Course: Data Structures, Algorithms, and Their Applications – CPSC 319
Assignment 1

Instructor: Leonard Manzara
T.A.: Xi Wang
Tutorial Section: T04

Student Name: Danielle Jourdain
UCID: 30114566
Date Submitted: February 3, 2022

Table of Contents

Table of Figures and Tables

1 - Experimental Method

To collect data for this experiment, I first had to create the program to run the algorithms. Before running my official tests, I did some preliminary testing to make sure the values I was getting for the time of the algorithms made sense. These results were not recorded since they were only for my own reference.

After these preliminary tests, I moved on to collecting official data. To do this, I ran my program 72 times. There were 3 categories, algorithm, order, and size. There were 4 options for algorithm – Selection, Insertion, Merge, and Quick sort – and the program was run 18 times for each algorithm. There were 3 choices for order – ascending, descending and random – and the program was run once for each order at each size. Finally, there were 6 choices for size. The choices were 10, 100, 1000, 10,000, 100,000, and 1,000,000. The program was run once for each unique combination of size, order, and algorithm.

2 - Data Collected

Table 1: Data Collected from Algorithms

Algorithm	Size / Order	10	100	1,000	10,000	100,000	1,000,000
Selection	Ascending	2.73E-06	8.14E-05	2.59E-03	2.46E-02	1.97E+00	1.06E+02
	Descending	3.28E-06	8.87E-05	2.61E-03	6.18E-02	5.78E+00	6.45E+02
	Random	3.02E-06	8.86E-05	2.61E-03	5.49E-02	5.15E+00	5.18E+02
Insertion	Ascending	1.84E-06	4.20E-06	2.71E-05	2.54E-04	1.76E-03	5.33E-03
	Descending	2.90E-06	9.57E-05	2.88E-03	2.25E-02	1.66E+00	1.62E+02
	Random	2.64E-06	5.12E-05	2.09E-03	1.87E-02	9.48E-01	9.06E+01
Merge	Ascending	7.61E-06	7.14E-05	1.72E-03	8.08E-02	2.19E+00	1.80E+02
	Descending	1.02E-05	9.94E-05	1.96E-03	8.68E-02	2.43E+00	2.09E+02
	Random	7.79E-06	7.28E-05	1.86E-03	8.36E-02	2.21E+00	2.04E+02
Quick	Ascending	5.26E-06	2.07E-05	1.89E-04	9.54E-04	2.24E-02	4.19E-02
	Descending	5.72E-06	2.60E-05	2.11E-04	1.02E-03	2.26E-02	5.66E-02
	Random	1.84E-06	2.88E-05	3.53E-04	1.49E-03	3.55E-02	1.05E-01

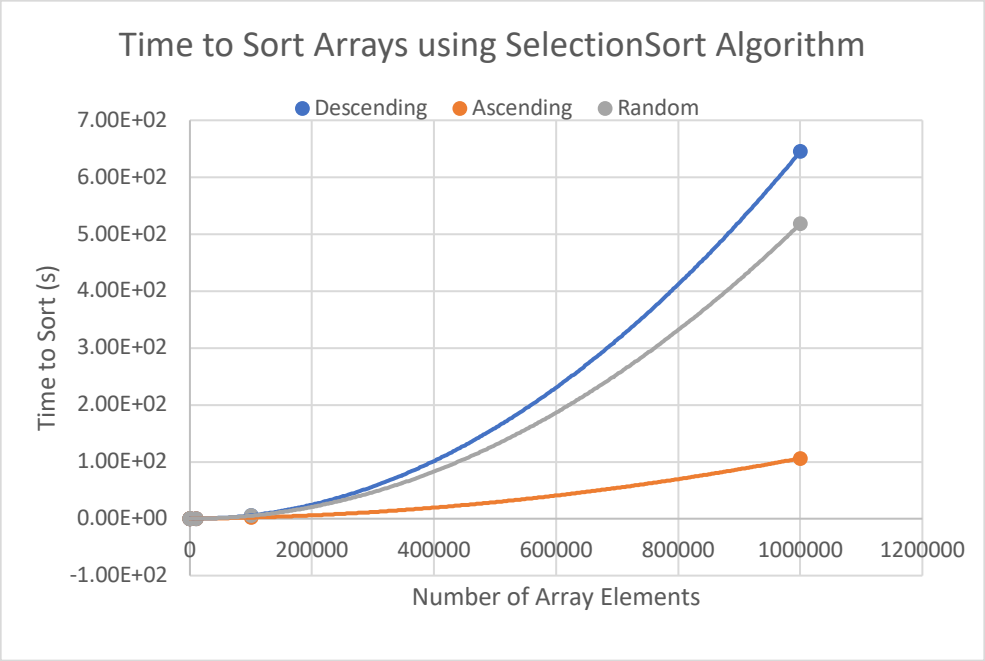

Figure 1: Graph Showing Number of Elements vs Time to Sort using Selection Sort

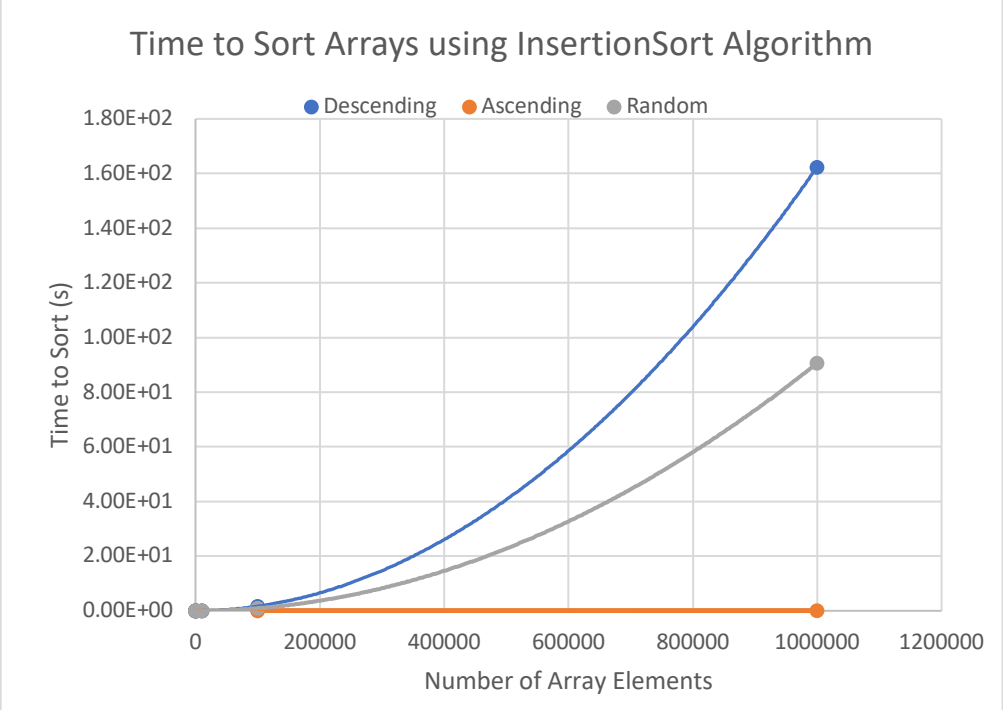

Figure 2: Graph Showing Number of Elements vs Time to Sort using Insertion Sort

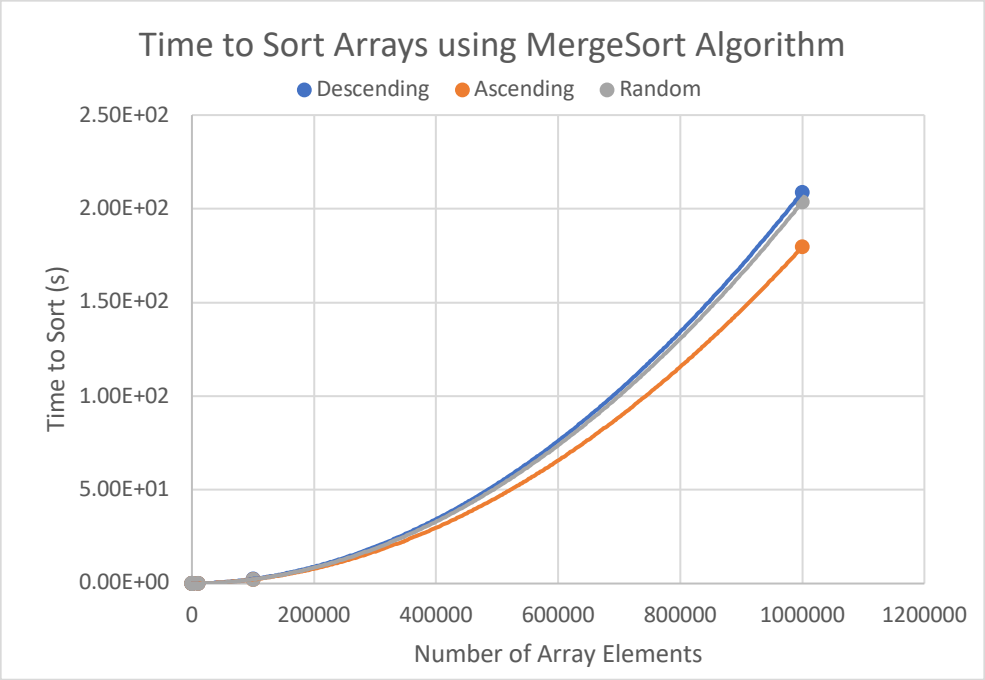

Figure 3: Graph Showing Number of Elements vs Time to Sort using Merge Sort

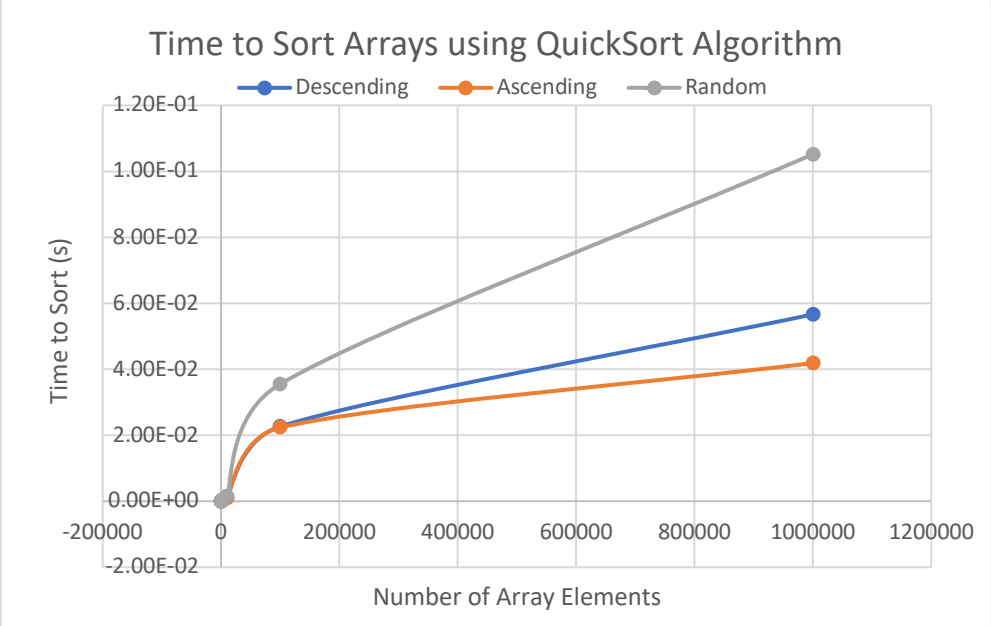

Figure 4: Graph Showing Number of Elements vs Time to Sort using Quick Sort

3 - Data Analysis

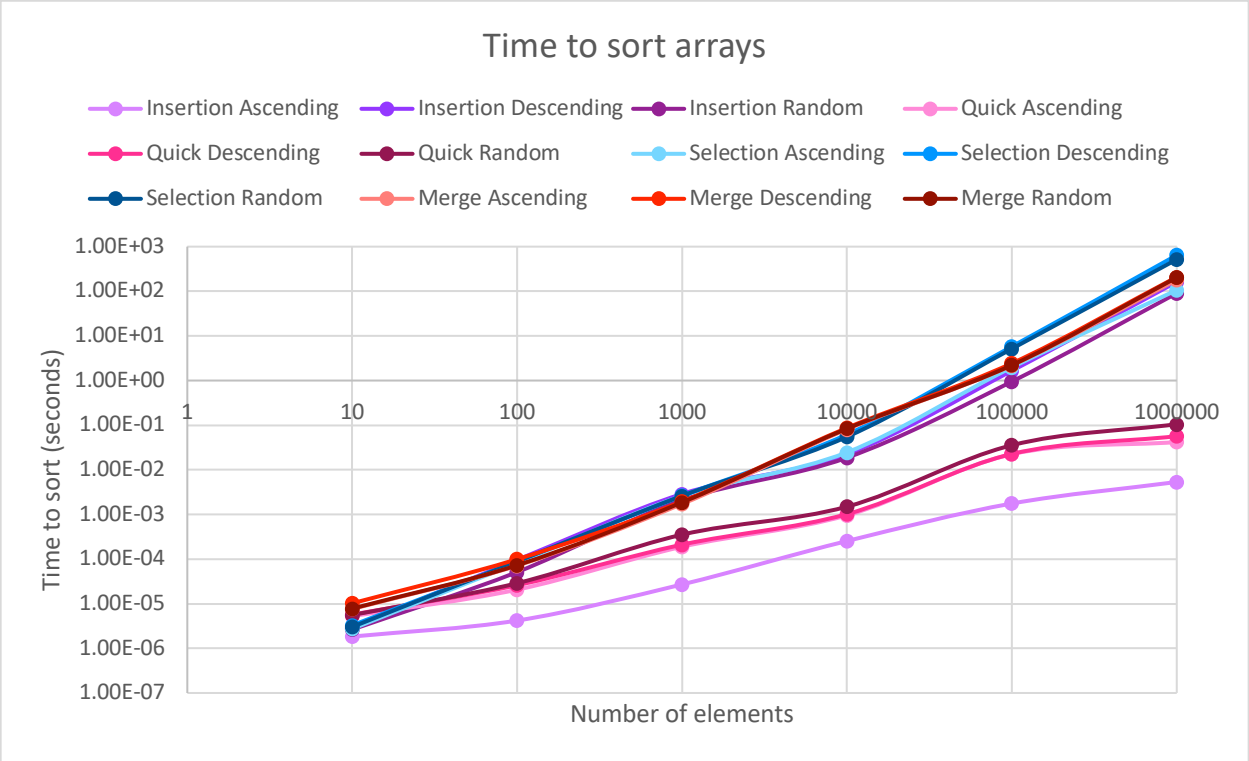

Figure 5: Graph showing all algorithms and times on a logarithmic axis

From the data in Section 2, we can see in all cases, data that is already in ascending order is much faster to sort. In most cases, descending order data takes the longest and random order data falls somewhere in between the others. However, for the Quick Sort algorithm, random order data takes the longest to sort. Figure 5 shows a graph with all data on one set of axes. This graph shows that overall, the Quick Sort algorithm is the fastest, although Insertion Sort is even faster, but only when the data is already in ascending order. The upper portion of this graph is too convoluted to get many details from, but it does show that overall Selection Sort's efficiency decreases as the number of elements goes up, and that Merge Sort's efficiency is consistent.

3.1 – Selection Sort

The data for Selection Sort shows that the trendline is quadratic for all 3 orders (Figure 1). There is a large difference between the times for ascending and descending data. The random data tends to be closer to descending data than the ascending data. This shows that if data is not pre-sorted the algorithm is much more inefficient.

3.2 – Insertion Sort

For Insertion Sort, there is a quadratic trendline for descending and random data, but a linear trendline for ascending data (Figure 2). This shows off the efficiency of this algorithm if data is already sorted. However, for unsorted data, Insertion Sort takes much longer than it does for sorted data. The sorting time for random data falls in the middle of ascending and descending times, but it is a bit closer to descending data.

3.3 – Merge Sort

For Merge Sort, all three orders of data what appears to be an n log(n) trendline. The curves in Figure 3 are more subtle than the quadratic trendlines of Selection and Insertion Sort, which leads me to believe that these are not quadratic lines. There is a much tighter margin between ascending and descending data than other algorithms, showing that the order of data does not have as much of an effect as other algorithms. Much like Selection and Insertion Sort, the ascending data for Merge Sort has the fastest sorting time and the descending data has the slowest time. For Merge Sort, the random data sorting time is much closer to the descending data than the ascending data, but it still falls in between the two.

3.4 – Quick Sort

The graph in Figure 4 shows what appears to be a logarithmic trendline. Quick Sort shows a different pattern for ascending, descending, and random sorting times than all the other algorithms. For Quick Sort, ascending and descending data have similar times, with ascending data being sorted slightly faster. However, random data takes much longer to sort than the other two orders. This difference makes sense since the pivot/bound chosen for my implementation of Quick Sort is the centre element of the array. For ascending and descending data this is the ideal pivot element. Choosing a pivot for Quick Sort is difficult since choosing the value closest to the median is ideal. However, calculating the median of the data would be a costly operation, so another method must be chosen. Choosing the centre element of the array is perfect for ascending and descending data. This is an issue with random data however, since the centre element of the array could be any value, so the sorting will not be as optimized.

4 - Complexity Analysis

4.1 – Assumptions

To complete this complexity analysis, some assumptions were made. Firstly, all declarations, initializations, arithmetic operations, value fetching and *if* statements were counted as a single operation. The second assumption is that a single loop (*for*, *while*, or *do while*) has a complexity of O(n). To complete the analysis, the product and summation rules will be applied to the code.

4.2 – Selection Sort

```
Line 1    procedure selectionSort(int[] arr)
Line 2        for i = 0 to length(arr)
Line 3            minElement = arr[0]
Line 4
Line 5            for each unsorted element
Line 6                if element < minElement
Line 7                    minElement = element
Line 8
Line 9            swap (minElement, first unsorted position)
Line 10   end selectionSort
```

Figure 6: Selection Sort pseudocode

4.2.1 – Best Case

The best case for this algorithm is when the data is pre-sorted, since no elements need to be swapped. There are two *for* loops (Lines 2 and 5) with one nested inside the other with an *if* statement (Line 6) inside the innermost loop. By applying the product rule, we get a complexity of $O(n^2)$ since each individual *for* loop has a complexity of $O(n)$.

4.2.2 – Worst Case

The worst case for this algorithm is when the data is in descending order. When this is the case, the algorithm must swap elements the maximum number of times which is *n-1* times. As with the best case, there is a set of two nested *for* loops with an if statement in the inner loop. This gives the same complexity of $O(n^2)$.

4.3 – Insertion Sort

```
Line 1    procedure insertionSort(int[] arr)
Line 2        for i = 0 to length(arr)
Line 3            current = arr[i]
Line 4
Line 5            j = i
Line 6            for j = i to 0 && arr[j – 1] > current
Line 7                arr[j] = arr[j – 1]
Line 8
Line 9            arr[j] = temp
Line 10   end insertionSort
```

Figure 7: Insertion Sort pseudocode

4.3.1 – Best Case

The best case for this algorithm is when the data is pre-sorted, since no elements need to be swapped. There are two *for* loops (Lines 2 and 6). However, when data is in ascending order, the inner loop will not be executed because of the second condition. This gives a complexity of $O(n)$.

4.3.2 – Worst Case

The worst case for this algorithm is when the data is in descending order. When this is the case, the algorithm must shift elements the maximum number of times which is *n* times. As with the best case, there is a set of two nested *for* loops, but in the worst case, the inner for loop will be executed on every iteration of the outer loop. These nested loops give a complexity analysis of $O(n^2)$.

4.4 – Merge Sort

```
Line 1    procedure merge(int[] arr, int leftPos, int rightPos, int rightEnd):
Line 2        int[arr.length] temp
Line 3        int leftEnd = rightPos – 1
Line 4        tempPos = leftPos
Line 5
Line 6        while(leftPos <= leftEnd && rightPos <= rightEnd)
Line 7            if(arr[leftPos] <= arr[rightPos])
Line 8                temp[tempPos++] = arr[leftPos++]
Line 9            else
Line 10               temp[tempPos++] = arr[rightPos++]
Line 11
Line 12       while(leftPos <= leftEnd)
Line 13           temp[tempPos++] = arr[leftPos++]
Line 14
Line 15       while(rightPos <= rightEnd)
Line 16           temp[tempPos++] = arr[rightPos++]
Line 17
Line 18       for i = 0 to arr.length
Line 19           arr[i] = temp[i]
Line 20   end merge
Line 21
Line 22   procedure mergeSort(int[] arr, int left, int right)
Line 23       int centre = (left + right) / 2
Line 24       mergeSort(arr, left, centre)
Line 25       mergeSort(arr, centre + 1, right)
Line 26       merge(arr, left, centre + 1, right)
Line 27   end mergeSort
```

Figure 8: Merge Sort pseudocode

4.4.1 – Best Case

The best case for this algorithm is when the data is in either ascending or descending order. If this is the case, the algorithm will divide the data into two sub-arrays of equal length. The length of each of these sub-arrays will be n / 2. Then the algorithm will divide these sub-arrays into new sub-arrays each with a length of n / 4. This method will continue until the sub-array length is 1. This gives a complexity of $O(\log n)$ for the mergeSort method. After this, the merge method must be considered. Since this iterates over the entire array it has a complexity of $O(n)$. It is called every time the array is divided into new sub-arrays. Using the product rule, the overall complexity for the best case is $O(n \log n)$.

4.4.2 – Worst Case

The worst case for this algorithm is when the data is in a random order, giving the maximum number of comparisons. However, even in the worst case, the array is divided into sub-arrays half of its size giving $O(\log n)$ for the mergeSort method. Again, like the best case, the merge method has a complexity of $O(n)$, and is performed once every time the array is divided. This gives an overall complexity of $O(n \log n)$.

4.5 – Quick Sort

```
Line 1    procedure quickSort(int[] arr, int start, int end)
Line 2        int lower = start + 1
Line 3        int upper = end
Line 4
Line 5        swap(arr, start, (start + end) / 2)
Line 6        int bound = arr[start]
Line 7
Line 8        while(lower <= upper)
Line 9            while(arr[lower++] < bound)
Line 10           while(arr[upper--] > bound)
Line 11
Line 12           if(lower < upper)
Line 13               swap(arr, lower++, upper--)
Line 14           else
Line 15               lower++
Line 16
Line 17       swap(arr, upper, start)
Line 18       if(start < upper – 1)
Line 19           quickSort(arr, start, upper – 1)
Line 20       if((upper + 1) < endx)
Line 21           quickSort(arr, upper + 1, start
Line 22   end quickSort
Line 23
Line 24   procedure swap(int[] arr, int i, int j)
Line 25       int temp = array[i]
Line 26       array[i] = array[j]
Line 27       array[j] = temp
Line 28   end swap
```

Figure 9: Quick Sort pseudocode

4.5.1 – Best Case

The best case for this algorithm is when the data is in either ascending or descending order. This means that with my implementation of Quick Sort the ideal pivot would be chosen. If this is the case, the algorithm will divide the data into two sub-arrays of equal length. The length of each of these sub-arrays will be n / 2. Then the algorithm will divide these sub-arrays into new sub-arrays each with a length of n / 4. This method will continue until the sub-array length is 1. This gives a complexity of $O(\log n)$ for the quickSort method. Once in

4.5.2 – Worst Case

The worst case for this algorithm is when the data is in a random order, giving the maximum number of comparisons. However, even in the worst case, the array is divided into sub-arrays half of its size giving $O(\log n)$ for the mergeSort method. Again, like the best case, the merge method has a complexity of $O(n)$, and is performed once every time the array is divided. This gives an overall complexity of $O(n \log n)$.

5 - Interpretation

The data and analysis show that each algorithm has unique behaviour. All four of the algorithms tested have strengths and weaknesses in different scenarios. The simpler sequential search algorithms performed best with small amounts of data (no more than 100 elements per array). This lines up with the complexity analysis especially for Insertion Sort, since its best case give O(n) complexity. This is obviously more efficient than some of the binary search algorithms which have an O(n log(n)) complexity even in their best cases.

On the other hand, for large groups of data, the sequential search algorithms are not good at sorting quickly. In this case the binary search algorithms performed much better. This makes sense when compared to the complexity analysis for each of the algorithms. For both Selection and Insertion Sort, in average and worst cases, the complexity is $O(n^2)$. Comparing that to the O(n log(n)) complexity of Merge and Quick sort shows that these algorithms will be much better for large amounts of data.

Size of data is not the only important factor when choosing a sorting algorithm. As seen from the graphs in Section 2 of the report, order of the data makes a large difference in sorting times. If data is pre-sorted, most of the algorithms are much faster at sorting this data. When the data is in reverse order, the algorithms usually take significantly more time to sort through all the data. Lastly, the random order data generally the sorting time falls somewhere between the best (ascending data) and worst (descending data) cases.

The one exception to this trend was the Quick Sort algorithm. This makes sense when analyzing the algorithm. Since the Quick Sort algorithm requires that a pivot/boundary value is required, this value can change the effectiveness of this method. For this experiment, the pivot was chosen as the middle index of the array. In the cases where data was ascending or descending, this made the pivot value the middle value of the array which is the ideal value for a pivot. However, when using random data, the middle index could hold a value that falls anywhere in the array. This explains why the random case in the Quick Sort algorithm is worse than the descending case.

Even though some of the algorithms have the same big-O classification, they do not have the exact same behaviour. One of the examples is Insertion Sort compared to Selection Sort. For both, their average and worst cases have a $O(n^2)$ complexity. However, generally Insertion Sort sorts through data faster than Selection Sort does. A similar case can be seen with Quick Sort and Merge Sort. Both have $O(n \log n)$ complexity but they have very different behaviour. Merge Sort follows the same general pattern as the rest of the algorithms where descending data is the slowest, followed by random data, then ascending data. For Merge Sort the gap between ascending and descending data is smaller than others. As discussed above, Quick Sort completely breaks this pattern showing the difference in behaviour.

6 – Conclusion

As shown throughout this report, there is no one way to choose a sorting algorithm. Many factors must be considered including size of data, and approximate ordering of the data. If there is a small amount of data (less than 100 elements) a sequential search algorithm like Selection or Insertion Sort will be best. If some or all of this data is pre-sorted Insertion Sort will absolutely

be the fastest algorithm. On the other hand, if the data is very large (more than 100,000 elements) a binary search algorithm like Merge or Quick Sort is the best choice.

How the data is arranged before sorting is another important factor to consider. All the algorithms tested in this report perform best when the data is pre-sorted. This is unrealistic in real scenarios that need sorting algorithms, but if a large portion of data is pre-sorted, it will improve the performance of the algorithms. If data is in mostly descending or random order, the algorithms will take much longer to sort through this data in most cases. If it is completely unknown what order the data is in, Merge Sort would likely be the best algorithm since its best, worst, and average cases were quite close together compared to others. Quick Sort is another good choice if data order is unknown, since it was generally the fastest algorithm tested.

From all the experiments and analysis done through this report, it has been shown what criterion must be considered before choosing a sorting algorithm. There is no one, easy solution that will fit in every case. Careful analysis and consideration are required especially when the data has millions of elements. Choosing incorrectly in these cases can cost hours of program running time. Analyzing each potential algorithm carefully ensures that the best one is selected for each case.

References

Davis, K. (2019, January 30). *Basic algorithms: Insertion sort and pseudo-code*. LinkedIn. Retrieved February 3, 2022, from https://www.linkedin.com/pulse/basic-algorithms-insertion-sort-pseudo-code-loop-invariant-davis/

Merge sort in data structure. TechVidvan. (2021, July 3). Retrieved February 3, 2022, from https://techvidvan.com/tutorials/merge-sort/

Selection sort in data structure. TechVidvan. (2021, September 16). Retrieved February 2, 2022, from https://techvidvan.com/tutorials/selection-sort/

Simple to understand Quick Sort. Department of Math/CS - Home. (n.d.). Retrieved February 3, 2022, from http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/7-Sort/quick-sort1.html

Weiss, M. A. (2012). 7.6.1 Analysis of Mergesort. In Data structures and algorithm analysis in Java (3rd ed., pp. 285–286). Chapter, Pearson.