2XC3 Lab 2 + 3

Implementing, Analyzing, and Optimizing Sorting Algorithms

January 23, 2023

Andy Luo

luoa22@mcmaster.ca

Kieran Henderson

hendek12@mcmaster.ca

Corinna Lin

linc94@mcmaster.ca

Table of Contents

Table of Figures

Summary

- For reference, during the conduction of all experiments, computer processes were kept to a minimum and the same devices were used. As such, runtime variants dependent on computer processing speeds should be negligible.

Experiment 1

Using the implementations of selection sort, bubble sort, and insertion sort given in the bad_sorts.py file, we ran several experiments to compare the runtimes of these three algorithms. In the first experiment we ran, we had the following variables for each algorithm's sorting test:

- 1000 lists that were sorted by each algorithm (3000 generated lists total)
    - Where each algorithm had a newly generated list
- Lists generated from random integers ranging from 0-1000
- List lengths that increased by 1 starting from 0, up to 1000

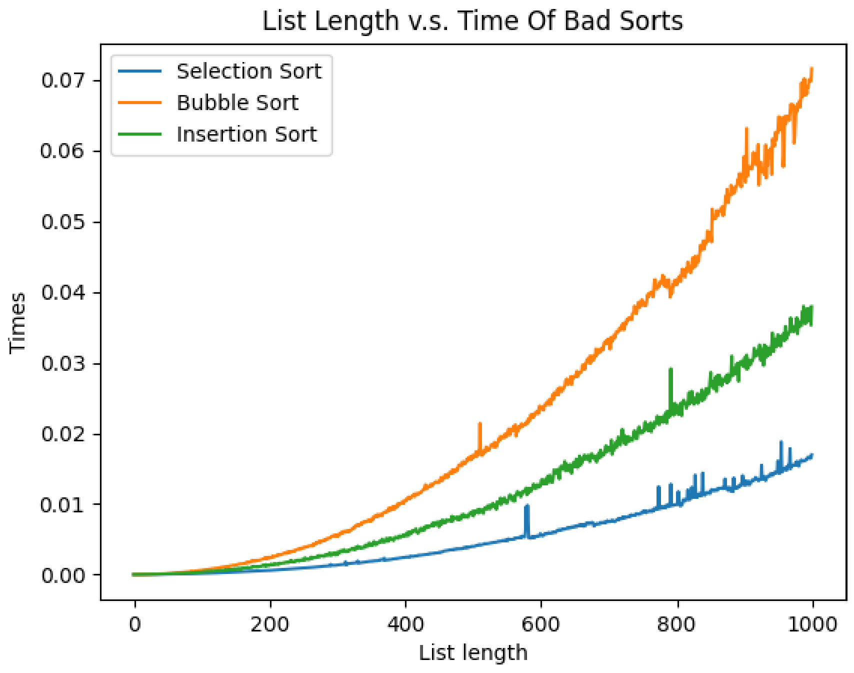

Figure 1. List Length vs Runtime for Different Lists (Bad Sorts)

Based on the results seen in Figure 1, when looking at how the length of the list affected the time taken to sort the list, all 3 sorting algorithms demonstrated a quadratic relationship. Given that all 3 algorithms also have a time complexity of O(n^2), the pattern in the figure makes sense. Though there are small spikes within each algorithm's curve, they can be attributed to combined factors such as computer processes and worst case list generation, as these spikes do not seem to occur with a pattern.

The slowest of the three algorithms is bubble sort, and the fastest is selection sort. When looking at why this may be the case, we focus on the number of swaps each algorithm makes, since swaps are an operation that have a decently high overhead. The bubble sort implementation given to us functions by taking the element at the start of the list and swapping it with the next element if it is greater. This process continues until the greatest element is at the end of the list, and then repeats again until the entire list is sorted. For our implementation, this means that one of the worst case scenarios would be a list that is sorted in descending order, because every 2 elements that are compared would need to be swapped. Having said this, the max number of swaps bubble sort has is n(n-1)/2 where n is the size of the list. When comparing this to selection sort, our implementation functions by taking a pivot point (starting at the beginning of the list) and swapping it with the smallest element within the list. This continues until the pivot point reaches the last element of the list, which means that the list is sorted. Once again, one of the worst case scenarios for this algorithm would be a list that is sorted in descending order, because every element would need to be swapped. Having said this, this means that the max number of swaps selection sort has is n-1 where n is the size of the list. For smaller lists, the difference between n(n-1)/2 and n-1 aren’t too large, which is why for lists sizes < 200, all 3 curves are somewhat similar. Once the lists keep on increasing, the multiplier in n(n-1)/2 continues to increase faster than n-1 does because it has no multiplier. This also means that the number of swaps performed in bubble sort will continue to increase much faster than the number of swaps performed in selection sort, which is why bubble sort’s times continue to grow much faster than selection sort’s times as the list length increases, as seen in the graph.

Insertion sorts functions by slowly splitting up the list into sorted and unsorted portions. As the list is iterated through, each element is inserted into its correct position within the sorted portion of the list. This continues until the list is sorted. Having said this, for our implementation, one of the worst case scenarios would be a list sorted in descending order once again because the number of comparisons made for each swap would increase as you iterate through the list. This means that the max number of swaps insertion sort has is n-1, where n is the size of the list. Though this is the same number as selection sort, insertion sort still requires more comparisons because it does not separate the list into sorted and unsorted portions. As such, it still needs to make comparisons between all elements of the list, giving it a larger overhead than selection sort.

After completing the first experiment, we realized that using different lists every time for each separate algorithm may skew the results because each algorithm may get a different ratio of best case, average case, and worst case lists. This means that one of the algorithms may have gotten lucky and gotten better lists than another, which would give it a faster performance overall, regardless of how well the algorithm itself performs. Having said this, though we didn’t expect too large of a change, we decided to eliminate the factor of luck by giving each algorithm a copy of the same list to sort, making the results more accurate to each algorithm’s general performance. We now have the following variables:

- 1000 lists that were sorted by each algorithm (1000 generated lists total)

- Lists generated from random integers ranging from 0-1000
- List lengths that increased by 1 starting from 0, up to 1000

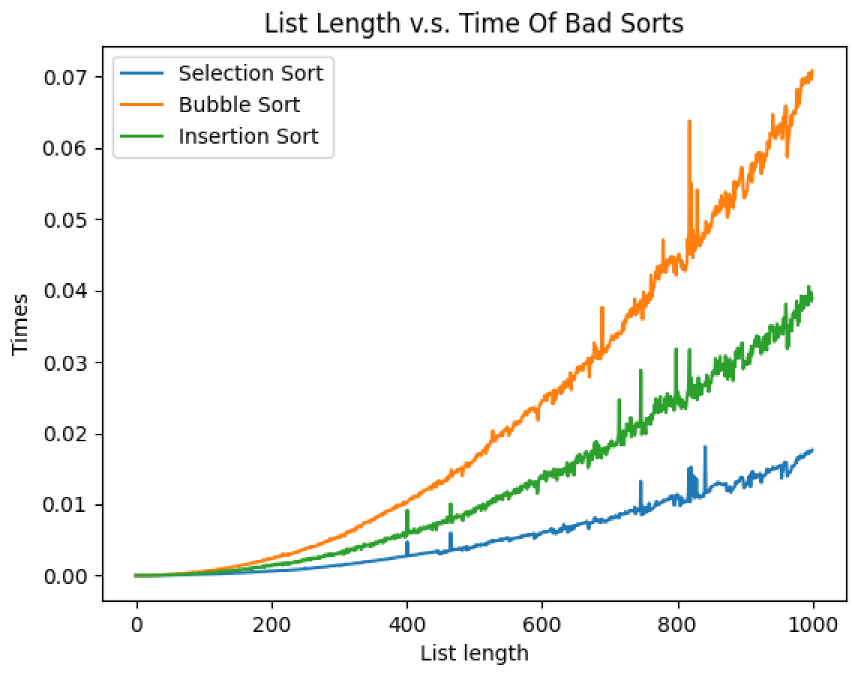

Figure 2. List Length vs Runtime for the Same Lists (Bad Sorts)

As we expected, the overall trends in Figure 2 were the same as Figure 1. If we look at some of the spikes in each curve, some of them seem to align, such as those at list lengths 400 and ~460. All 3 of our algorithms share some of the same worst case lists, such as lists sorted in descending order. Having said this, the algorithms may have gotten the same worst case scenario, causing them all to spike in runtime at the same list length.

In conclusion, all 3 sorting algorithms demonstrated a quadratic relationship when looking at how list length affected its runtime. This makes sense because all algorithms have a time complexity of $O(n^2)$. When looking at worst case scenarios, out of the three, bubble sort has the largest max number of swaps, followed by selection sort and insertion sort which are tied. However, insertion sort makes more comparisons than selection sort, giving it a higher overhead. Due to these factors, selection sort has the best run time, followed by insertion sort, then bubble sort.

Experiment 2

Starting with the implementations given to us for bubble and selection sort, after we modified each algorithm, we ran several experiments to compare how the modifications affected their runtimes. In order to compare the runtimes of our given and optimized bubble sort algorithms to see potential improvements, we had the following variables:

- Lists generated from random integers ranging from 0-1000
- List lengths that increased by 1 starting from 0, up to 1000
- 50 randomly generated lists per list length, sorted by each variation of bubble sort (50,000 generated lists total)
    - Both variations were given a copy of the generated list each time (to eliminate the possibility of one variation generating 'luckier' lists than the other)

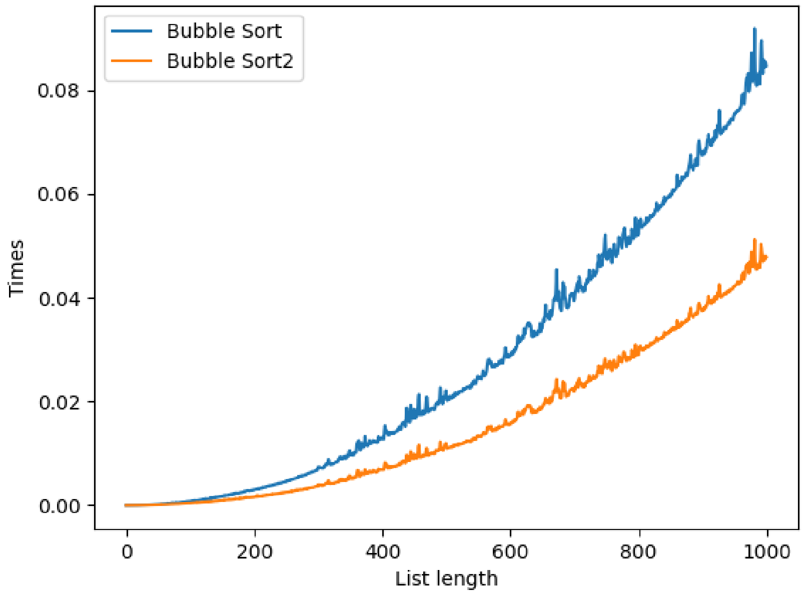

Figure 3. List Length vs Runtime (Bubble Sort Variations)

In the original bubble sort implementation, every time an element was larger than its following element, they would be swapped, which has a high overhead when looking at runtime. In lecture we saw that we could optimize insertion sort by keeping track of our value to be inserted and shifting values instead of swapping. For our bubble sort optimization, a similar approach was used. We kept track of our max element as we iterated through the list, and shifted the elements that were smaller than this max element appropriately. This continued until the max element reached the sorted portion of the list. In our implementation, the sorted and unsorted portions of the list were separated through the nested for loop condition, since we knew that one element would be added to the sorted portion after every iteration of the first for loop. As seen in Figure 3, our optimization had better runtimes than the original implementation. This can be majorly attributed to the fact that we no longer have to swap values every time element i is greater than element i + 1. Again, instead of swapping values, we track our element to be moved and simply shift the values instead which reduces the overhead involved with swapping. When looking at Figure 3, we can see spikes that occur on both curves, at the same list lengths (such as list length 690 and 990), which makes sense because both variations would've sorted the same lists, and worst case list generations should take longer to sort for both variations. It should be noted that as mentioned earlier, 50 lists were generated and sorted per list length, which gives us somewhat of an average sort time for each list length as opposed to the sort time of one generated list. This makes the overall trend of the graph more accurate because each point is averaged and does not depend as much on the random list that was generated (i.e. a generated list that gives the algorithm worst case runtime or best case runtime). Along with this, both curves demonstrate a quadratic relationship, which makes sense because they are both variations of bubble sort which has a runtime of O(n^2). When looking at lists with smaller lengths, the algorithms would not be performing many swaps in general because there are few elements, which is why the runtimes are relatively similar for both variations.

For the next experiment, we used the implementation of selection sort given to us as well as our own optimized variation. We had the following variables:

- Lists generated from random integers ranging from 0-1000
- List lengths that increased by 1 starting from 0, up to 1000
- 50 randomly generated lists per list length, sorted by each variation of selection sort (50,000 generated lists total)
    - Both variations were given a copy of the generated list each time

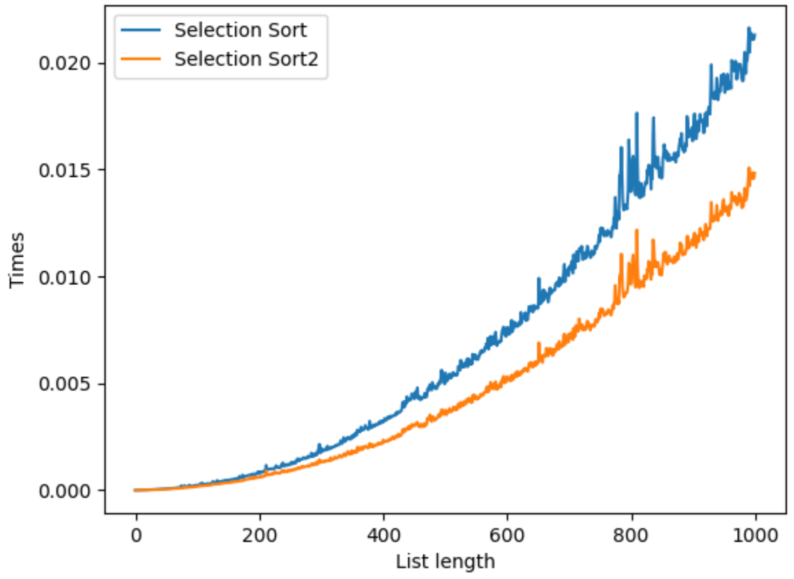

Figure 4. List Length vs Runtime (Insertion Sort Variations)

In the original selection sort implementation, we took a pivot point (starting at the beginning of the list) and swapped it with the smallest element within the list, then shifted our pivot point right once, repeating this until the list was sorted. For our variation, with each iteration, we kept track of our minimum and maximum values within our list, instead of just swapping the minimum element with our pivot point. In the original, every element from our pivot point to the end of the list would need to be compared with every iteration, in order to identify which element would need be shifted. Now that we keep track of our minimum and maximum element, we are comparing two values with every iteration, however, after swapping both elements to their respective indexes, we no longer need to compare those values (because we already know that they are our minimum and maximu values). So, in our original, with each iteration, we are shrinking the number of elements needed to be compared by 1, and with our optimization, we are shrinking the number of elements needed to be compared by 2. Essentially, instead of sorting the list from left to right, we are now sorting the list from the left and right, working towards the middle. For our optimization, the number of elements that we need to compare with each iteration will be smaller than the number of elements that need to be compared with each iteration for our original implementation. As the list length increases, the time saved will increase because the number of elements compared per iteration is larger, therefore the difference will also become larger. This is why our optimized variation's curve seems to 'split off' our original's curve as the list length increases. Once again, the spikes in the graph seem to be shared between each curve, such as the one at list length 800. This would make sense because both implementations were given the same set of lists to sort, so if the general average of lists were worst-case randomized (sorted in descending order for example), both implementations should have a longer runtime when sorting the lists.

Experiment 3

For this experiment, we ran experiments to compare how the number of swaps made to randomize the list affects the runtime of our three algorithms. We had the following variables:

- A constant list length of 5000
- 10 randomly generated lists per number of swaps, sorted by each algorithm (50,000 generated lists total)
    - The number of swaps used to randomize lists ranged from 0-400, increasing by 1 every 10 lists (because we averaged the results)
    - All 3 algorithms were given a copy of the same lists to sort
- All original implementations were used to hopefully decrease the effect of better optimized implementations

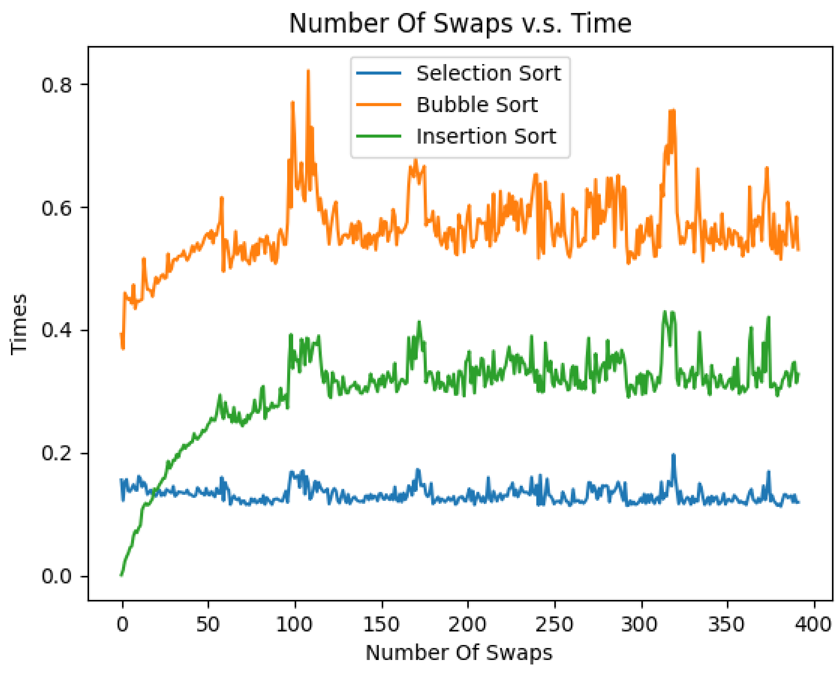

Figure 5. Number of Swaps vs Runtime

Based on our experiment, we can see that insertion sort and bubble sort seem to display a logarithmic curve, whereas selection sort seems to stay at a relatively constant runtime. It makes sense why bubble sort and insertion sort would have a logarithmic curve because once the number of swaps hits a certain threshold, all elements of the list would've been swapped randomly, so further random swaps would be like generating another completely random list. It seems like the threshold for number of swaps is around 130, once we continue swapping past that point, all 3 graphs seem to relatively level out in runtime. When we have a smaller number of swaps, since we start with a sorted list, the number of swaps will determine how sorted a list is. If no swaps are made, the list is sorted, and if 1 swap is made, the list is very close to sorted. Luck does not affect the sorted-ness of the list as much. As we increase the number of swaps, it just becomes a matter of luck, whether the lists get a worst-case, average, or best-case sort. At the beginning, when the list is already sorted, insertion sort has the best runtime because it just needs to iterate through the list comparing each element to its previous. This is followed by selection sort, which takes slightly longer because it is comparing each element with every element after it. Bubble sort takes the longest because it compares each element to its previous iteratively until the pivot point that began at the end of list is shifted to the beginning of the list. It makes sense that bubble sort and insertion sort begin to increase in runtime based on the number of swaps because the number of swaps and comparisons both algorithms make increase greatly as the sorted-ness of the list increases. For selection sort, though we did expect a small curve (because you are starting with a sorted list), we did not expect it to stay as constant as it did. This may be because the comparisons made in selection sort does not change much based on the sorted-ness, but the number of swaps does. Since the number of swaps is directly proportional to the list length, the same number of swaps would be made, because our list length stays constant.

Experiment 4

For this experiment, we ran experiments to compare how list length affects the runtimes of our three algorithms, quick sort, merge sort, and heap sort. We had the following variables:

- 50,000 lists that were sorted by each algorithm
  - Took an average of 50 lists per list length
- Lists generated from random integers ranging from 0-1000
- List lengths that increased by 1 starting from 0, up to 1000

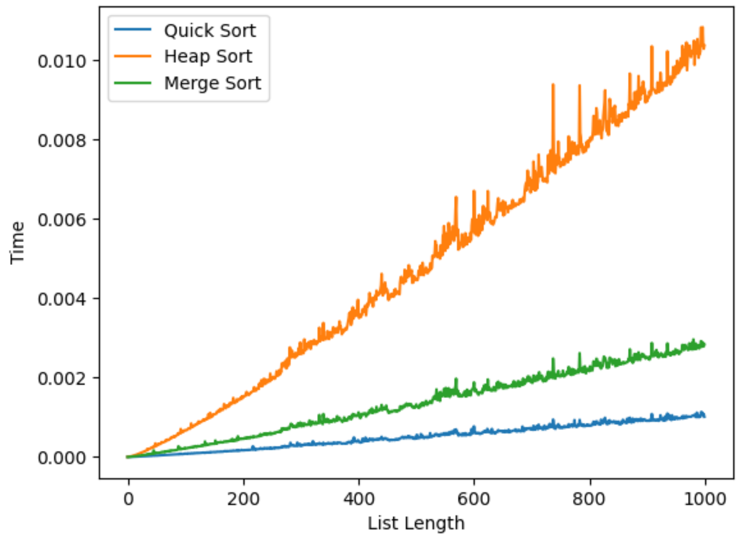

Figure 6. List Length vs Time (Good Sorts)

As the graph displays, all of the sorts display a consistent runtime, however, heap sort's runtime increases much faster than merge sort and quick sort, with quick sort being the fastest by a longshot. Heapsort is approximately 3 times slower than quicksort which makes sense because heap sort preforms a considerable amount more instructions than quicksort. Merge sort preforms less actions than heap sort, but it is still not enough to rival quicksort.

Experiment 5:

From my experiment, I have determined that about after 35 swaps, quick sort becomes more appealing than heap or merge sort. The experiment I ran involved graphing the time it took for each algorithm to sort a list of constant length 500 with values from 0-1000. For each time plotted on the graph, it would be the average of 500 sort times, each for a distinct list and each graph receiving the same list. The only variable that changes is the number of swaps applied, which increases by 1 from 0-50(non-inclusive). Figure 1 displays my results.

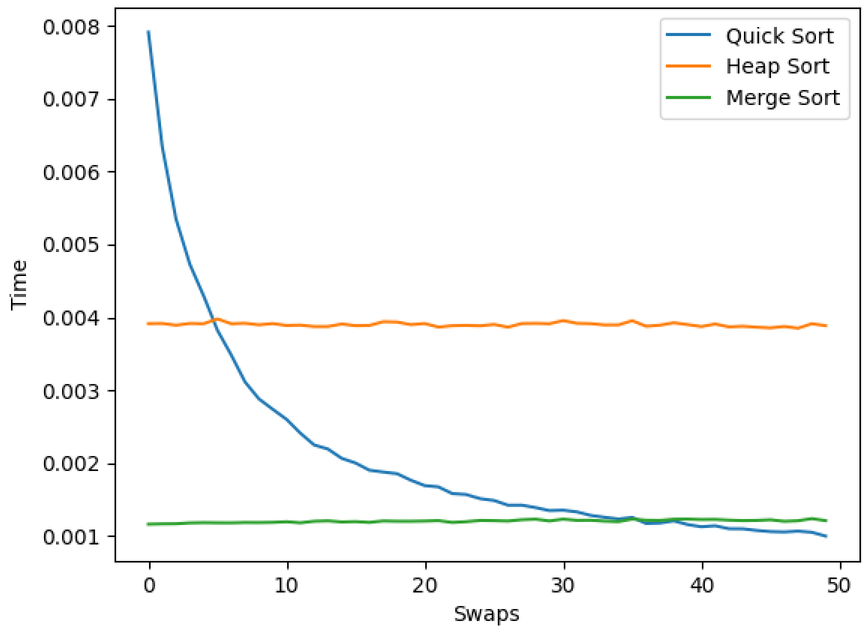

Figure 7. Number of Swaps vs Runtime (Good Sorts)

My conclusions from these results are that initially quick sort is terrible with very near sorted lists, but as the lists become increasingly unsorted, it eventually outperforms both heap and merge sort. I also noticed that heap and merge sort stay very consistent in performance regardless of the sorted-ness of a list.

Experiment 6:

From our experiments, we have determined that dual pivot quicksort is a considerable improvement over normal quicksort. As the length of the list increases, dual pivot quicksort runs an average of 54% faster than normal quicksort for list of length 0 to 2500. We ran tests on the length of the list, and how sorted the list was to begin with. We ran the experiment for lists of length 0 to 2500, taking the average of 50 sorts on each length to determine an average runtime. We also ran an experiment to determine whether the dual quicksort algorithm would also work better on input lists which are more sorted. We determined that the dual quicksort algorithm is consistently better than the quicksort algorithm with more sorted lists, however, how sorted the list is does not factor into the runtime. We tested 0 to 250 swaps on a list of length 2500 and got an average of 50 runs on each amount of swaps.

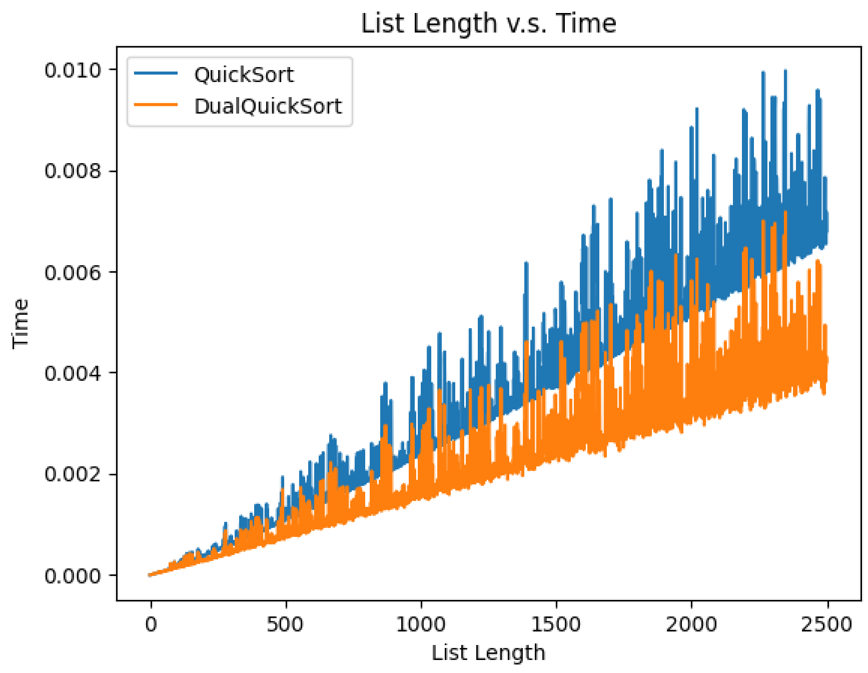

Figure 8. List Length vs Runtime (Quick Sort Variations)

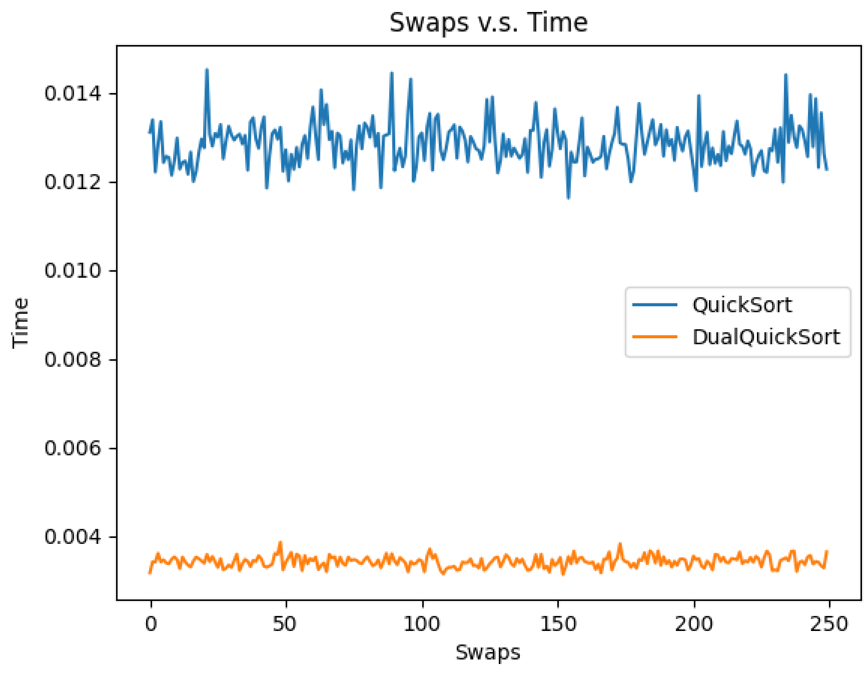

Figure 9. Number of Swaps vs Runtime (Quick Sort Variations)

We conclude that based on these results that dual quicksort is approximately 54% faster than the original quicksort algorithm on average for list length of 0 to 2500. It is more consistent and preforms better than the original quicksort algorithm on all lists, including those which are more sorted at the beginning of the algorithm. This confirmed our hypothesis that dual pivot quicksort would be faster than the original version because it makes better use of a computer's caches so that there are less cache misses occurring which increases latency which causes more stalls.

Experiment 7:

From our experiments we can determine that bottom-up merge sort is slightly better than recursive merge sort. We ran the experiment for lists of length 0 to 2500 and took an average of 50 runs for each list length. Additionally, we ran a second experiment to determine if how sorted the input array is affects the runtime of the algorithms. For this experiment, we used a fixed list length of 2500, 0 to 250 swaps on a sorted list and took an average of 50 runs for each amount quantity of swaps.

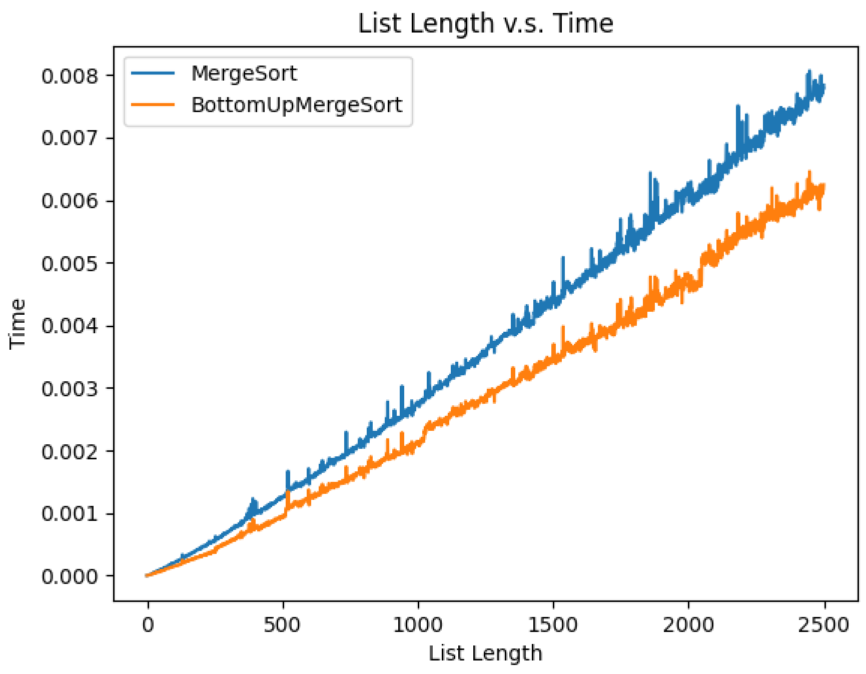

Figure 10.List Length vs Runtime (Merge Sort Variations)

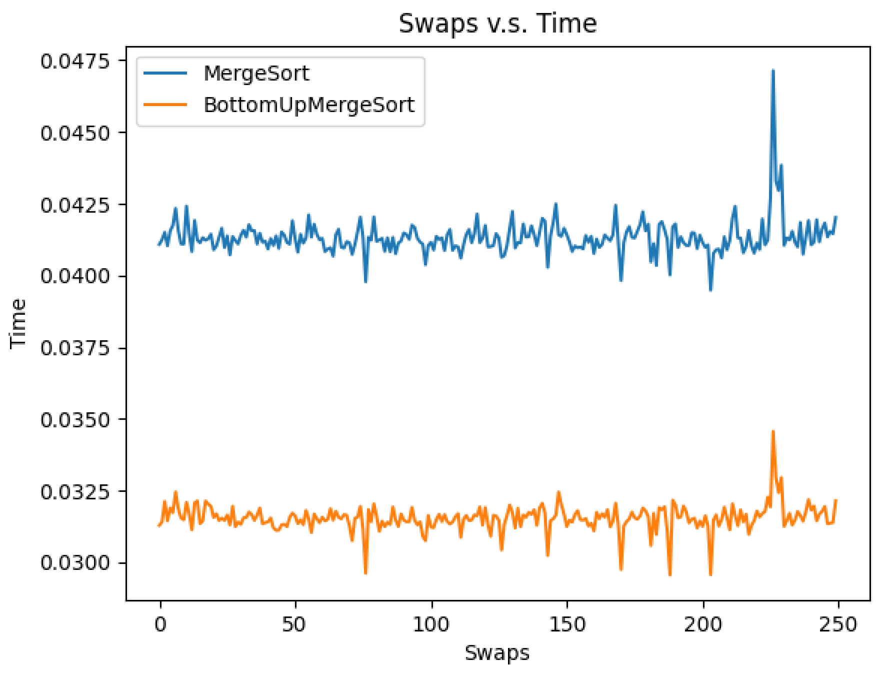

Figure 11. Number of Swaps vs Runtime (Merge Sort Variations)

From these results, we can conclude that iterative merge sort is only slightly better than traditional recursive merge sort. We believe that this is because the recursive portion of merge sort is the log(n) in its O(n*log(n)) time complexity, thus as we are optimizing the recursive implementation by using an iterative approach, thereby removing some recursive call overhead, we are only optimizing the log(n) portion of the algorithm which does not account for a huge part of the run time.

Experiment 8:

From my experiments, I have determined that insertion sort becomes “bad” at around a list length of 20 relative to Merge sort and Quick sort. The experiments I ran consisted of graphing the time it took for each algorithm to sort a randomly generated list with a list length from 0 to n (non-inclusive). Each plotted time for each algorithm would be an average of 1000 timed sorts. Each algorithm would be given the same random list where the values range from 0 to 1000 to sort and a new list will be generated after every timed sort. To determine when insertion sort becomes “bad”, I started with n = 10 (plotted times from 0-9 list length) and noted that insertion sort was outperforming both merge sort and quick sort. I continued incrementing n by 10 until I found a point at which the Insertion sort graph intersected with any other graphs. Figure 1 displays my results.

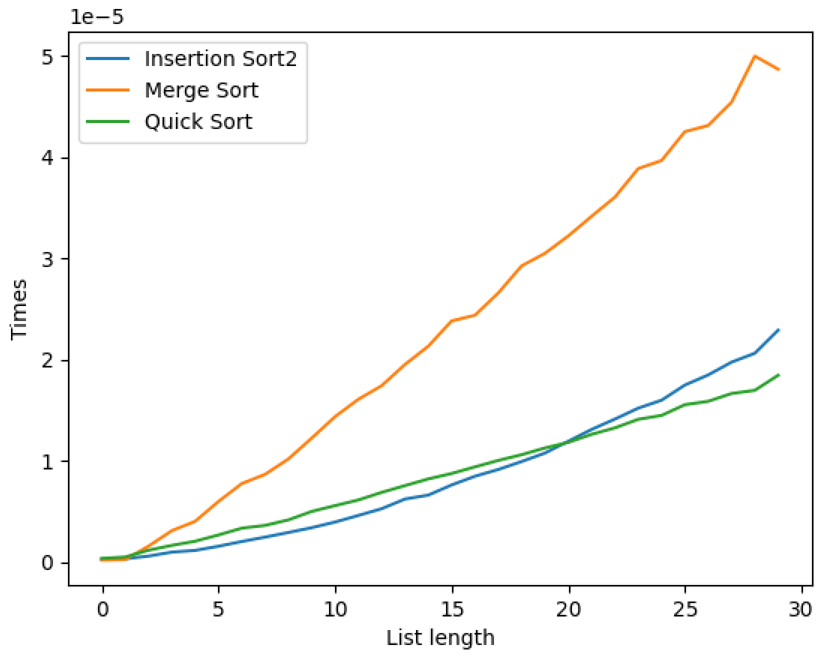

Figure 11. List Length vs Runtime (Bad and Good Sorts)

My conclusions from these results are that insertion sort can outperform merge sort and quick sort but only while the list length is notably small. This possibility starts to intuitively make sense to me since big O notation only determines the growth of the time it takes for an algorithm to sort over an increasing n.

These results are important and practical since it can help determine which sorting algorithm to use based on the size of the list to be sorted. From these results I have determined a scenario in which insertion sort can most definitely be used over merge or quick sort. For example, if the list is small (e.g. less than 20) then we now know to choose insertion sort over merge or quick sort. This can be helpful since insertion sort is significantly less complicated and easier to implement than the latter two. In understanding when to use either one, it is possible to make a “hybrid” sort of insertion and merge sort. This “hybrid” algorithm would use insertion sort for small lists and switch to Merge sort for larger lists. Through this method, the ”hybrid” benefits from the most optimal performance of both insertion and merge sort.

Appendix