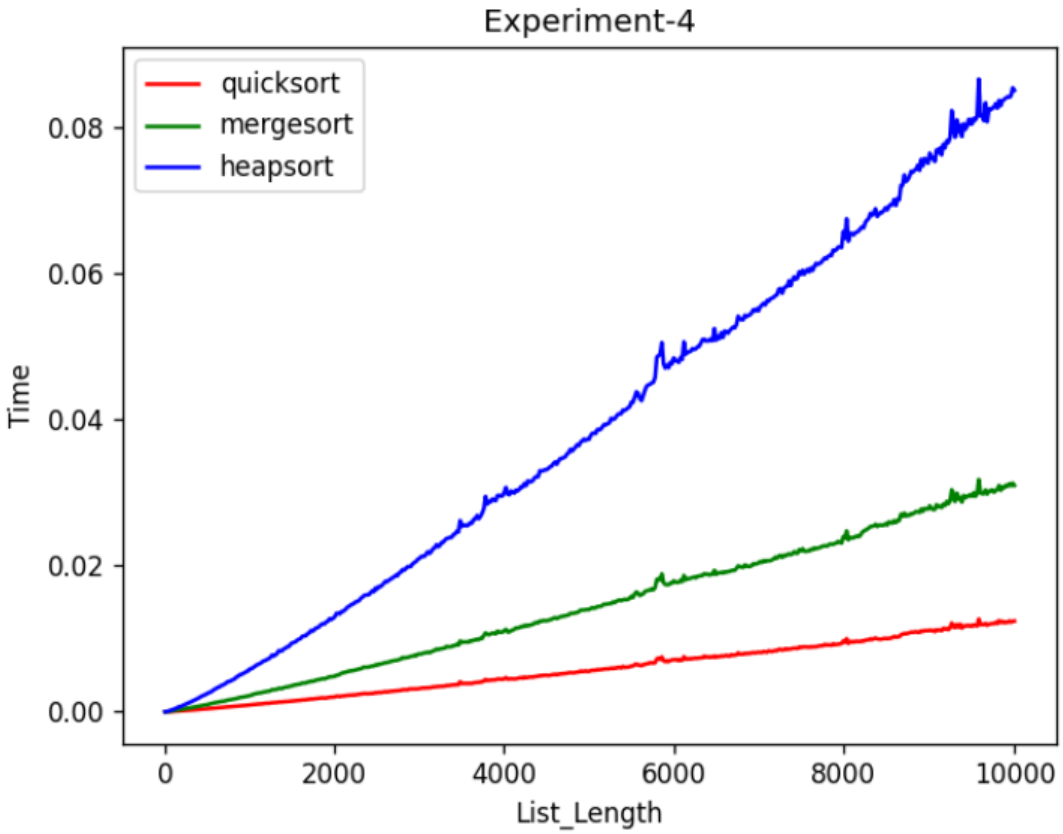



Figure 10: Experiment 4-2

For our next graph, the length of the list starts from 0 and goes to 50,000. But the maximum value in the list is only 100. The goal of this test is to analyse the performance of algorithms when there are a lot of duplicate values in the list.

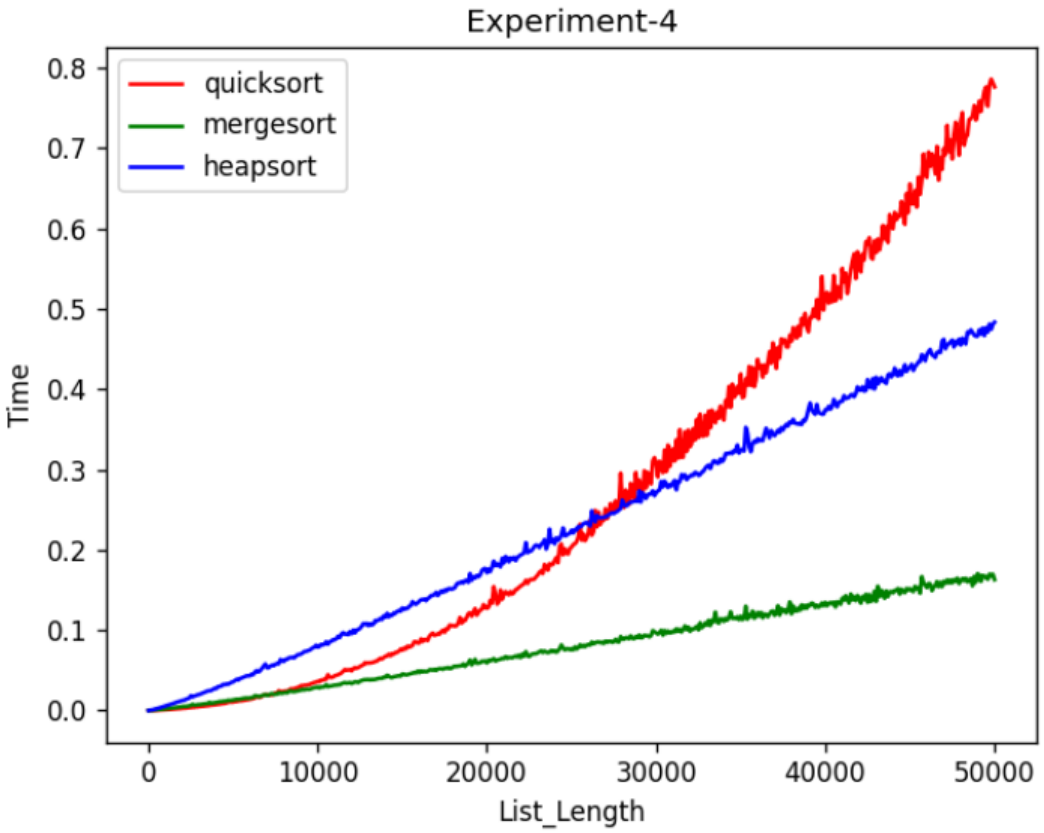



Figure 11: Experiment 4-3

For the graph below, the length goes to 1000, and the maximum value in the list is 100. We ran each iteration 100 times before taking their average. Instead of using a random list for this test, we first sorted a random list and then swapped 10 random values in the list. The goal of the test is to analyse the performance of algorithms when we have to sort a nearly sorted list.

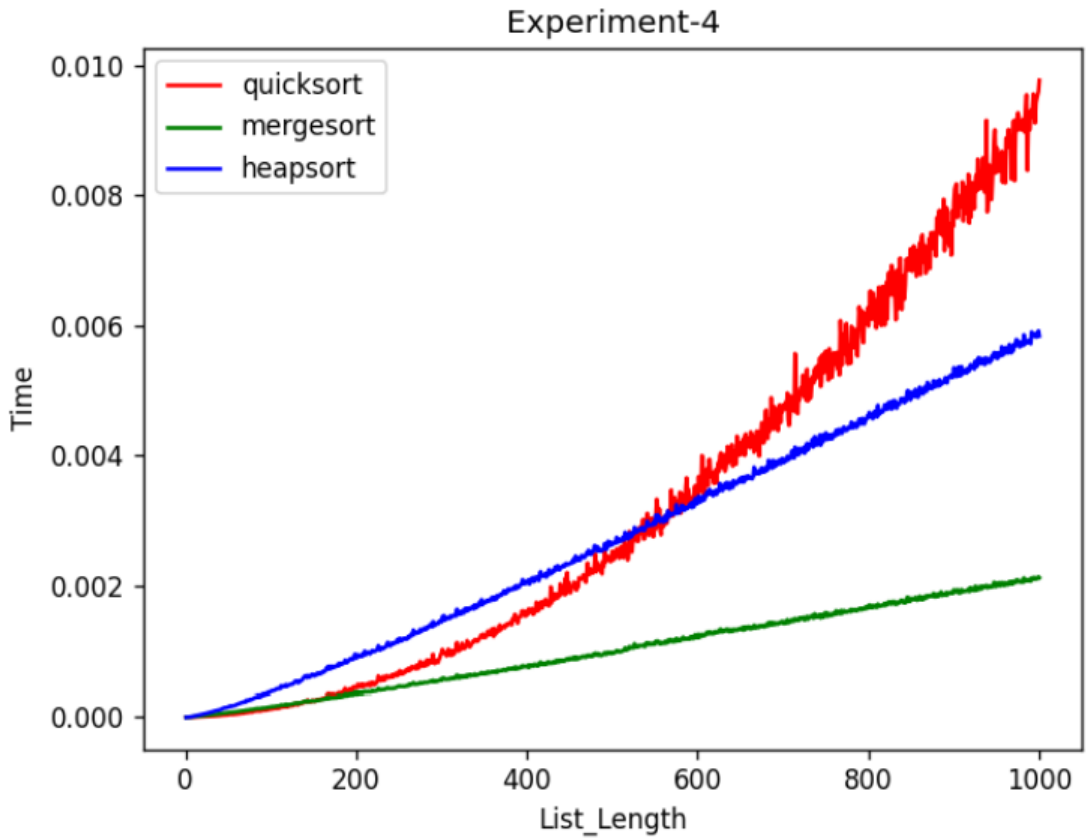



Figure 12: Experiment 4-4

For our last graph of this experiment, the length goes to 1000, and the maximum value in the list is 1000. We ran each iteration 100 times before taking their average. But this time we only swapped once in our already sorted list. The goal was to analyse the performance of our algorithms when the list is sorted except for only 1 swap.

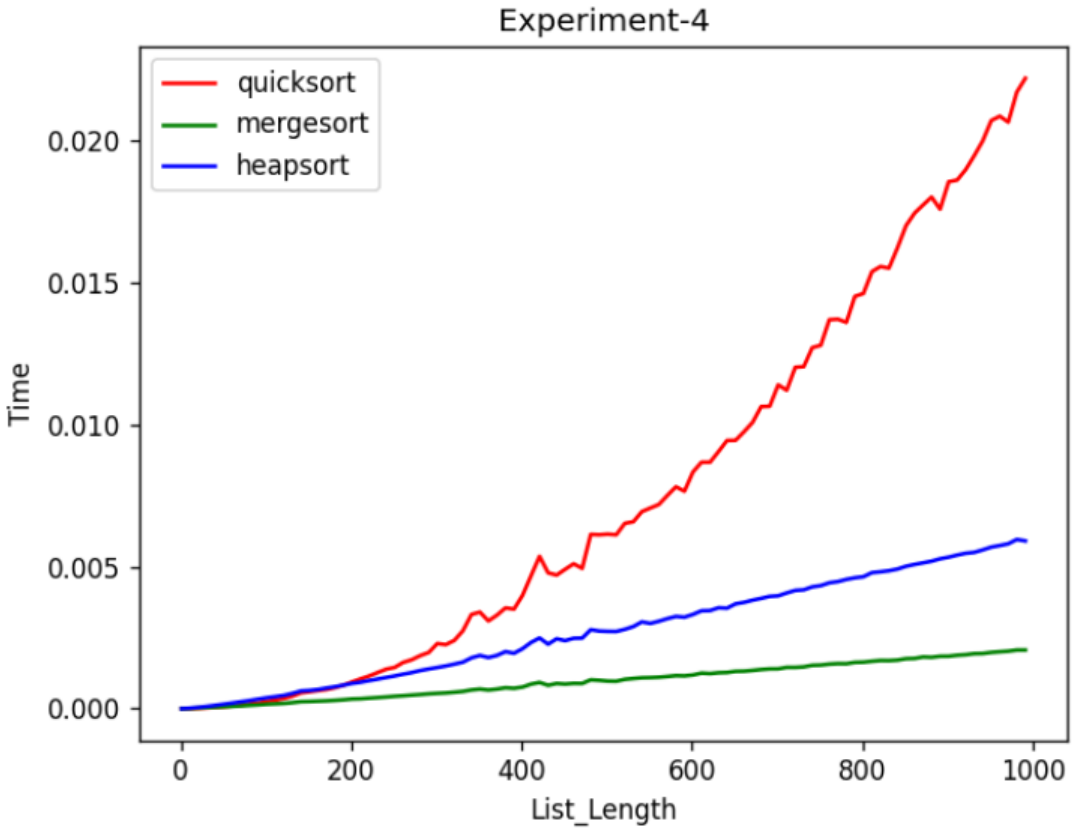



Figure 13: Experiment 4-5

**Conclusion:**

To conclude, When the list does not have many duplicate values and is not nearly sorted. heapsort performs the worst among the 3 algorithms. After that, mergesort performs better than heapsort but worse than quicksort which is the best-performing algorithm. if our list has a lot of duplicate values, then quicksort's runtime is better than heapsort initially but it gets worse as the length of the list increases, since the runtime complexity of quicksort shifts from $nlogn$ to $n^2$. Finally, if our list is nearly sorted with few random swaps, mergesort performs the best, heapsort's runtime is a little worse than mergesort. Quicksort's runtime is the worst and it gets even worse as the number of swaps decreases or the length of list increases or both. In conclusion, quicksort is the best algorithm if the list is not nearly sorted otherwise, mergesort is the best overall among these three sorting algorithms.

## 0.5 Experiment 5

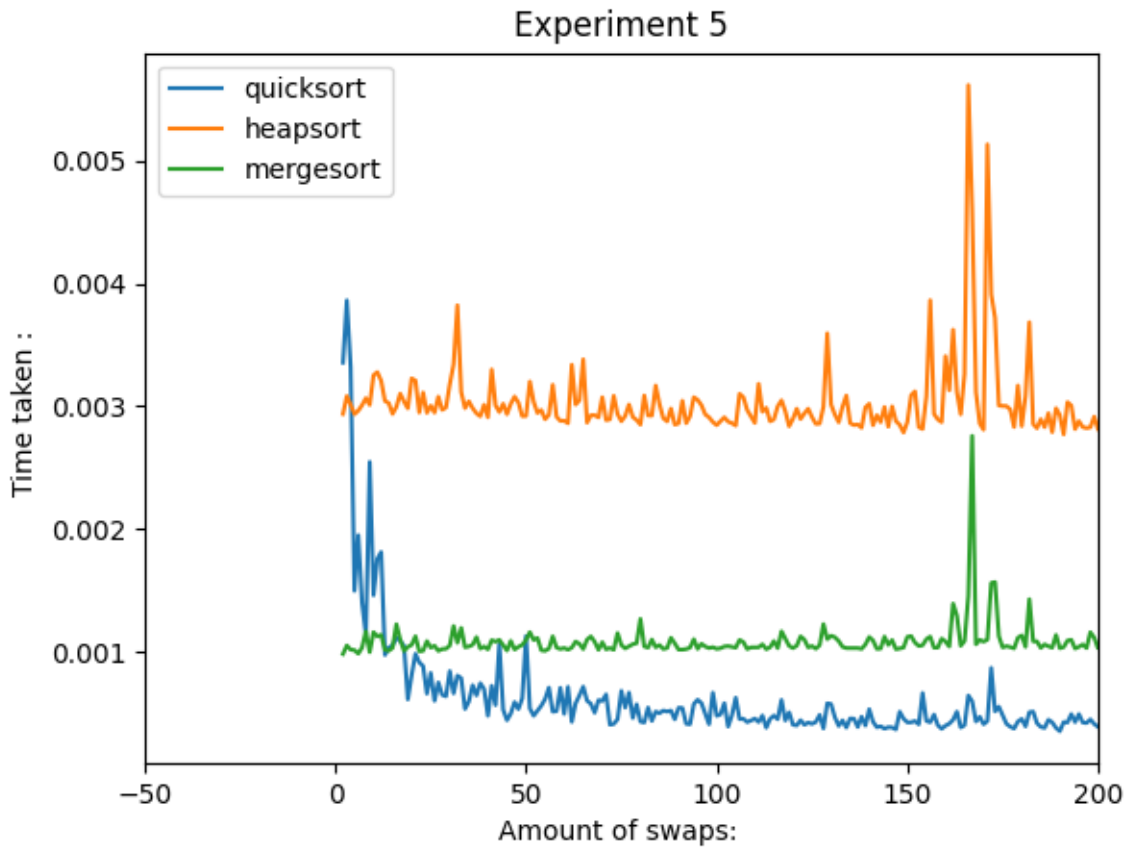
### Description:

The objective was to test the given implementation of quicksort which used the first element as the pivot. The number of swaps can affect the runtime of the quicksort algorithm, our task was to test this by comparing it to the other algorithms such as merge and heap sort and determine how unsorted a list has to be inorder for quicksort to be better.

### Procedure:

200 runs were performed where the average of 100 runs per algorithm was taken per iteration. The maximum value used was 1000 . The length of the list was taken as a constant of 100 for all 3 sorts . The highest limit of swaps was taken as 200 and was incremented by 1 for each iteration. The timeit module allows us to time taken to complete the 100 runs per iteration and the average in then appended to a list. There are 3 list namely - "quicksort1", "merge" and "heap" , there was also another list named "swp" which contained the number of swaps incremented by 20. The 3 lists were plotted with swp as the x axis and the time taken as the y axis.

Figure 14: Experiment 5




**Conclusion:**

We observe that the number of swaps affects the runtime of quicksort, it is always faster when the list is unsorted. . Quicksort is the worst algorithm for near sorted arrays, we observe that when there are less than 25 swaps, the algorithm performs significantly worse than the other two. Although it may perform worse with no swaps, when compared to the other sorts, quicksort catches up to the other sorts and is consistently better than mergesort and heapsort when the swaps are higher.

.

## 0.6 Experiment 6

**Description:**

The objective of this experiment is to implement a dual pivot quicksort algorithm and compare it to our traditional quicksort to determine if our new algorithm performs any better.

**Procedure:**

For figure15, t size of the list ranges from 0 to 100,000. The maximum value in the list is 100,000. We ran each iteration 10 times and then took their average.

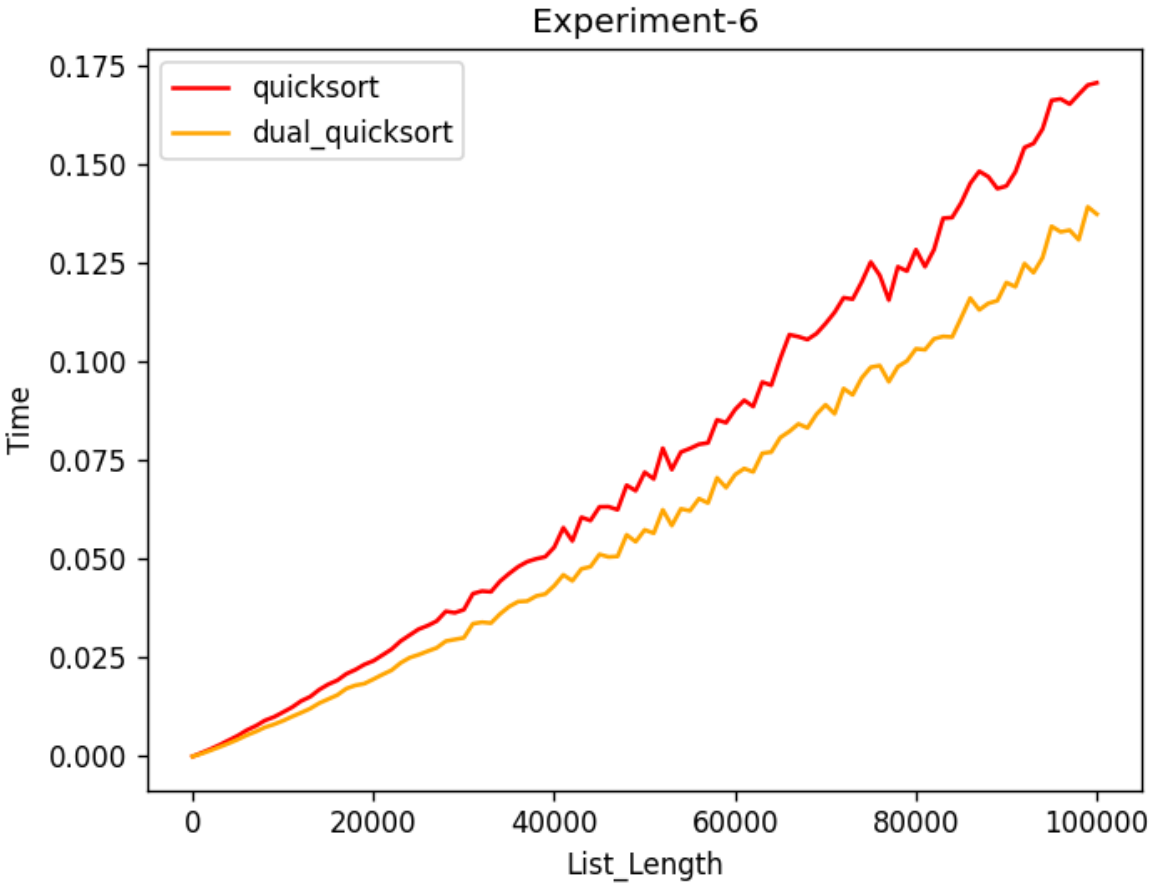



Figure 15: Experiment 6-1

In this test(figure 16), the length of the list starts from 0 and goes to 50,000. But the maximum value can only be 100. The goal of this test is to analyse the performance of algorithms when there are a lot of duplicate values.

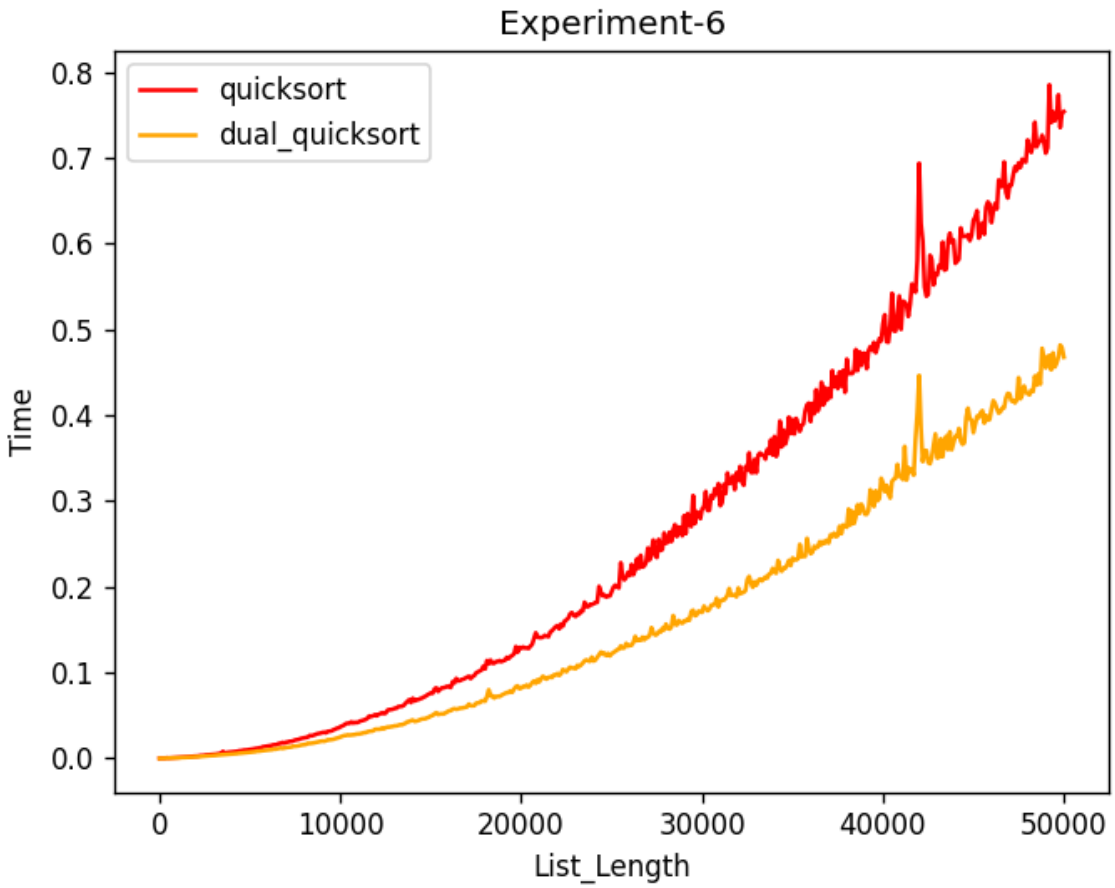



Figure 16: Experiment 6-2

For our next test (figure 17), the length of the list goes to 1000, and the maximum value in the list is 100. We ran each iteration 100 times before taking their average. Instead of using a random list for the test, we first sorted a random list and then swapped 10 random values in the list. The goal of the test is to analyse the performance of algorithms when we have to sort a nearly sorted list.

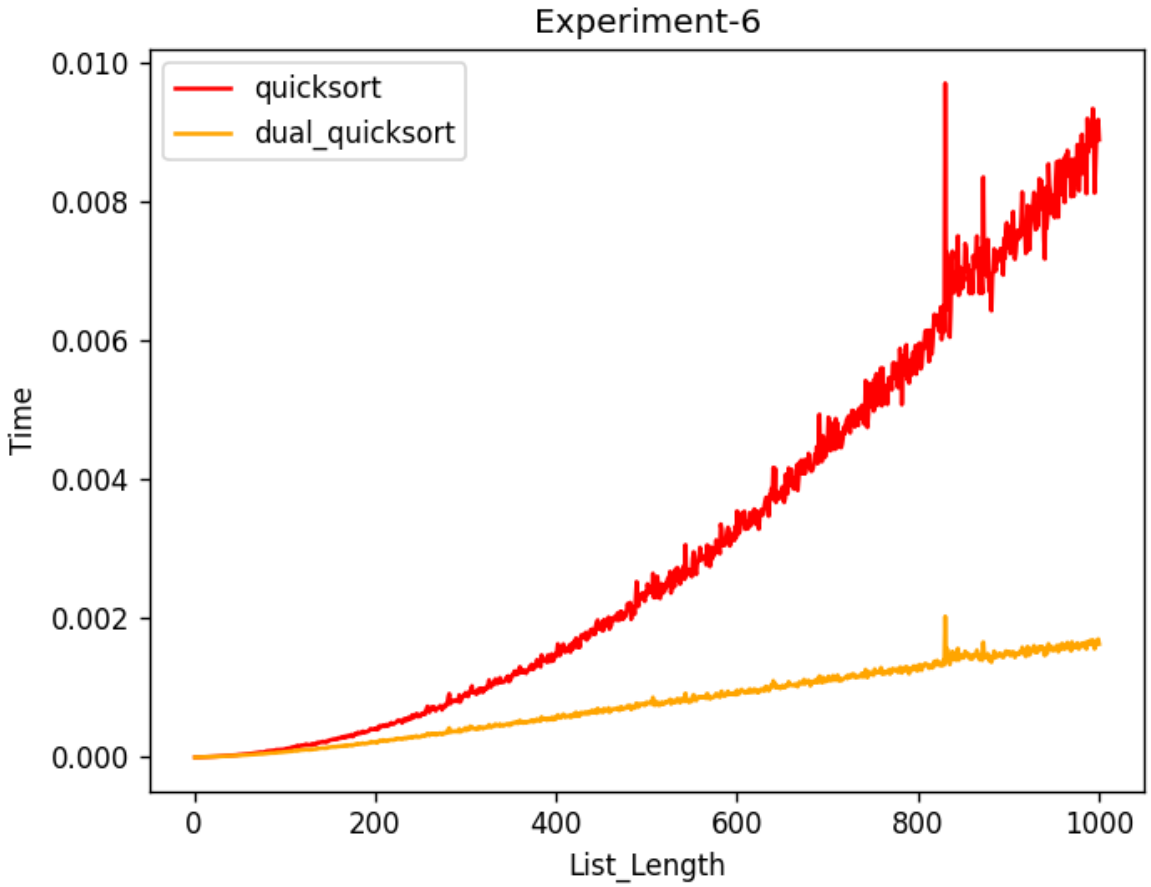



Figure 17: Experiment 6-3

For our last test of this experiment(figure18), the length goes to 1000, and the maximum value in the list is 1000. We ran each iteration 100 times before taking their average. But this time we only swapped once in our already sorted list. The goal was to analyse the performance of our algorithms when the list is sorted except for only 1 swap.

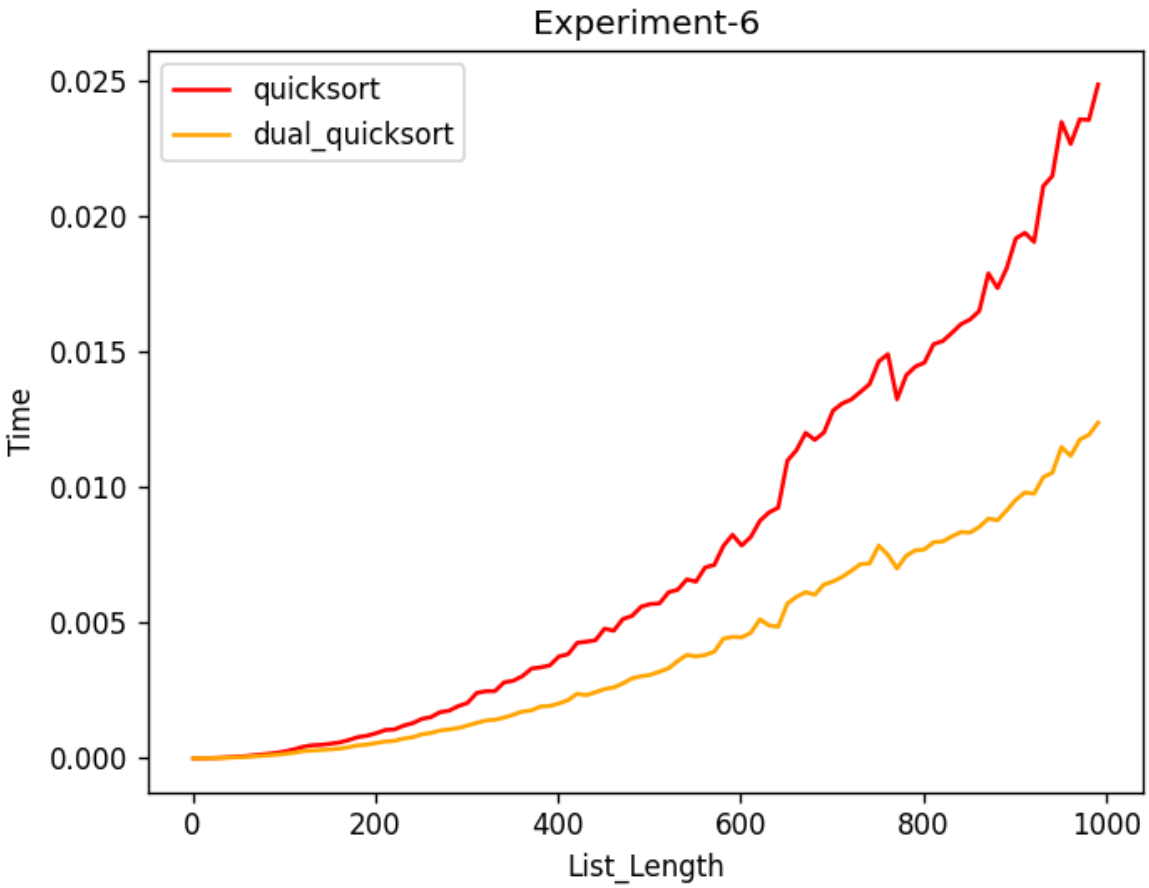



Figure 18: Experiment 6-4

**Conclusion:**

After running our tests of this experiment, we observed that our dual pivot quicksort performs better than a single pivot or traditional quicksort in every case. The difference in the performance widens when we have a nearly sorted list. From experiment 4, we concluded that quicksort is the worse when we have a nearly sorted list. But our new implementation of quicksort improves its weak leg. I would say increasing the number of pivots in quicksort is totally worth it.

## 0.7 Experiment 7

### Description:

The objective is to approach with merge sort with a different variation, rather than recursively splitting it, we iteratively go over the list and merge is together by taking a value n hence known as the bottom up merge .

### Procedure:

```python
def bottom_up_mergesort(arr):
    n = len(arr)
    size = 1
    while size < n:
        for start in range(0, n, 2*size):
            midpoint = start + size - 1
            end = min(start + 2*size - 1, n-1)
            left_half = arr[start:midpoint+1]
            right_half = arr[midpoint+1:end+1]
            arr[start:end+1] = merge(left_half, right_half)
        size *= 2
    return arr
def merge(left, right):
    L = []
    i = j = 0

    while i < len(left) or j < len(right):
        if i >= len(left):
            L.append(right[j])
            j += 1
        elif j >= len(right):
            L.append(left[i])
            i += 1
        else:
            if left[i] <= right[j]:
                L.append(left[i])
                i += 1
            else:
                L.append(right[j])
                j += 1
    return L
```

Figure 19: Experiment7Code

Instead of normal merge-sort, we avoid using recursion and just use iterations. We treat them as already sorted and then we merge pairs of arrays. The merging is usually done in $2^n$ ways until the whole array is merged and sorted. We can utilize the same merge function but we skip

the division of the large array into smaller ones.

**Conclusion:**

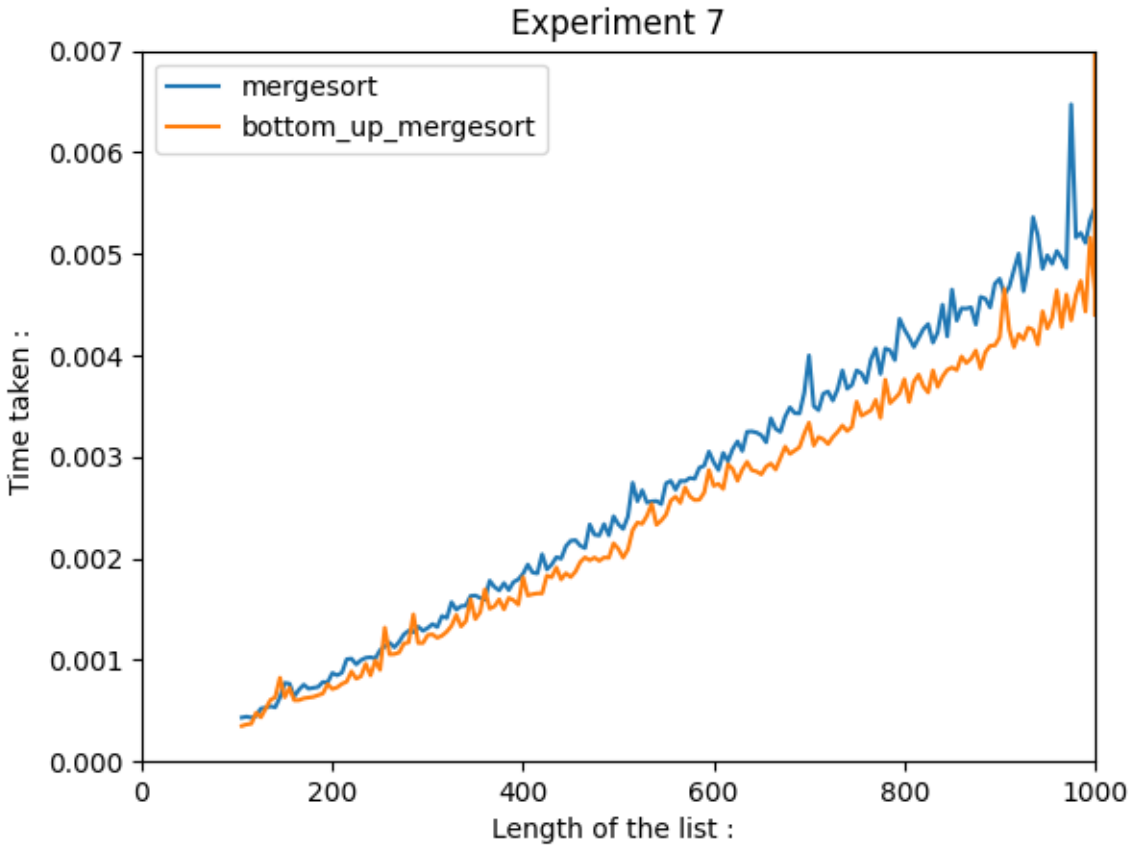



Figure 20: Experiment7

The performance is pretty comparable to the standard mergesort. Since both algorithms time complexity is O(n log n) and the space complexity is 2N. We do observe a slight increase in performance for the bottom up merge sort when the size of the list is $\leq 600$, the gap in run time just keeps increasing with the size of the list. When the iterative vs recursive approach of binary search is because of its fundamental approach to dividing an array. In the iterative approach of a binary search the algorithm splits the array until the value is either found or is not present in the array , this method reduces overhead function calls whereas, recursive calls have more overhead function and a call stack which makes the algorithm less efficient. When considering mergesort, it is most definitely

recursive by nature since it involves splitting / division of the array. The reason for why bottom up merge sort might be better is because that, the dividing step has less to do with the runtime than that of the merging step. The bottom-up approach may have a slightly better runtime than the top-down approach because it avoids the overhead of recursion and reduces the need for function calls.

## 0.8 Experiment 8

### Description:

In this experiment, we are working on comparing the runtime for the three sorting algorithms with small lists! We still need the "create_random_list" function and "create_near_sorted_List" functions because the degree of disorder will affect the result relatively.. In order to prevent the result we get is a coincidence, I make 2000 runs to reduce potential errors!

### Procedure:

The size of the list could be in the span of 0 to 50 with the random value from 0 to 100. And we made three lists to store the result. And there's still an external function to help produce the solid results.

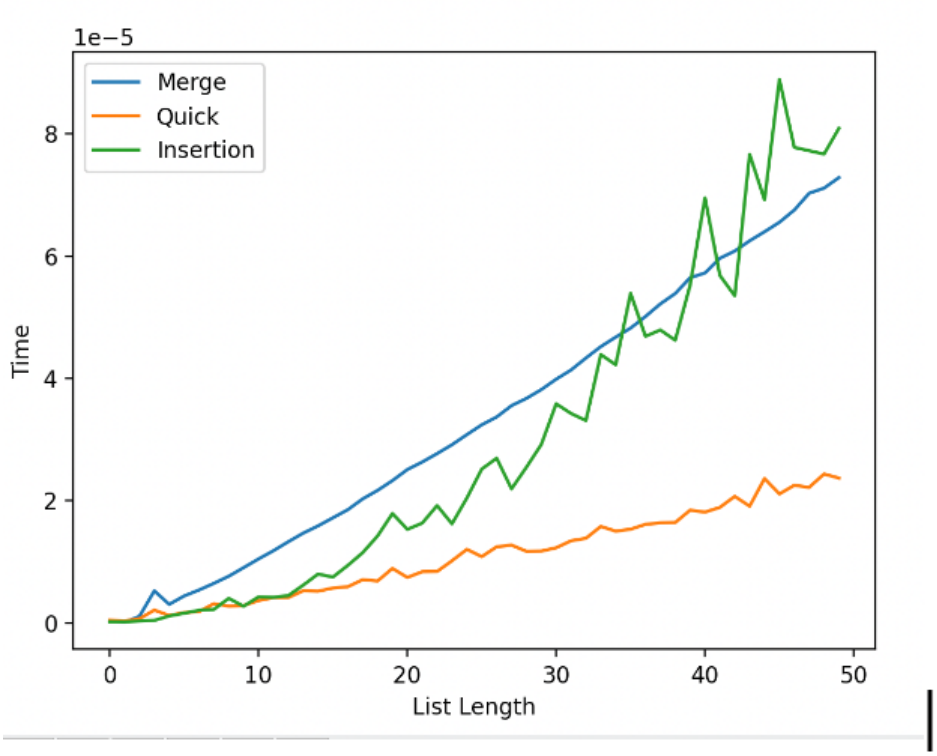



Figure 21: Experiment 8_1

In the above Figure 21, the list is randomly generated. We can see that when the length of the list is 0, the runtime of those three are the same, which makes sense. In this diagram, we can clearly see that Quicksort performs the best among those three sorting algorithms. From this graph, we can roughly get that when the size of the randomly generated array is from 0 to10, Insertion sort is as good as Quicksort, and when the size of the array is between 0 to 35, insertion sort performs better than Mergesort. When the size of the array gets to 35, it performs worse and worse.

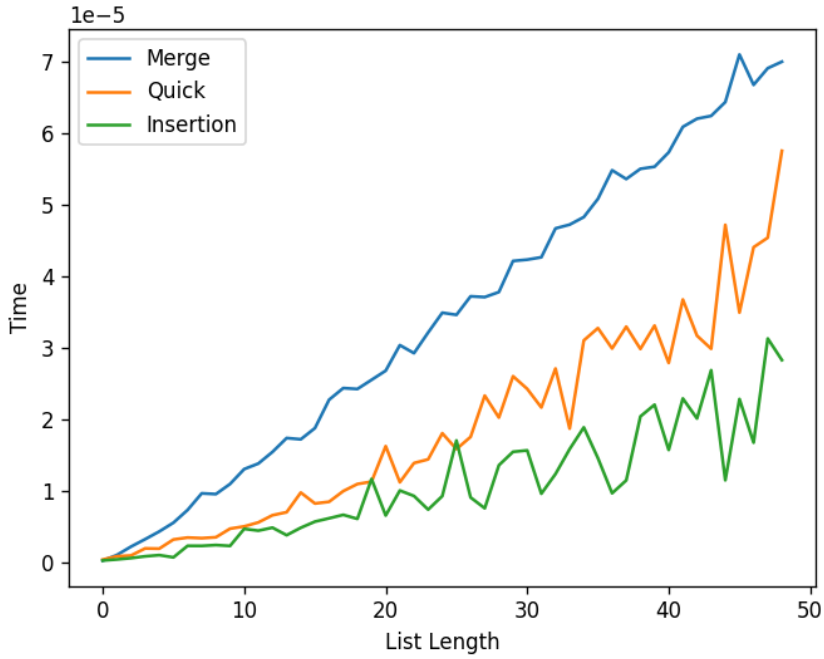



Figure 22: Experiment 8_2

For the above Figure 22, the lists of length from 0 to 50 are all sorted before putting them in sorting algorithms. And there's no major difference between those two graphs.

**Analysis/Conclusion:**

Quicksort performs well regardless of the size of the list in this graph. Mergesort performs better on large size lists. And when the size of the list is small, insertion sort performs nearly as good as Quicksort. Getting to know their advantages and disadvantages is very important because it can help us cope with different situations, like when we need to apply insertion sort, when we can combine two sorting algorithms to achieve greater runtime complexity. For example, combine merge sort and insertion sort to a new sorting algorithm, which contains the advantages of those two sorting algorithms that minimizes the worst case time complexity for smaller length of a list, $\leq 35$ in this case.

# Appendix

## 0.9 How to navigate the code:

**Experiment1:**

Open the file "Lab1_exp1.py" and run the code. There are 5 tests and 4 of them are commented out. The function test1 creates a random list whereas test2 creates a nearly sorted list.

**Experiment2:**

The file name "Lab1_exp2.py" contains the functions "SelectExp2" and "bubbleExp2" which take the parameters max_runs, max_iter, max_length,max_value. These both produce graphs of both the Selection sorts and bubble sorts respectively. There is also a completion meter to help know the progress.

**Experiment3:**

Open the file "Lab1_exp3.py", and run the code. Everything is there. The list length is set to 500 and each data we run 3 times and get the average to ensure the accurateness.

**Experiment4:**

Open the file "Lab1_exp4,6.py" and run the code. There are 5 tests and 4 of them are commented out. The function test1 creates a random list whereas test2 creates a nearly sorted list. Functions test3 and test4 are for experiment-6.

**Experiment5:**

By opening the file "Lab_exp57.py", it contains the function "exp5" which takes in the parameters max_iter,runs,max_length,max_value,max_swaps. This can be used to test the 3 functions with predefined max_swaps. The output is the graph of 3 sorting algorithms with no of swaps on the x axis and the time taken on the y axis.

**Experiment6:**

Open the file "Lab1_exp4,6.py" and run the code. There are 4 tests and 3 of them are commented out. The function test3 creates a random

list whereas test4 creates a nearly sorted list. Functions test1 and test2 are for experiment-6.

**Experiment7:**

By opening the file "Lab_exp57.py", it contains the function "exp7" which takes in the parameters max_iter,runs,max_length,max_value. This can be used to test the 2 merge functions with predefined max list length. The output is the graph of 2 merge sorting algorithms with the lenght of list on the x axis and the time taken on the y axis.

**Experiment8:**

Open the file "Lab1_exp8.py" and run the code. The list length is set from 0 to 50 and the list will be randomly generated, that means the degree of disorder is not guaranteed.

# End of Lab Report

*–Experiments Performed:–*
*Yu Chang : 3,8.*
*Suhaas : 2,5,7*
*Sandhu - 1,4,6 .*

*Group 20, L01.*