# Lab Report 1: **Analysis of Sorting Algorithms**

Akash Santhanakrishnan, Paarth Kadakia, Virendra Jethra

February 8 2023

# Contents

# List of Figures

# 1 Introduction

## 1.1 Overview

The lab focuses on implementing, analysing, and optimizing some traditional sorting algorithms which covers but not limited to:

- Runtime comparisons between sorting algorithms (Bubble, Insertion)
- Experiments of different variations of each sorting algorithm
- Analyse performance of algorithms for specific edge cases
- Determine if *hybrid* search strategy would be beneficial
- Implementation of sorting algorithms via strategy pattern

## 1.2 Purpose

This lab report conducts and discusses a variety of different experiments with these traditional sorting algorithms, specifically, Bubble, Selection, Insertion, Merge, Quick, and Heap. There will be code, graphs, and other visuals to supplement the analysis of these algorithms.

# 2 Executive Summary

The lab report focuses on the implementation, analysis and optimization of traditional sorting algorithms, including Bubble, Selection, Insertion, Merge, Quick, and Heap. It examines the run time comparisons between algorithms, their performance in specific edge cases, and the possibility of an optimized sorting strategy. The report includes 8 experiments, each with different graphs and visuals to supplement the analysis. The conclusion for each experiment highlights the difference in performance between the algorithms and the impact of different parameters such as the number of pivots and list size on their performance. Also through this lab, we found that it's possible to have a search strategy that combines insertion sort and quick sort. This hybrid sort can make use of the best features of both algorithms to achieve improved performance. However, it's worth noting that the implementation of such a hybrid sort can be complex and may not necessarily result in significant performance improvements in practice, especially for small input sizes. In such cases, it may be better to simply use one of the two algorithms alone, depending on the specific requirements and characteristics of the input data.

# 3 Lab Part 1

**Note:** For all experiments in this report, time was recorded in seconds and list size was simply the number of elements in the list.

## 3.1 Experiment 1

***Outline:***

- Comparing "bad" sorting algorithms
- Three different graphs with list sizes: **10, 100, 1000**
- Number of runs: **1000**

***Graphs:***

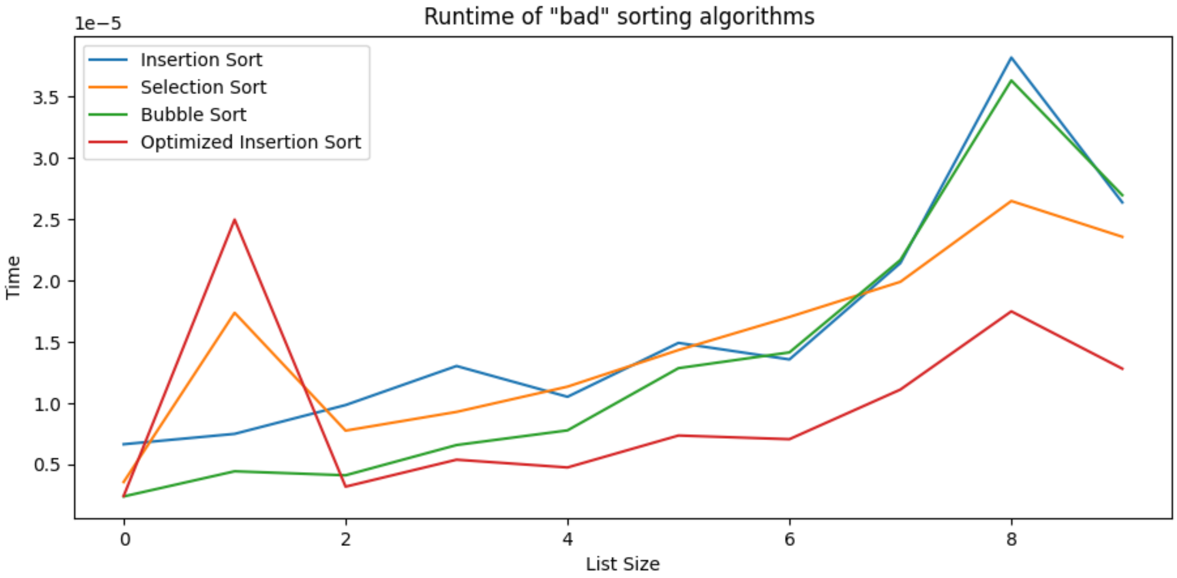

**Figure 1:** List Size (10) vs Time

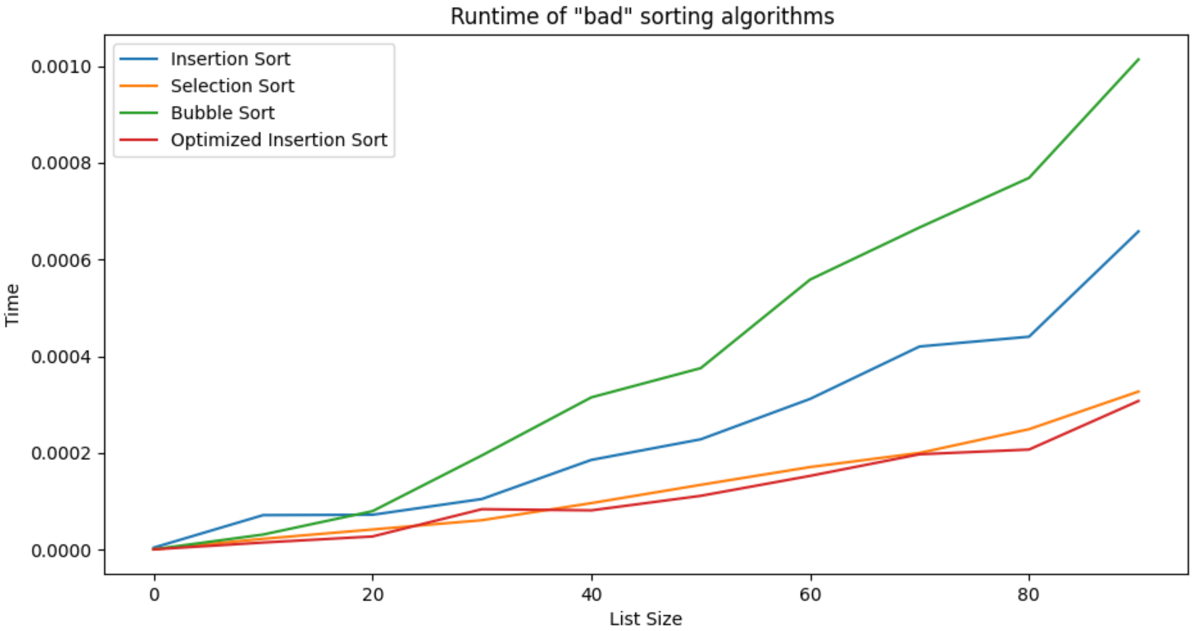

**Figure 2:** List Size (100) vs Time

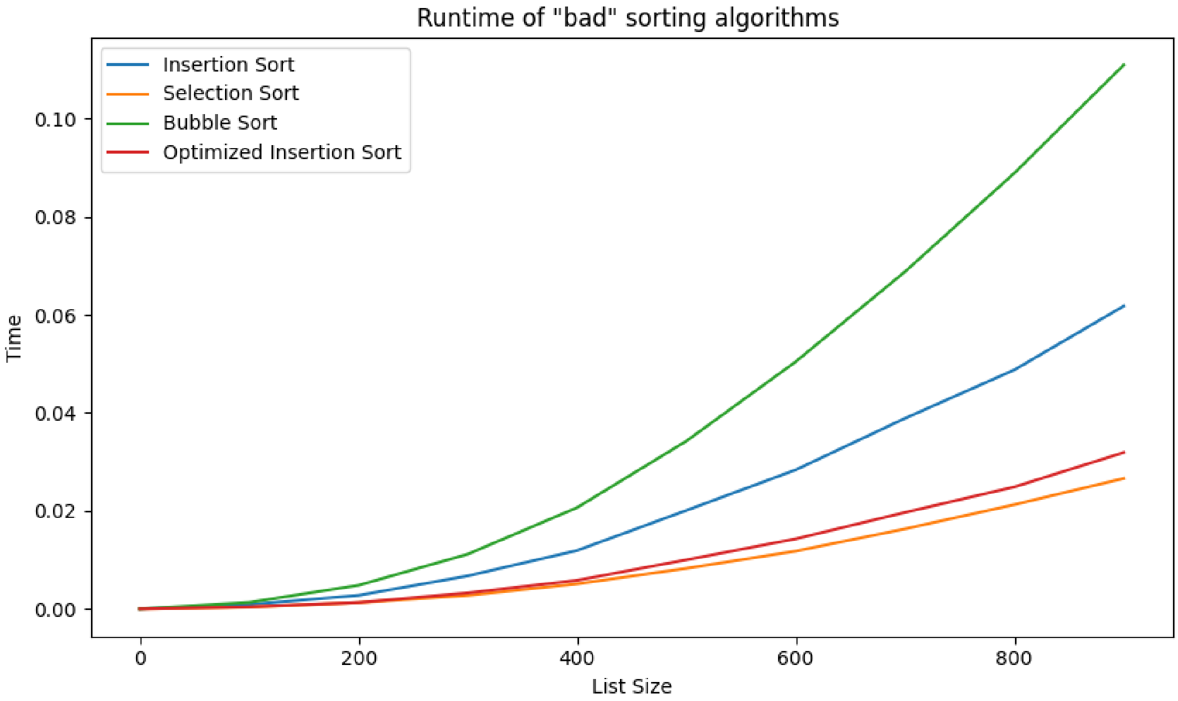

**Figure 3:** List Size (1000) vs Time

***Conclusion:***
For the small and normal list sizes, insertion and bubble sorts are faster than selection sort as they perform better when the list is closer to being sorted. As the list size increases, the growth of the sorting algorithms becomes clear. The growth of selection sort is less than bubble and insertion sorts showcasing how its run time is faster for larger list sizes and performs better on data sets that are more randomized.

## 3.2 Experiment 2

***Outline:***

- One graph compares traditional selection sort with an optimized selection sort algorithm
- One graph compares traditional Bubble sort with an optimized Bubble sort algorithm
- List Size: **1000** | Step: **100**
- Number of runs: **1000**

***Graphs:***

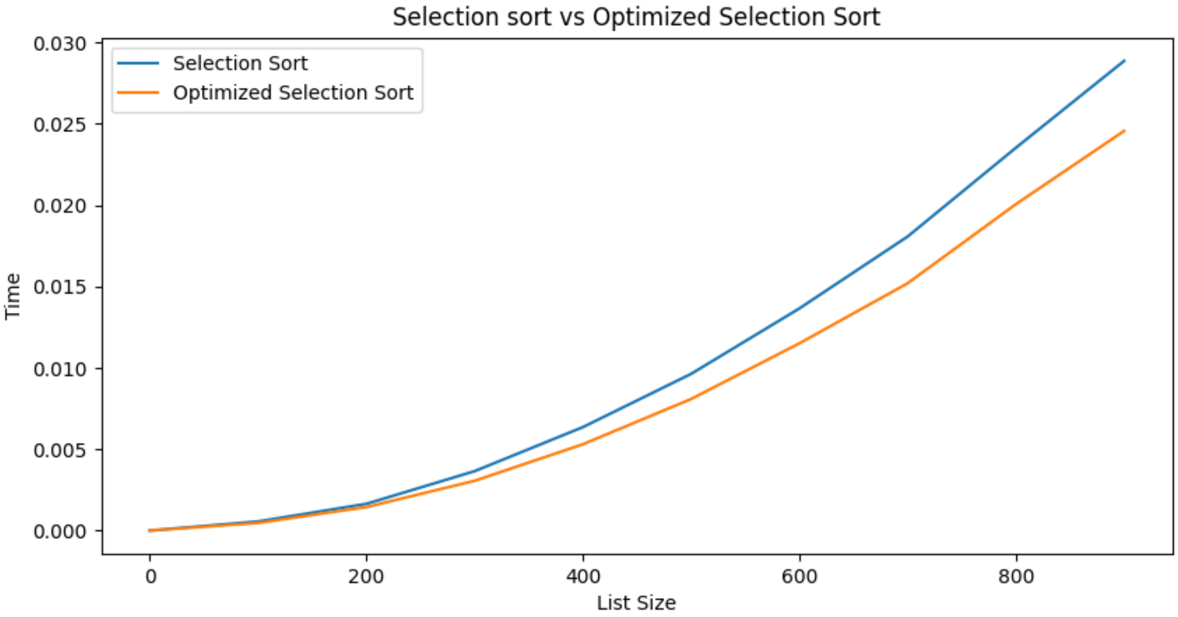

**Figure 4:** List Size (1000) vs Time

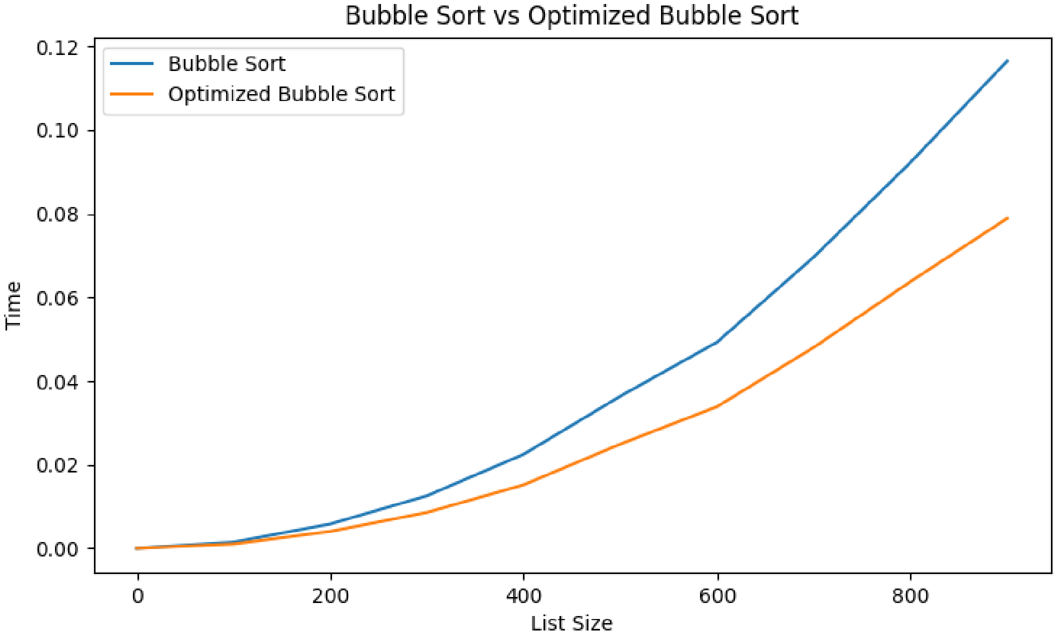

**Figure 5:** List Size (1000) vs Time

***Conclusion:***
The graphs clearly demonstrate the speed increase with the optimized sorting algorithms compared to their traditional counterparts. Although the growth looks the same, the traditional algorithms have more constant time operations than the optimized algorithms which are reflected in the graphs

### 3.3 Experiment 3

***Outline:***

- Looking at how sortedness of a graph impacts these optimized sorting algorithms
- One graph that compares time and number of swaps (more swaps means the array is more randomized)
- Number of Swaps: **3000** | Step: **500**
- Number of runs: **100**
- List Size(Constant): **5000**

***Graph:***

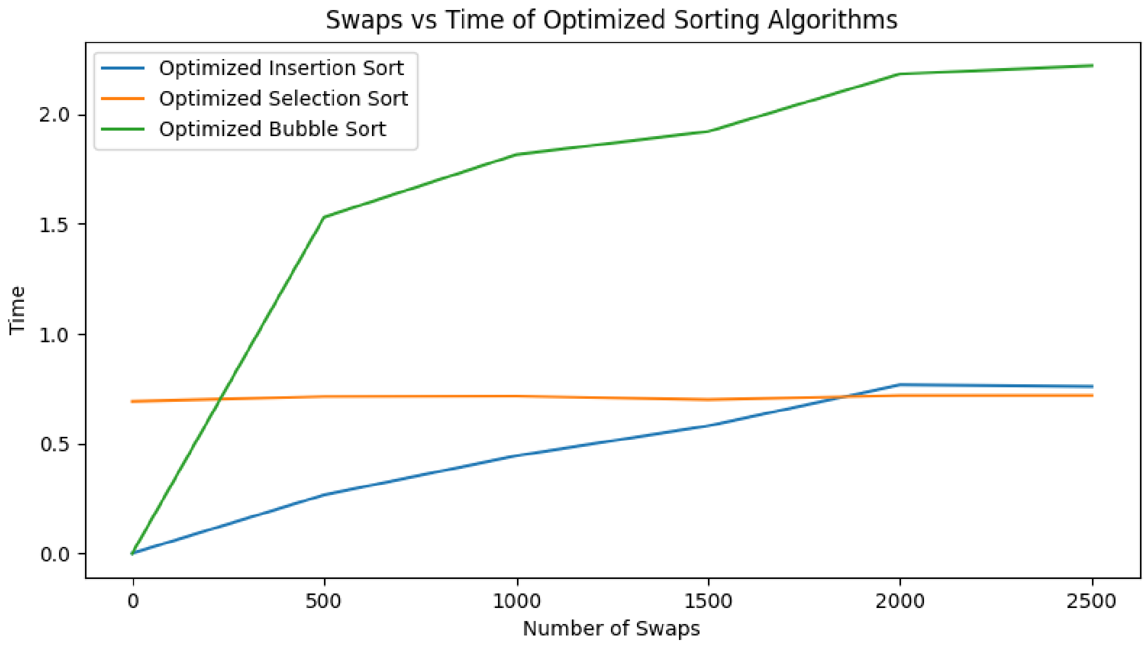

**Figure 6:** Number of Swaps (3000) vs Time

***Conclusion:***
Insertion and Bubble seem to perform well when the list is close to being sorted but decreases in performance as the randomization increases. Selection sort performance seems to be constant based on the number of swaps which makes sense as swapping cost does not matter and it is compulsory to check all the elements once.

# 4 Lab Part 2

## 4.1 Experiment 4

***Outline:***

- Comparing ”good” sorting algorithms
- Three different graphs with list sizes: **10, 100, 1000** | Steps: **1, 10, 100**
- Number of runs: **1000**

***Graphs:***

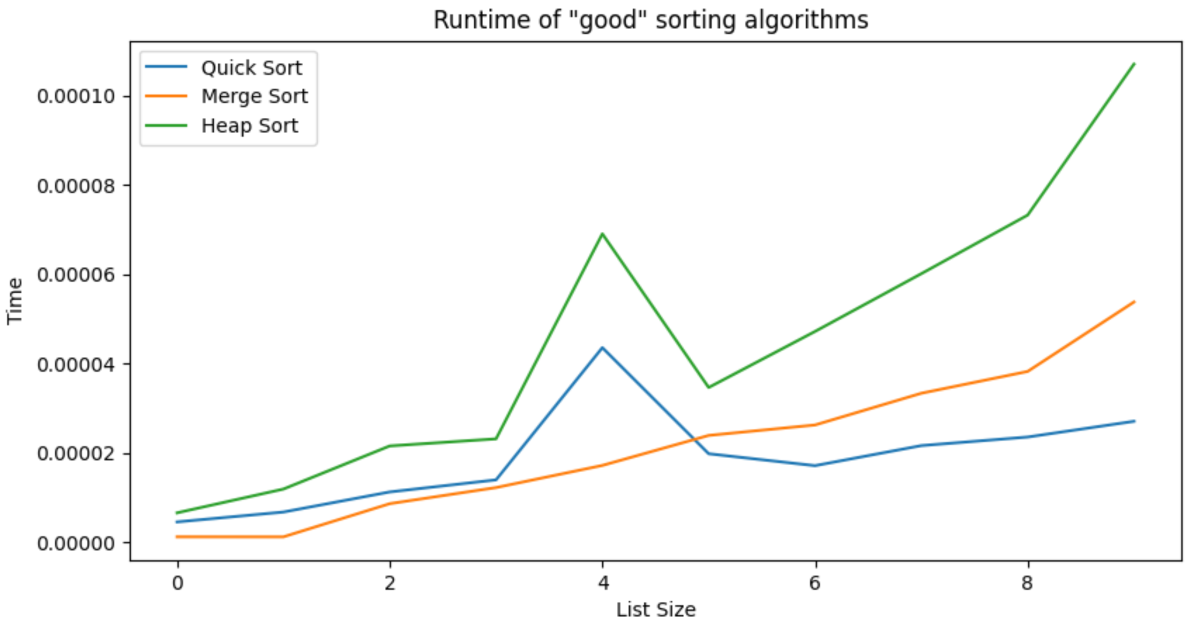

**Figure 7:** List Size (10) vs Time

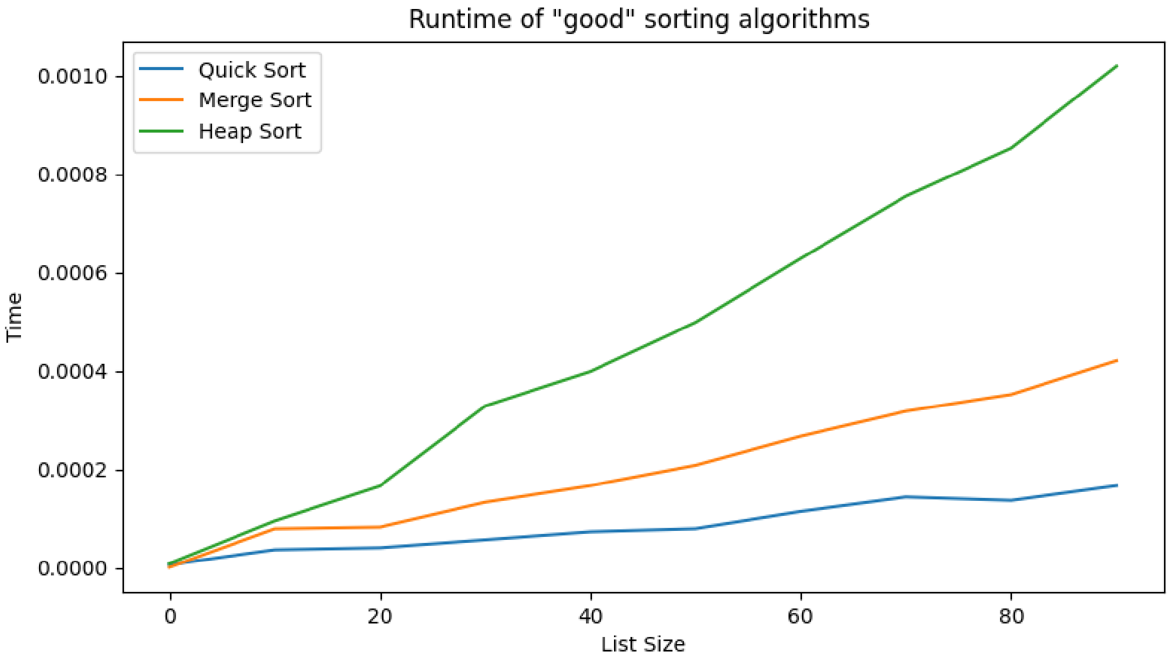

**Figure 8:** List Size (100) vs Time

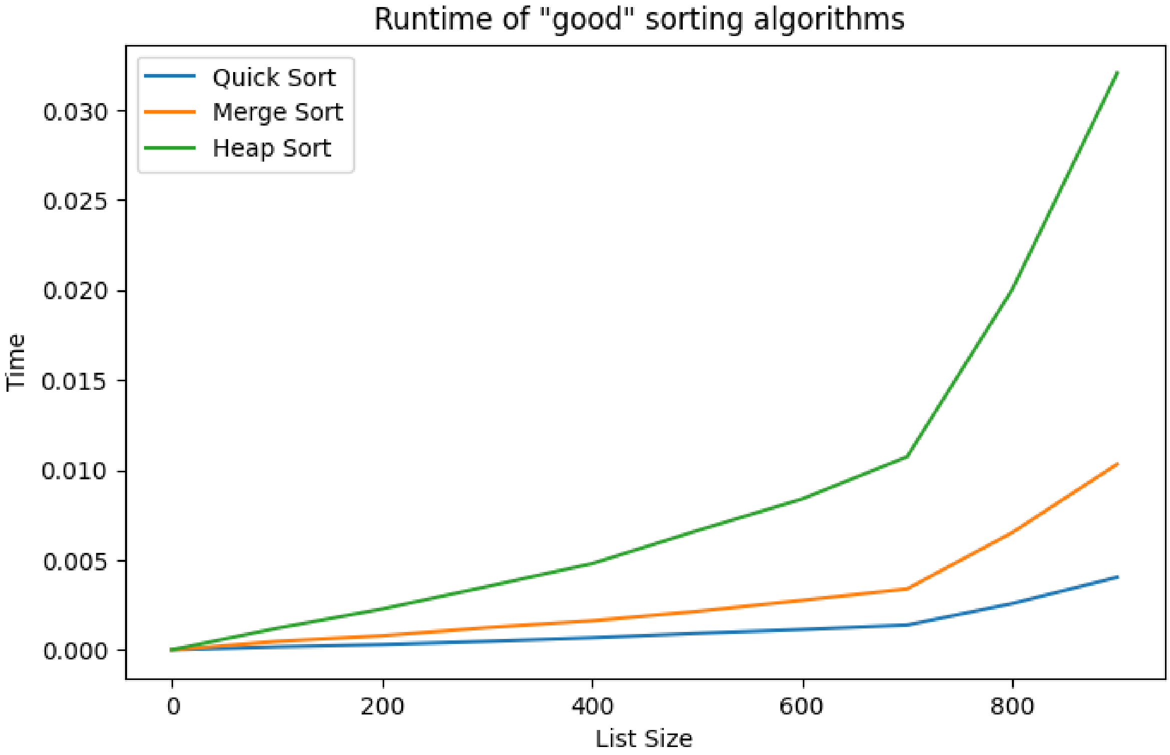

**Figure 9:** List Size (1000) vs Time

***Conclusion:***
Although Merge performs marginally better than Quick for very small list sizes, Quick outperforms both Merge and Heap for large data sets.

### 4.2 Experiment 5

***Outline:***

- Looking at how non-sorted does a list need to be for Quick Sort to be efficient?
- Number of swaps: **100** | Steps: **10**
- Number of runs: **1000**
- List Size (Constant): **5000**

***Graphs:***

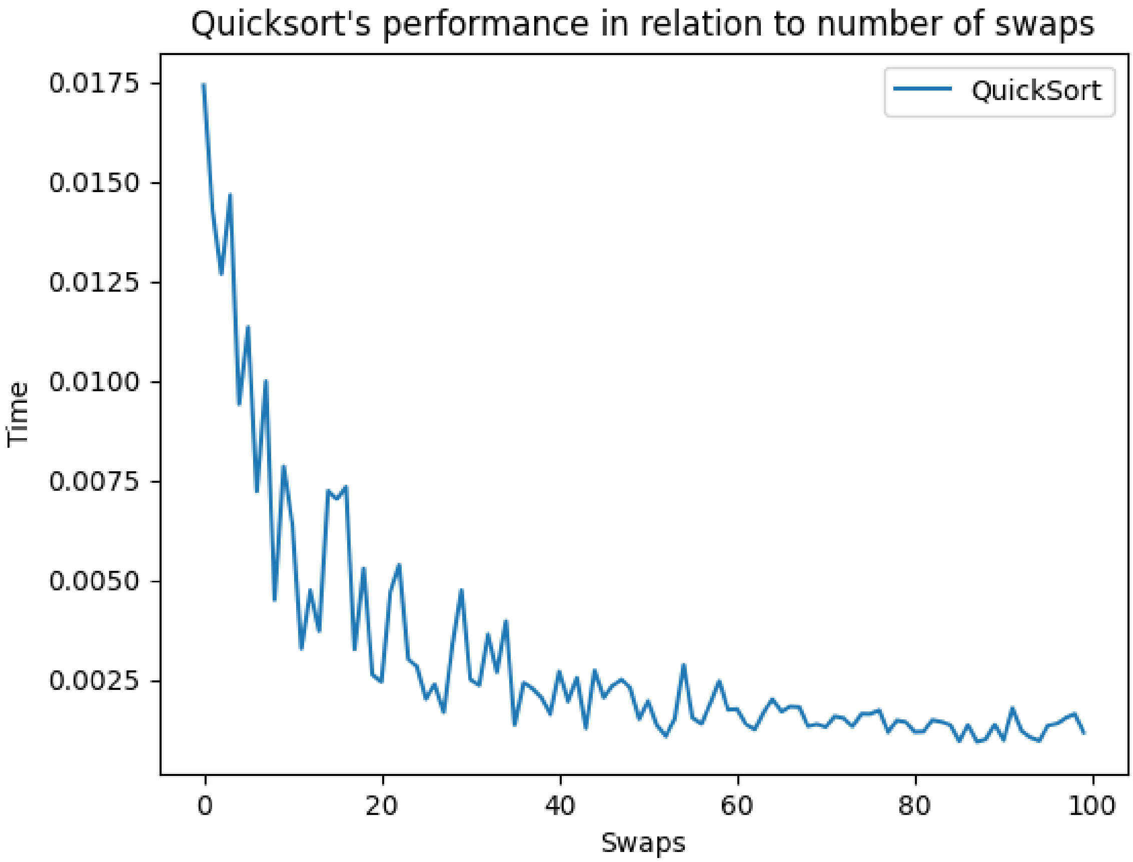

**Figure 10:** Number of Swaps(100) vs Time

***Conclusion:***
It seems once the data set is randomized by at least 40 swaps, the performance of Quick Sort starts to get efficient.

### 4.3 Experiment 6

***Outline:***

- Does the number pivots impact/improve the performance of Quick Sort?
- List Size : **100, 3000** | Step: **10,500**
- Number of runs: **1000**

***Graphs:***

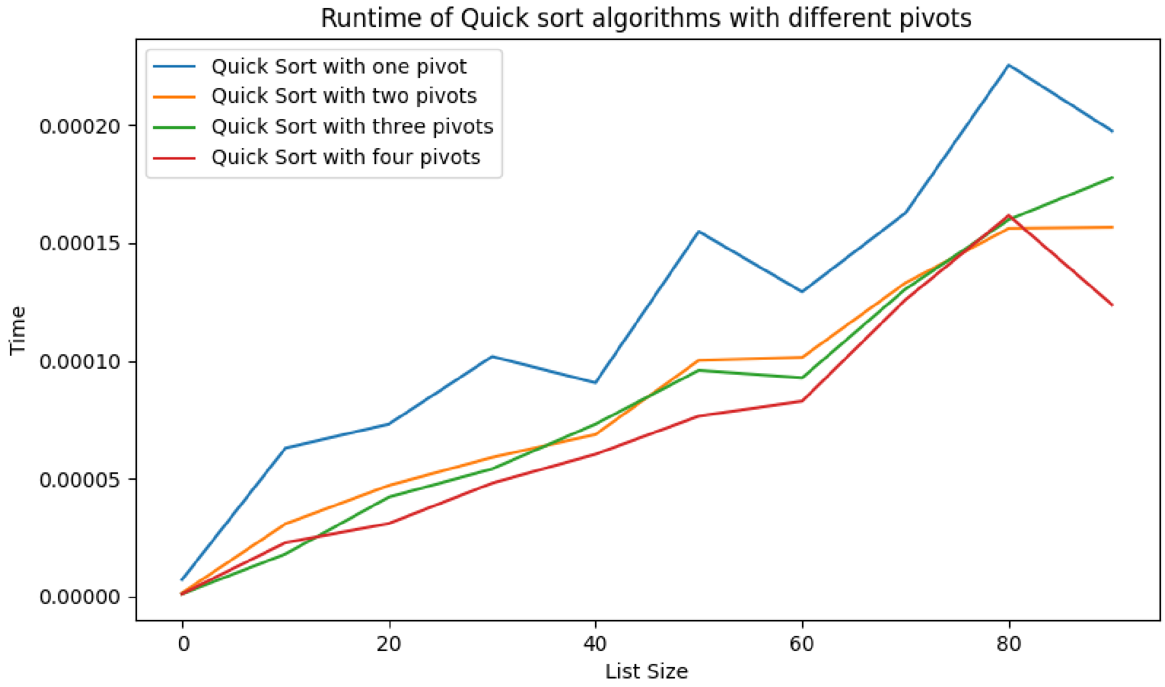

**Figure 11:** List Size (100) vs Time

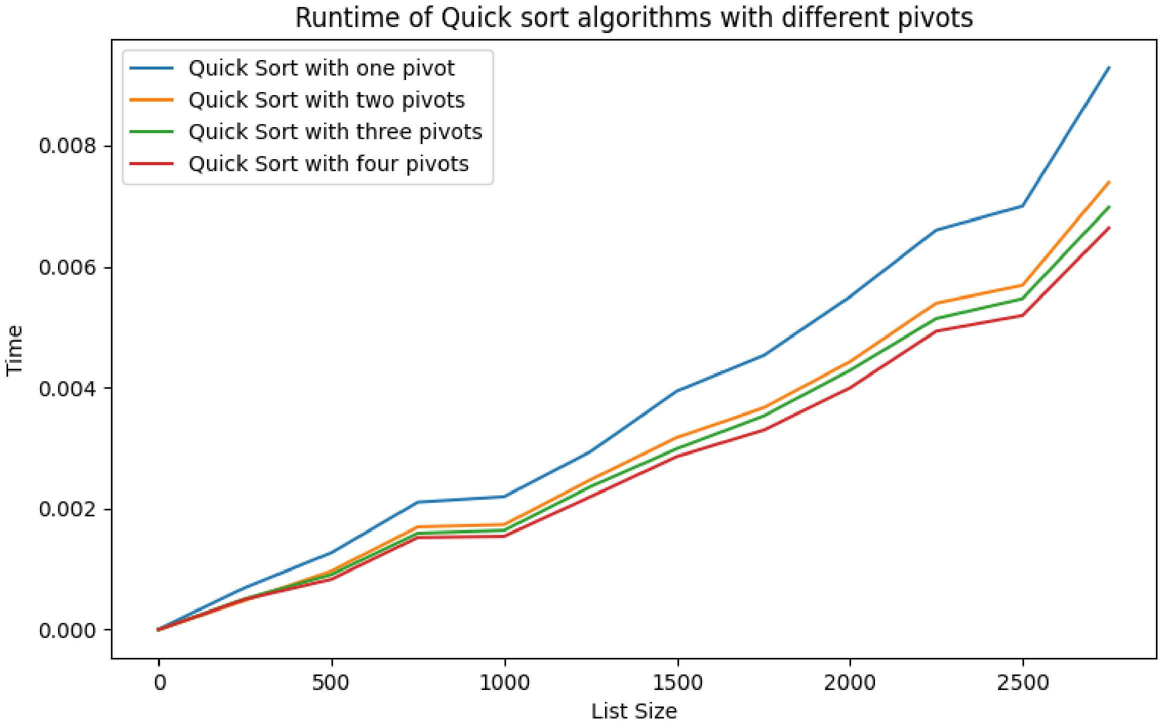

**Figure 12:** List Size (3000) vs Time

***Conclusion:***
The performance of the Quick Sort algorithm seems to improve as the number of pivots increases. Although

their growth is the same, the number of constant operations is less for the algorithms that have more pivots because their arrays are subdivided into smaller arrays.

## 4.4 Experiment 7

***Outline:***

- Comparing Merge sort with Bottom Up Merge sort
- Two graphs with Size Length: **100, 3000** | Steps: **10, 250**
- Number of Runs: **1000**

***Graphs:***

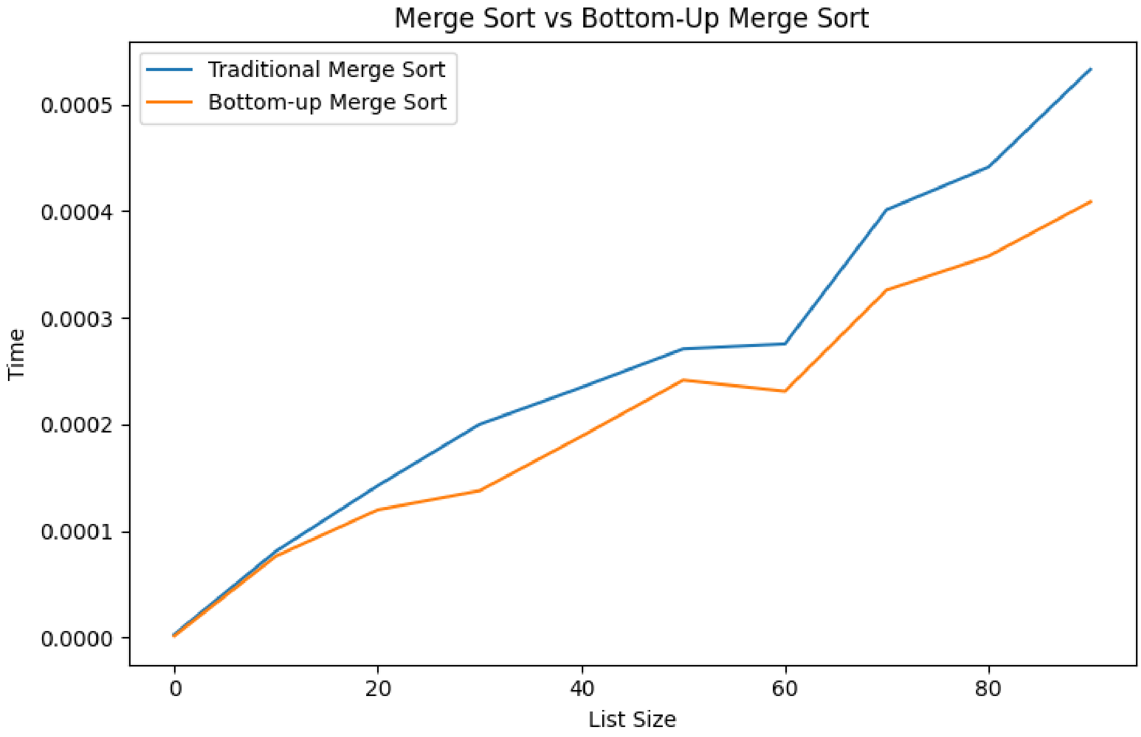

**Figure 13:** List Size (100) vs Time

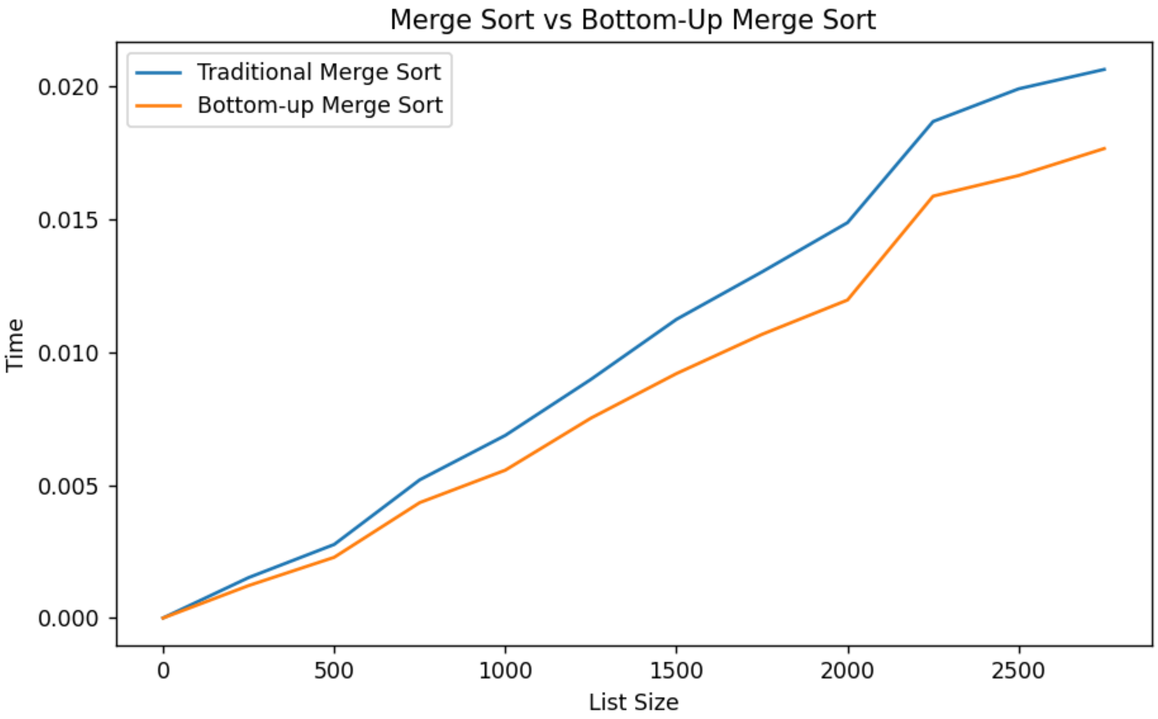

**Figure 14:** List Size (3000) vs Time

***Conclusion:***
Again the growth of both the traditional and optimized sorting algorithms is the same, there are fewer constant operations for the optimized algorithm decreasing run time.

## 4.5 Experiment 8

***Outline:***

- Comparing ”bad” sorts to ”good” sorts on small-size lists
- Two graphs with Size Length: **25, 10** | Steps: **1, 2**
- Number of Runs: **1000**

***Graphs:***

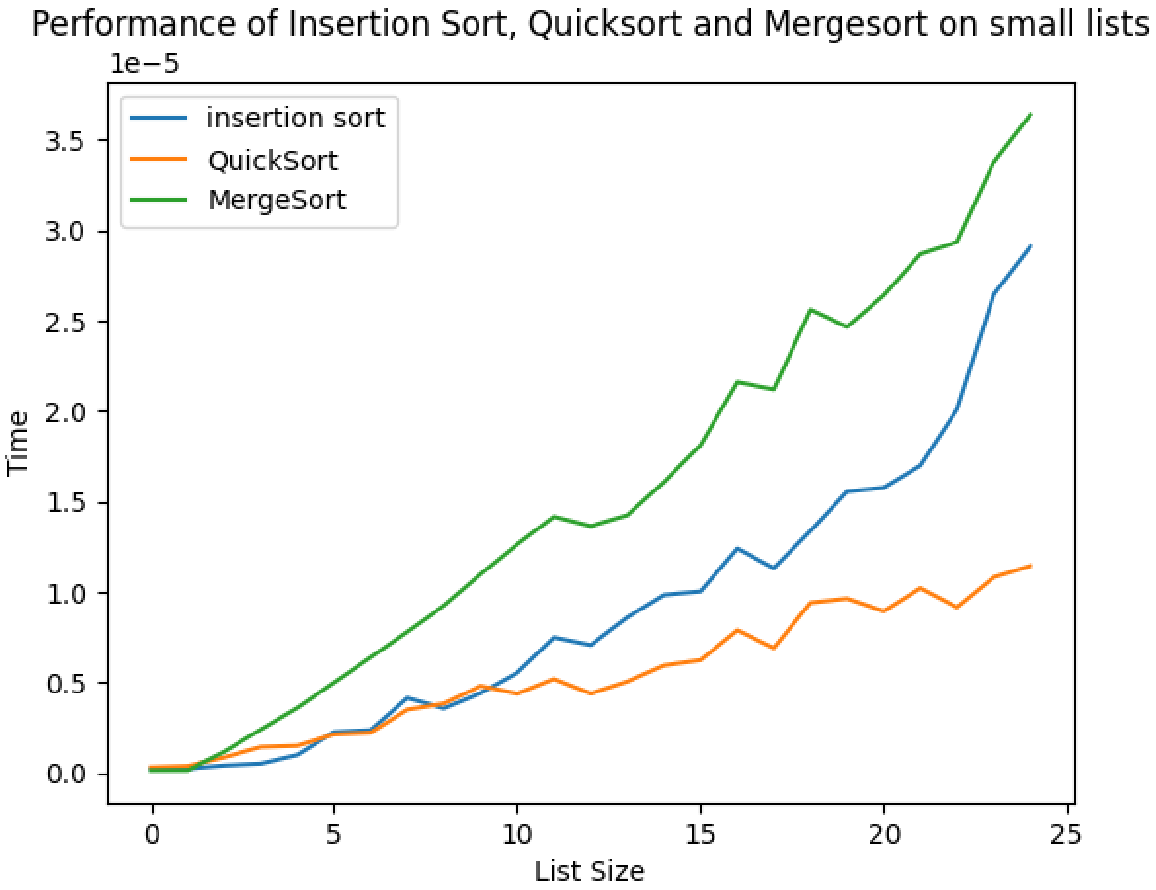

**Figure 15:** Size List (25) vs Time

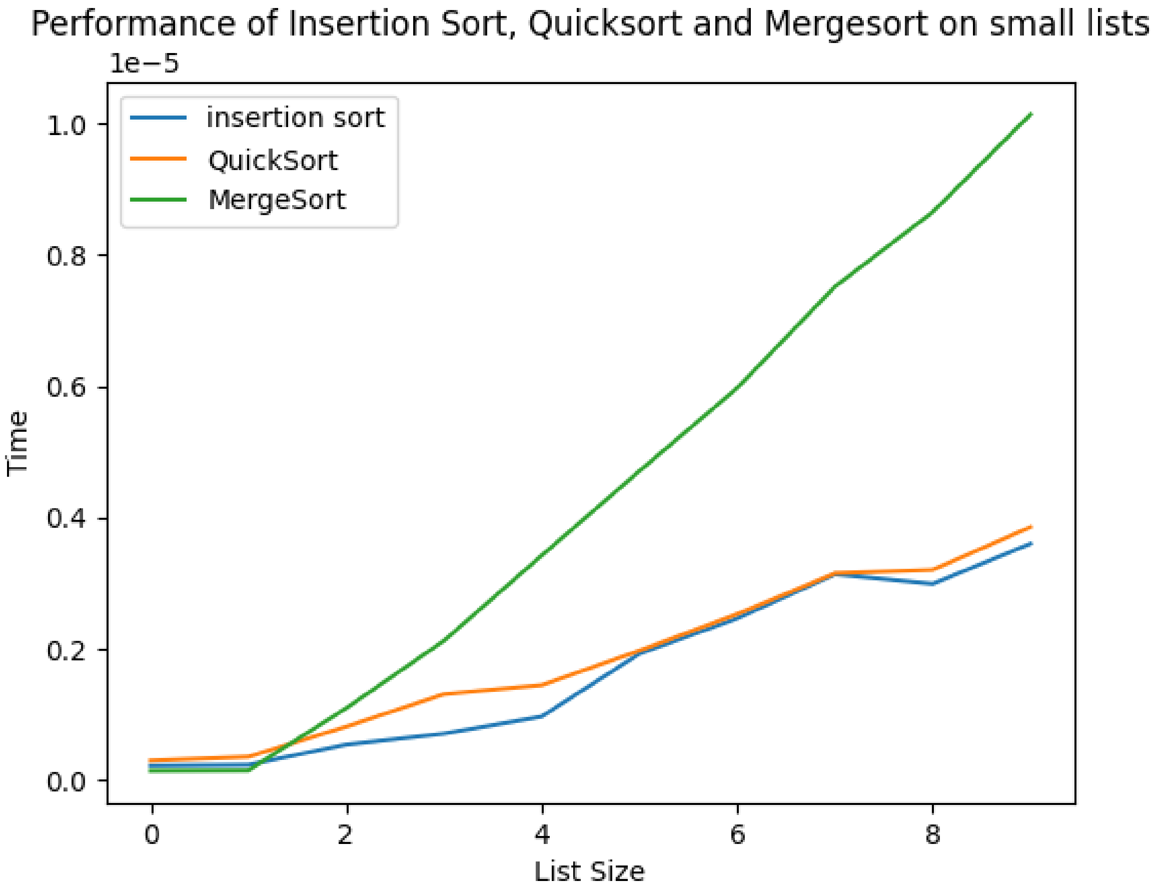

**Figure 16:** List Size (10) vs Time

***Conclusion:***
Insertion sort seems to do better than both Quick and Merge sort when the size list goes up to 10, after that point Quick sort outperforms Insertion. Insertion is better when the size list is small because the list is closer to being sorted when it is smaller, while the opposite is true for QuickSort.

## 5 Appendix

Code for all experiments can be found on this GitHub repository. There are two files one for Part 1 and one for Part 2. Each experiment corresponds to the method number, e.g. Experiment 1 corresponds to the method ”experiment1” in the file. Additional sorting algorithms are added underneath the original algorithms to maintain organization and easy navigation.