Lab 2 and 3

Nicholas, Jamie, Moein

COMPSCI 2XC3

Contents

1 Experiment 1 **3**
- 1.1 Testing Ouline . . . 3
- 1.2 Results . . . 3
- 1.3 Bad Sort Graph: Average Time x List Size . . . 3
  - 1.3.1 Bubble Sort . . . 4
  - 1.3.2 Insertion Sort . . . 4
  - 1.3.3 Selection Sort . . . 4

2 Experiment 2 **5**
- 2.1 Testing Outline . . . 5
- 2.2 Results . . . 5
- 2.3 Optimized Sorting Graphs vs. Original Sorting Graphs: Time x List Size . . . 6
  - 2.3.1 Bubble Sort . . . 6
  - 2.3.2 Insertion Sort . . . 6

3 Experiment 3 **7**
- 3.1 Testing Outline . . . 7
- 3.2 Results . . . 7
- 3.3 Bad Sort Graph: Swaps x Time . . . 7

4 **8**
- 4.1 Experiment 4 . . . 8
  - 4.1.1 Testing Outline . . . 8
  - 4.1.2 Results . . . 8
  - 4.1.3 Graphs . . . 9
- 4.2 Experiment 5 . . . 10
  - 4.2.1 Testing Outline . . . 10
  - 4.2.2 Results . . . 10
  - 4.2.3 Graphs . . . 11
- 4.3 Experiment 6 . . . 12
  - 4.3.1 Testing Outline . . . 12
  - 4.3.2 Graphs . . . 12
  - 4.3.3 Results . . . 16

4.4 Experiment 7 17
4.4.1 Testing Outline 17
4.4.2 Graphs 17
4.4.3 Results 19
4.5 Experiment 8 20
4.5.1 Testing Outline 20
4.5.2 Results 20
4.5.3 Graphs 21

Chapter 1

Experiment 1

1.1 Testing Ouline

- List Length: 1000
- Max Value: 1000
- Runs: 20 (Average per sample)

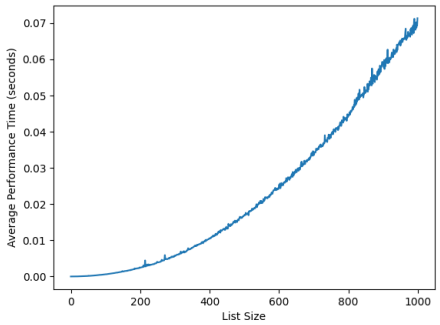
1.2 Results

Y-axis represents average time and x-axis represents list size.
Observing the graphs, we can see all three graphs show exponential growth as we approach a larger list size. Bubble sort performs the worse with the highest average time and Selection sort performing the fastest on average.

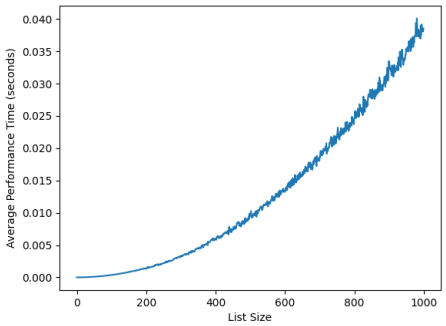
1.3 Bad Sort Graph: Average Time x List Size

Y-axis represents average time and x-axis represents list size.

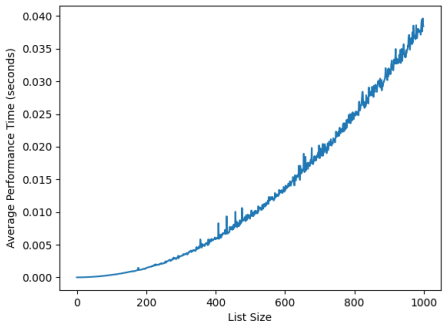
1.3.1 Bubble Sort



1.3.2 Insertion Sort



1.3.3 Selection Sort



Chapter 2

Experiment 2

2.1 Testing Outline

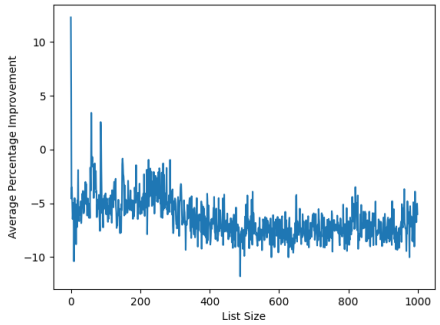
- List Length: 1000
- Max Value: 1000 (Fixed to array size)
- Runs: 20 (Average per sample)

2.2 Results

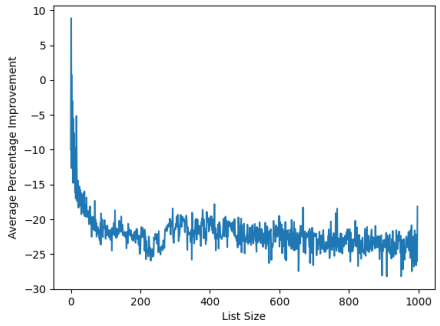
Bubble Sort: Though we improved the bubble_sort according to the format of the experiment 2, bubble_sort2 had in fact worse runtime than bubble_sort. Possibly, the increased variable assignments, conditional, and comparisons in bubble_sort2 could have increased the number of commands per iteration, increasing time to sort. In addition, with both bubble_sort methods, it will always run a set amount of iterations for a given list length. Hence, these two conditions could have caused the improvements to bubble sort to actually decrease the efficiency.
Selection Sort: With the improvements to Selection Sort, it also decreases performance time. This could be due to the decreased number iterations as we are sorting two indexes at a given time. For extremely small lists, the improved selection seems to have caused a increase in performance. Whereas, the rest of the lists had decreased performance ranging between approx. 15% - 20%.

2.3 Optimized Sorting Graphs vs. Original Sorting Graphs: Time x List Size

2.3.1 Bubble Sort



2.3.2 Insertion Sort



Chapter 3

Experiment 3

3.1 Testing Outline

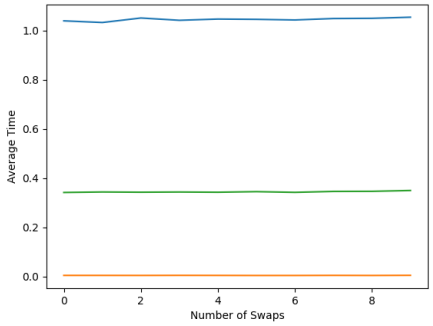
- List Length: 5000
- Max Value: 5000
- Runs: 20
- Swaps: 10

3.2 Results

Observing the graph, the average time remained around constant for all three sorting methods. Hence, it seems like the number of swaps has little to no impact on the average time of the three sorting methods.

3.3 Bad Sort Graph: Swaps x Time

Orange = Selection Sort, Green = Insertion Sort, Blue = Bubble Sort



Chapter 4

4.1 Experiment 4

4.1.1 Testing Outline

- List Length: 100 Arrays with lengths [10, 110, 210, ... 9810, 9910]
- Max List Length: 9910
- Max Value: fixed to the size of the array to reduce duplicates (Although it won't make any difference)
- Runs for each sample to take the average: 20

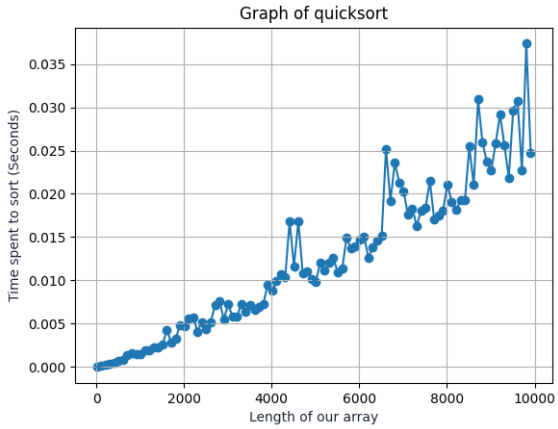
4.1.2 Results

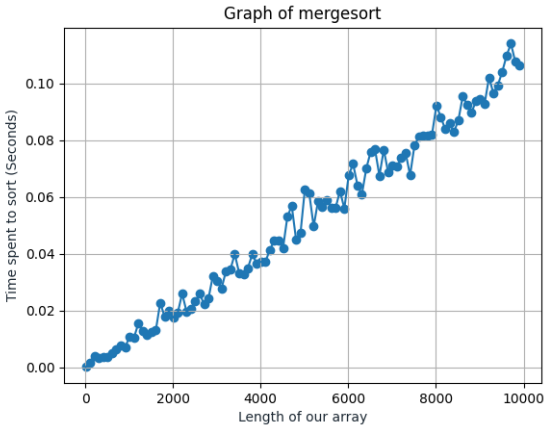
Please find the corresponding python code, in 'lab-experiments.py'

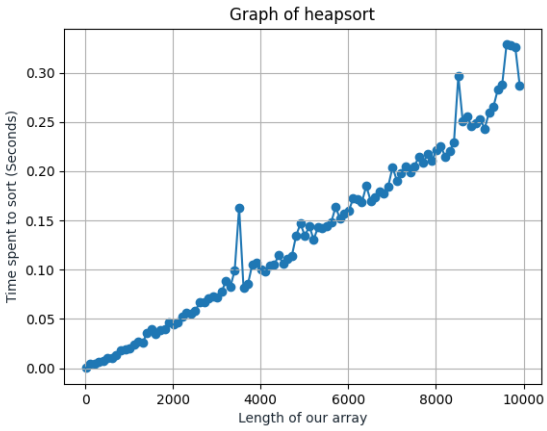
According to our experiment with the sorting algorithms (quicksort, mergesort, heapsort), the following graphs have been generated with respect to our testing outline. As we can see the other algorithms will pare in comparison with quicksort at run-time. When we are at a pint in our graph where our list has a length of above 9500, the performance of quick sort is approximately %300 faster than merge sort, and around %900 faster than heap sort. Hence we can order the algorithms with respect to their run-time as the following:

- Quick Sort
- Merge Sort
- Heap Sort

4.1.3 Graphs


Graph of quicksort
Time spent to sort (Seconds)
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
0
2000
4000
6000
8000
10000
Length of our array


Graph of mergesort
Time spent to sort (Seconds)
0.10
0.08
0.06
0.04
0.02
0.00
0
2000
4000
6000
8000
10000
Length of our array


Graph of heapsort
Time spent to sort (Seconds)
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
2000
4000
6000
8000
10000
Length of our array

4.2 Experiment 5

4.2.1 Testing Outline

- List Length: 1000
- Max Value: 1000
- Number of swaps: 50 [1, 11, 21, 31, ... 481, 491]
- Runs for each number of swaps to take the average: 10

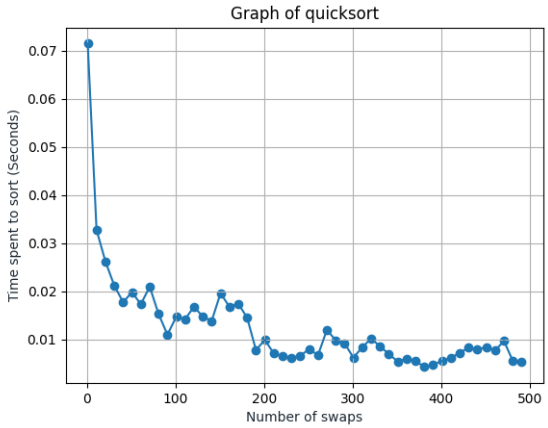
4.2.2 Results

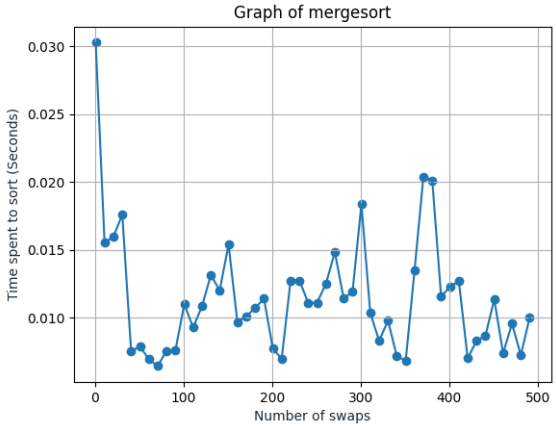
Please find the corresponding python code, in 'lab-experiments.py'

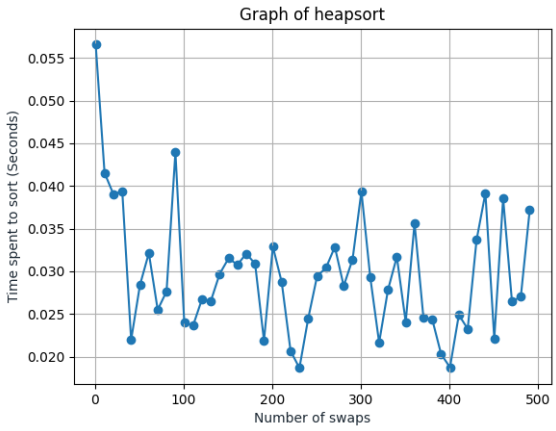
As we can see according to the graphs, keeping our testing outline in mind, the difference of quick sort compared to the other algorithms is clearly shown between 1 - 100 swaps on the x-axis. As we approach to 50 swaps both other algorithms perform better, however after that point quick sort will gradually beat heap-sort. Although merge-sort is still performing faster until we reach to the point where we have 100 swaps, and after that quick-sort will take over merge-sort.

Since we have a list of length 1000, and the difference is when we have swaps from 1 - 100, we can deduct that while our list is sorted around 1/10, or %10 sorted, then quick sort wont be as fast as others, however when our lost is %90 unsorted then we would have clear difference in the run-time with quick sort compared to other algorithms.

4.2.3 Graphs


Graph of quicksort
Time spent to sort (Seconds)
Number of swaps
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
100
200
300
400
500


Graph of mergesort
Time spent to sort (Seconds)
Number of swaps
0.030
0.025
0.020
0.015
0.010
0
100
200
300
400
500


Graph of heapsort
Time spent to sort (Seconds)
Number of swaps
0.055
0.050
0.045
0.040
0.035
0.030
0.025
0.020
0
100
200
300
400
500

4.3 Experiment 6

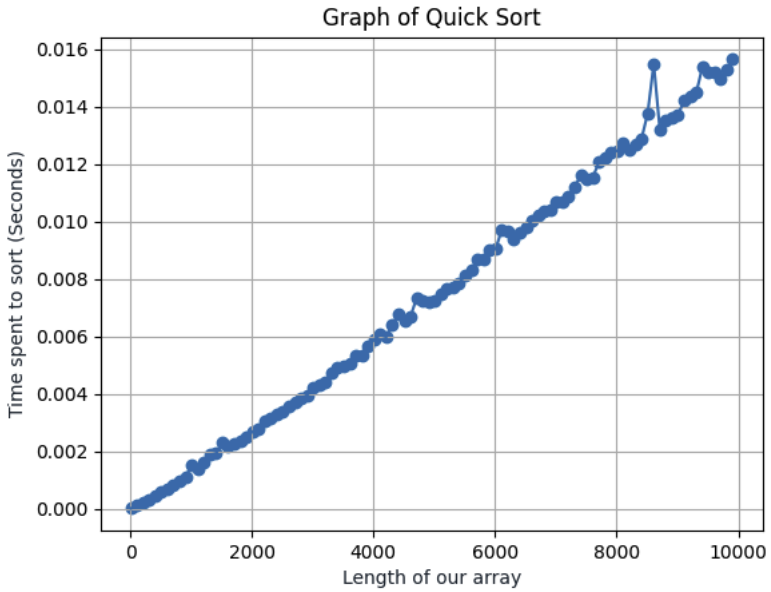
Please find the corresponding python code, in 'lab-experiments.py'

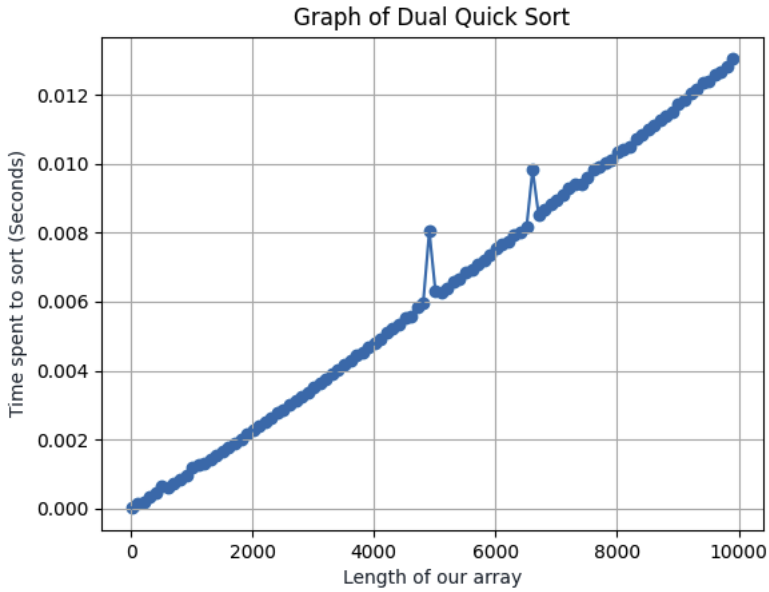
4.3.1 Testing Outline

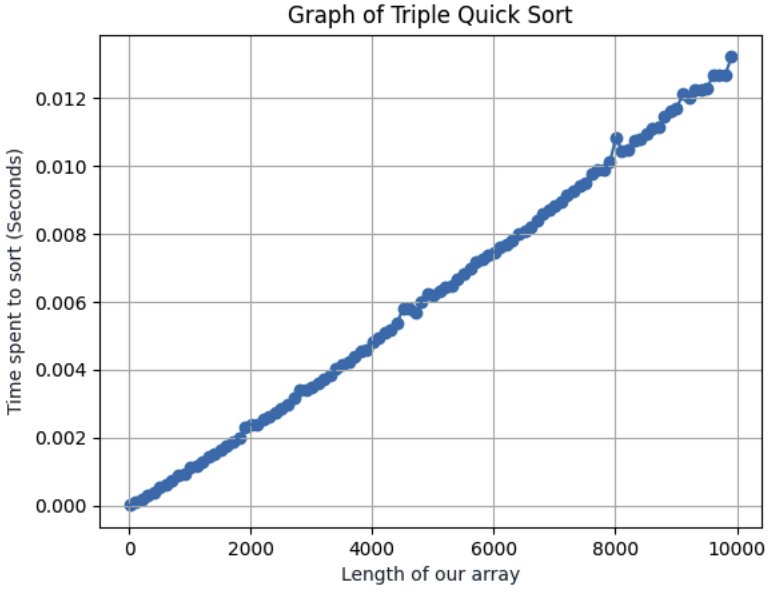
- List Length: 10000
- Max Value: 10000
- Taking the average of 20 runs for each list length

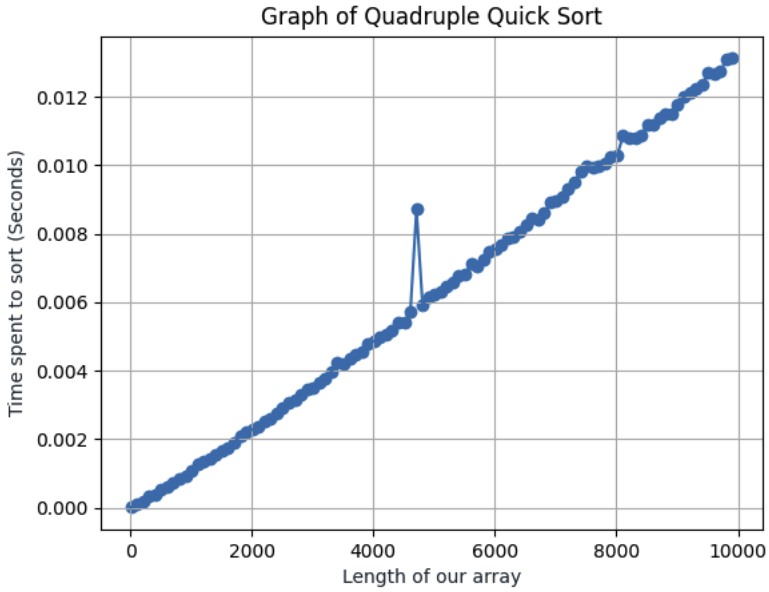
4.3.2 Graphs

Note that each quick-sort graph is named after the number of pivots it contains. For example, quick sort is the typical implementation, dual quick sort is with two pivots, triple is with three, and so on.



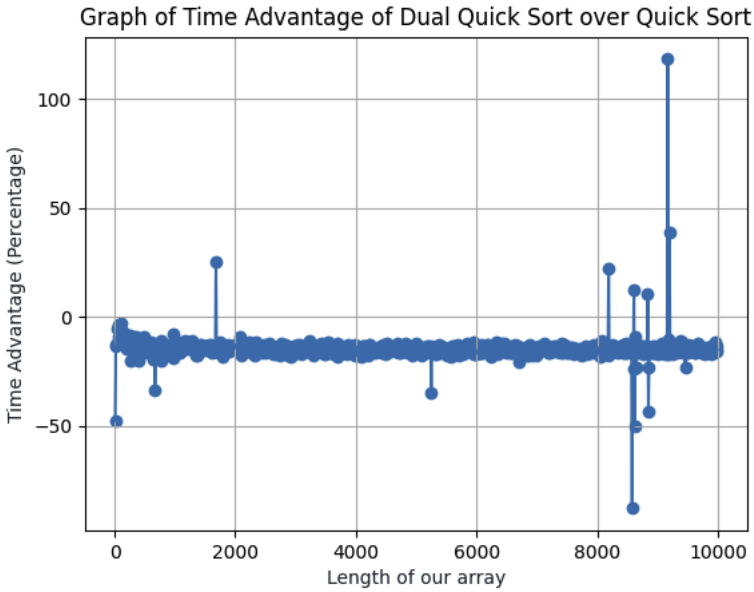

Graph of Dual Quick Sort
Time spent to sort (Seconds)
0.000
0.002
0.004
0.006
0.008
0.010
0.012
0
2000
4000
6000
8000
10000
Length of our array

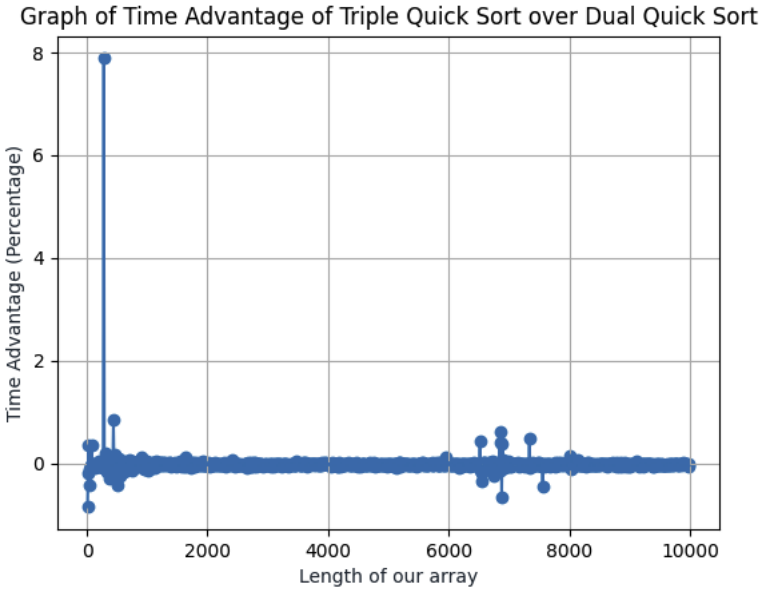

Graph of Triple Quick Sort
Time spent to sort (Seconds)
0.000
0.002
0.004
0.006
0.008
0.010
0.012
0
2000
4000
6000
8000
10000
Length of our array

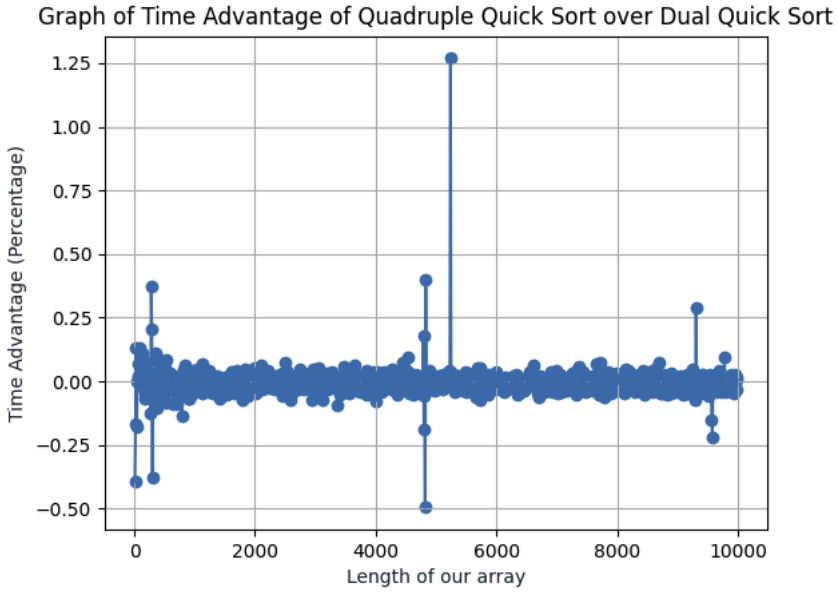


Looking at each of these graphs, we can see that the time of each of the quick sorts, no matter the amount of pivots, follows the expected $\Theta(nlogn)$ average time complexity.

Below are the graphs of the time advantages of dual quick sort compared to the typical implementation, and triple quick sort compared to dual quick sort.


Graph of Time Advantage of Dual Quick Sort over Quick Sort
Time Advantage (Percentage)
100
50
0
−50
0
2000
4000
6000
8000
10000
Length of our array


Graph of Time Advantage of Triple Quick Sort over Dual Quick Sort
Time Advantage (Percentage)
8
6
4
2
0
0
2000
4000
6000
8000
10000
Length of our array



4.3.3 Results

After analyzing these graphs, it's important to note that making use of two pivots creates a typical decrease in the amount of time taken to sort by roughly 0-20%. This makes sense in theory as if you were to draw a recursive tree for this, the depth of a two-pivot quick sort would be smaller than that of a single-pivot quick sort as you are left with shorter lists to sort. With that being said, it would increase the breadth of that recursive tree as you are making three, rather than two, recursive calls in some steps.

Additionally, it should be noted that when comparing a quick sort with three pivots and one with two pivots, the difference in time is quite small, and in some cases, there is practically no improvement at all for using more pivots.

4.4 Experiment 7

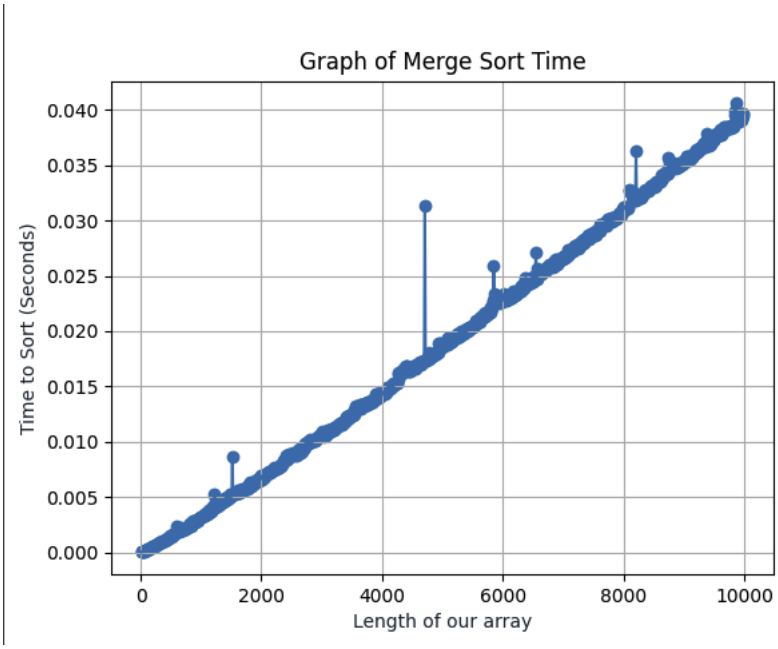
Please find the corresponding python code, in 'lab-experiments.py'

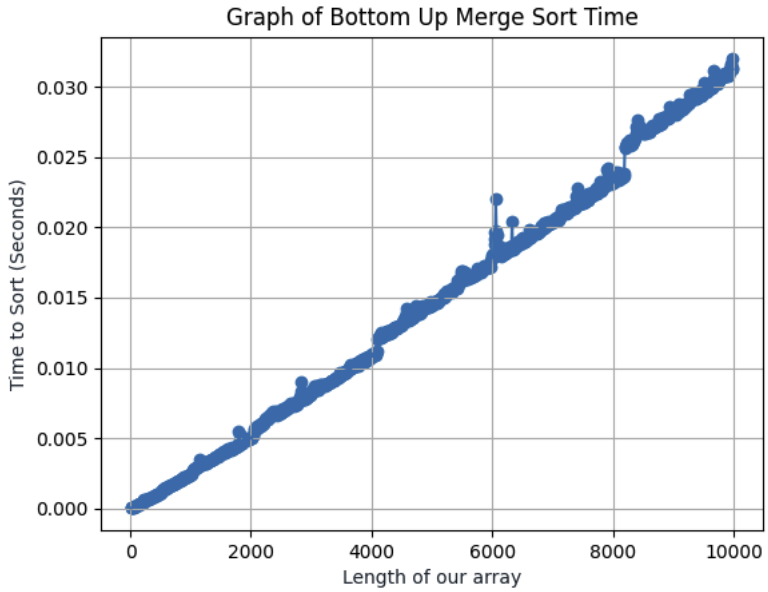
4.4.1 Testing Outline

- List Length: 10000
- Max Value: Equal to list length
- Runs: 10

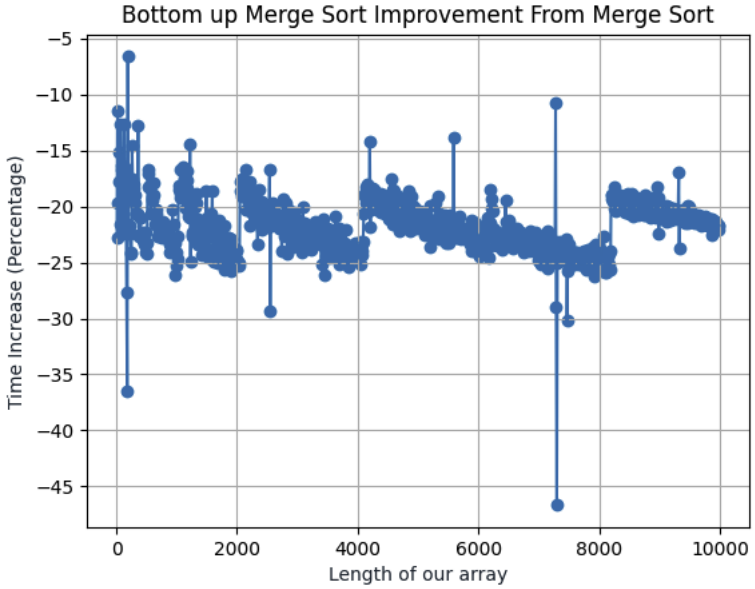
4.4.2 Graphs

Below are the graphs to display the time to sort compared to the length of a list for both the typical merge sort implementation and the bottom-up merge sort I've implemented in this experiment.





Below is a graph to display the time increase as a percentage when a list is sorted with the typical merge sort compared to the bottom-up merge sort.



4.4.3 Results

Looking at the graph of both the typical merge sort and the bottom-up merge sort I've implemented in this experiment the $O(nlogn)$ time complexity of merge sort is seen. It's especially noted how it is mainly driven by the linear factor. The bottom-up merge sort did tend to run quicker than the usual implementation of merge sort. As the comparison graph displays, there is a roughly 20% decrease in the amount of time it takes to sort the same list. However, as the lab document suggests, in theory, the iterative bottom-up approach is still somewhat recursive in nature as you are looking window by window and doubling your window size with each iteration, similar to how you would call merge sort on each half you split the list received into. As the bottom-up approach avoids recursive calls down and just needs to focus on the merges this could be a part of the reason that we see a time improvement. Additionally, it's important to note that the bottom-up approach merges in place unlike the typical implementation of merge sort as given in the 'good_sorts.py' file. This could be another reason that an improvement is seen.

4.5 Experiment 8

4.5.1 Testing Outline

- List Length: 47 arrays with lengths of [3, 4, 5, 6, ... 48, 49]
- Max Value: fixed with the array length
- Runs for each sample to take the average: 100

4.5.2 Results

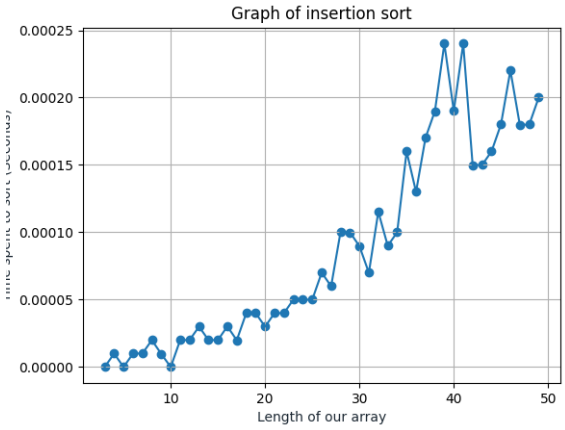
Please find the corresponding python code, in 'lab-experiments.py'

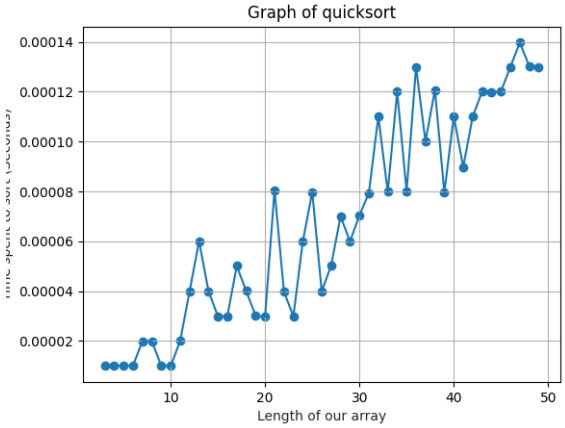
Considering the plotted graphs, we can see that for lists of length below 50, the optimized version of insertion sort could perform near, and some cases better than the other two algorithms.

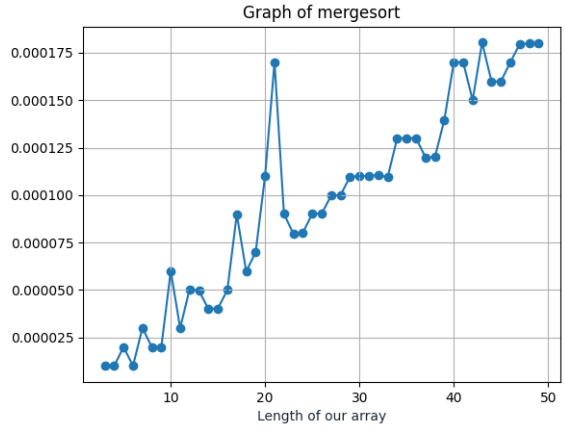
In the comparison between insertion sort and quick sort, it is clear that when we are dealing with list of length below 20, insertion sort will perform faster, and as we append to our list quick sort gradually becomes faster. Where when we are sitting at the point with 50 as our list length, quick sort will be around %60 faster than insertion sort.

As for the comparison between insertion sort and merge sort, for the list of length below 35, insertion sort will perform, nearly and faster compared to merge sort. Although this difference will become more as we append to our list and merge sort will have a better run time. Where with an array of length 50, merge sort will be approximately %35 faster than insertion sort for that length of the array.

4.5.3 Graphs


Graph of insertion sort
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
10
20
30
40
50
Length of our array


Graph of quicksort
0.00014
0.00012
0.00010
0.00008
0.00006
0.00004
0.00002
10
20
30
40
50
Length of our array


Graph of mergesort
0.000175
0.000150
0.000125
0.000100
0.000075
0.000050
0.000025
10
20
30
40
50
Length of our array