CPSC 319
Assignment 1

Aleksander Berezowski
Tutorial Section: T03
TA Name: Shopon
Email: aleksander.berezowsk@ucalgary.ca

Experimental Method

The experiment for this assignment was based off of the method detailed in the “CPSC 319 Assignment 1” handout. First each of the four algorithms (selection sort, insertion sort, merge sort, and quick sort) were coded in a Java class named “assign1.” Next, auxiliary code was added to measure run time, allow arguments such as array order and size to be inputted, and allow an output file to be created.

However, there was one major change; instead of running each algorithm and data set once, they were ran multiple times. This was due to preliminary results looking akin to a random number generator, therefore running the algorithm multiple times per data set helped eliminate external factors and create more accurate numbers. For data sets of length 10, 100, 1 000, and 10 000 the algorithms were run 100 times per array arrangement (ascending, descending, random) therefore resulting in 300 algorithm runs per data set length per algorithm. For length 100 000 the algorithms were only run 20 times per set due to each algorithm taking long enough to make 100 runs per set non-feasible. For this same reason there were only 5 runs of the 1 000 000 length arrays per algorithm per arrangement.

After finalizing all code by ensuring it was working, everything was run via a driver class called “main.” All 72 configurations (4 algorithms * 6 lengths * 3 array orders) were automatically run, each iterating several times to give several data points per configuration. All information as successfully outputted into “.txt” files and condensed into excel. Finally, data analysis was performed.

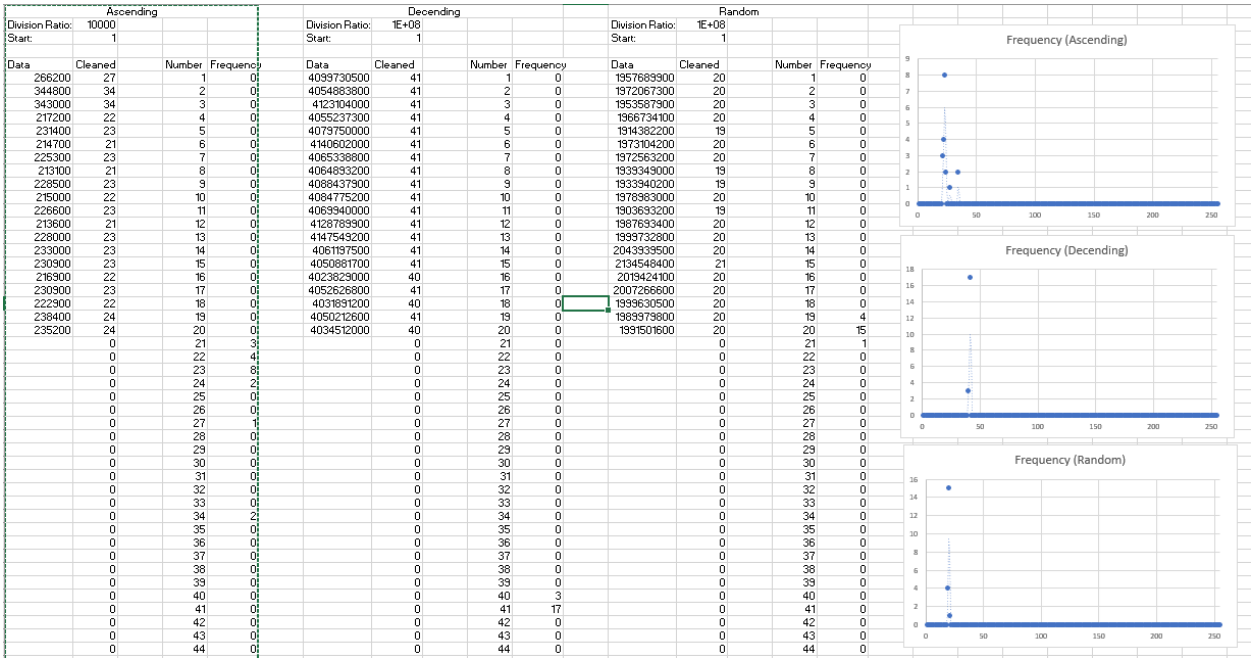
Data collected (tables and graphs)

Below is the data collected from this assignment. This includes:

- Frequency scatter plots of algorithm run time sorted by algorithms (insert, merge, quick, then selection), then array length (10, 100, 1 000, 10 000, 100 000, then 1 000 000), then by array order (ascending, descending, then random).
- Tables of the most frequent times derived from the frequency scatter plots.
- Scatter plots of algorithm run time showing the run time of the algorithms as the array length increases.

Frequency Scatter Plots

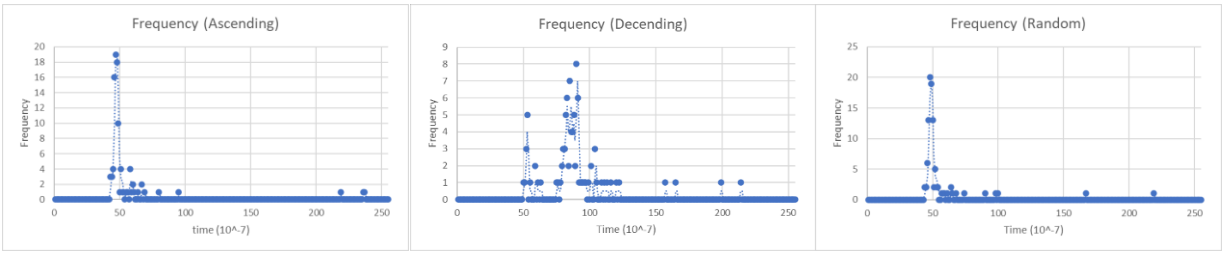
Frequency scatter plots were made by first importing all data for a specific configuration, as the Insertion sort for an ascending array with length 10 000. Then the data was divided by a multiple of 10 to allow for a generalization of data points. Finally, the frequency of each data point was counted using the COUNTIF() function in excel. The data point value and the frequency of said value were then plotted in the scatter plots shown below, allowing a selection of the best data point for the tables used for the rest of data analysis. An example of the excel sheet for data for Insertion sort for array length of 10 000 is shown below:



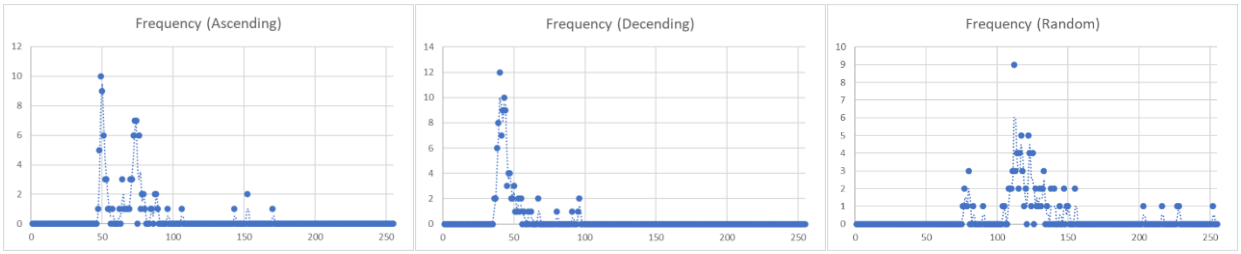
Ascending				Decending				Random			
Division Ratio:	10000			Division Ratio:	1E+08			Division Ratio:	1E+08		
Start:	1			Start:	1			Start:	1		
Data	Cleaned	Number	Frequency	Data	Cleaned	Number	Frequency	Data	Cleaned	Number	Frequency
266200	27	1	0	4099730500	41	1	0	1957689900	20	1	0
344800	34	2	0	4054883800	41	2	0	1972067300	20	2	0
343000	34	3	0	4123104000	41	3	0	1953587900	20	3	0
217200	22	4	0	4055237300	41	4	0	1966734100	20	4	0
231400	23	5	0	4079750000	41	5	0	1914382200	19	5	0
214700	21	6	0	4140602000	41	6	0	1973104200	20	6	0
225300	23	7	0	4065338800	41	7	0	1972563200	20	7	0
213100	21	8	0	4064893200	41	8	0	1939349000	19	8	0
228500	23	9	0	4088437900	41	9	0	1933940200	19	9	0
215000	22	10	0	4084775200	41	10	0	1978983000	20	10	0
226600	23	11	0	4069940000	41	11	0	1903693200	19	11	0
213600	21	12	0	4128789900	41	12	0	1987693400	20	12	0
228000	23	13	0	4147543200	41	13	0	1939732800	20	13	0
233000	23	14	0	4061197500	41	14	0	2043939500	20	14	0
230900	23	15	0	4050881700	41	15	0	2134548400	21	15	0
216900	22	16	0	4023829000	40	16	0	2019424100	20	16	0
230900	23	17	0	4052626800	41	17	0	2007266600	20	17	0
222900	22	18	0	4031891200	40	18	0	1999630500	20	18	0
238400	24	19	0	4050212600	41	19	0	1989979800	20	19	4
235200	24	20	0	4034512000	40	20	0	1991501600	20	20	15
	0	21	3		0	21	0		0	21	1
	0	22	4		0	22	0		0	22	0
	0	23	8		0	23	0		0	23	0
	0	24	2		0	24	0		0	24	0
	0	25	0		0	25	0		0	25	0
	0	26	0		0	26	0		0	26	0
	0	27	1		0	27	0		0	27	0
	0	28	0		0	28	0		0	28	0
	0	29	0		0	29	0		0	29	0
	0	30	0		0	30	0		0	30	0
	0	31	0		0	31	0		0	31	0
	0	32	0		0	32	0		0	32	0
	0	33	0		0	33	0		0	33	0
	0	34	2		0	34	0		0	34	0
	0	35	0		0	35	0		0	35	0
	0	36	0		0	36	0		0	36	0
	0	37	0		0	37	0		0	37	0
	0	38	0		0	38	0		0	38	0
	0	39	0		0	39	0		0	39	0
	0	40	0		0	40	3		0	40	0
	0	41	0		0	41	17		0	41	0
	0	42	0		0	42	0		0	42	0
	0	43	0		0	43	0		0	43	0
	0	44	0		0	44	0		0	44	0

Insertion Sort

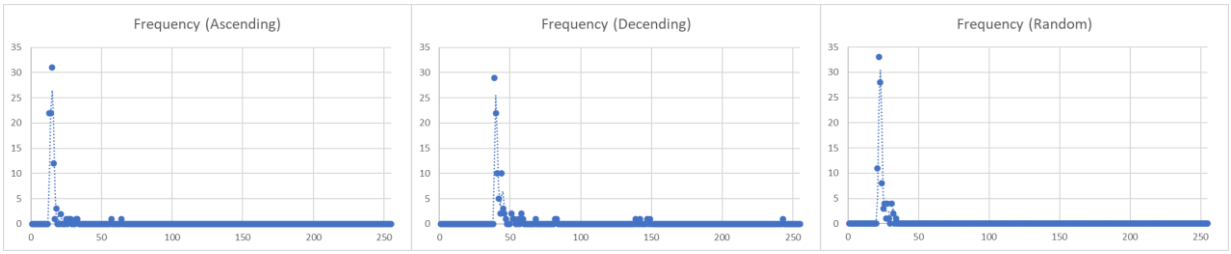
Insertion Sort 10



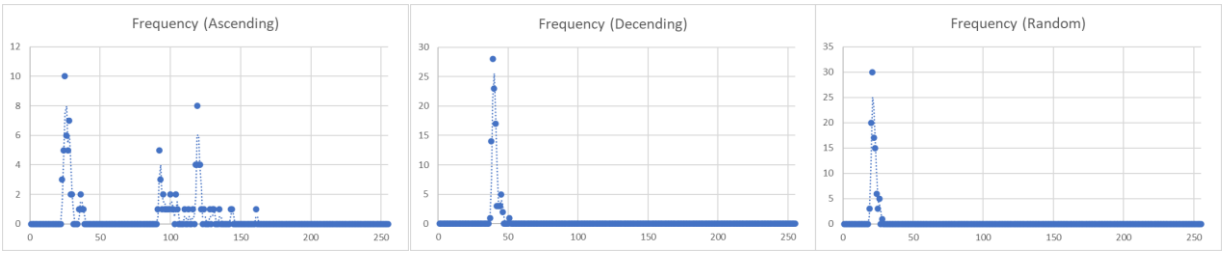
Insertion Sort 100



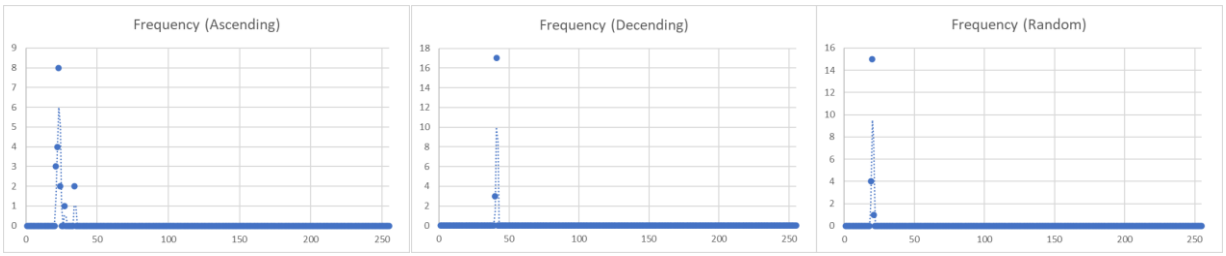
Insertion Sort 1 000



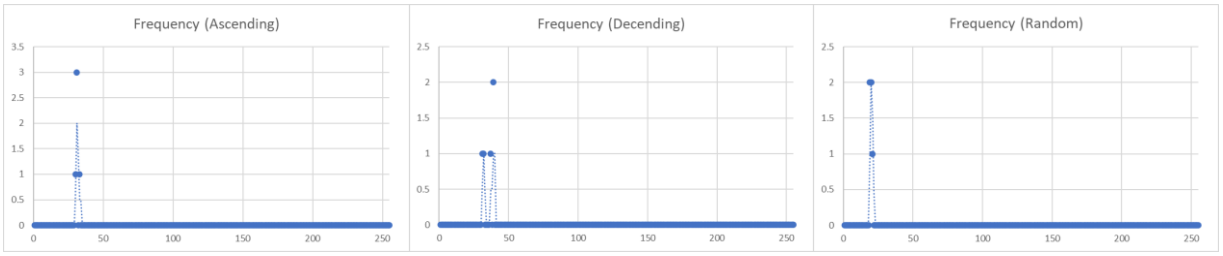
Insertion Sort 10 000



Insertion Sort 100 000

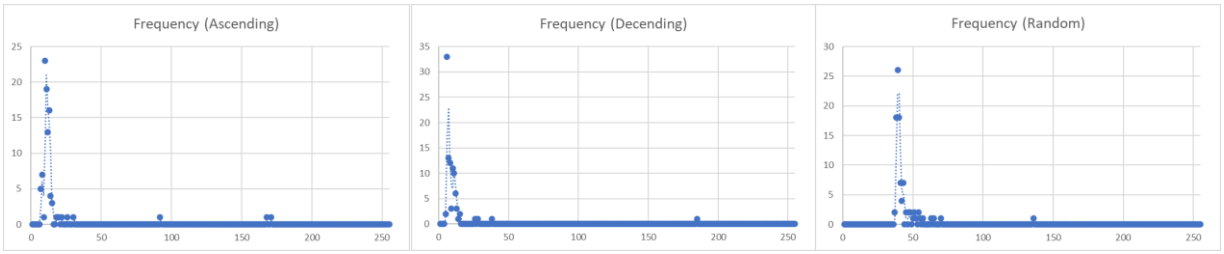


Insertion Sort 1 000 000

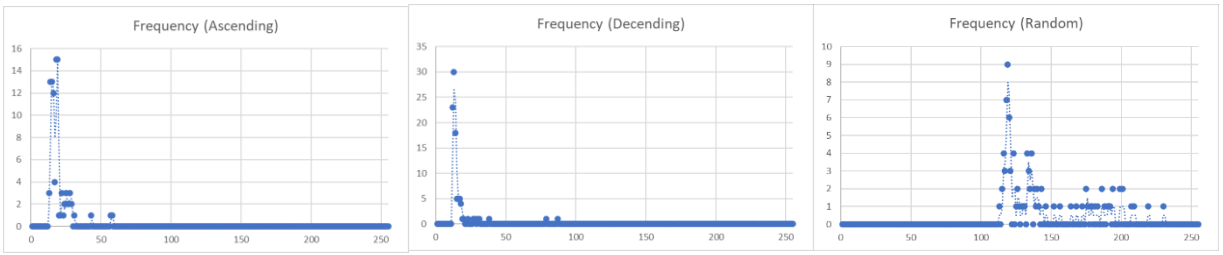


Merge Sort

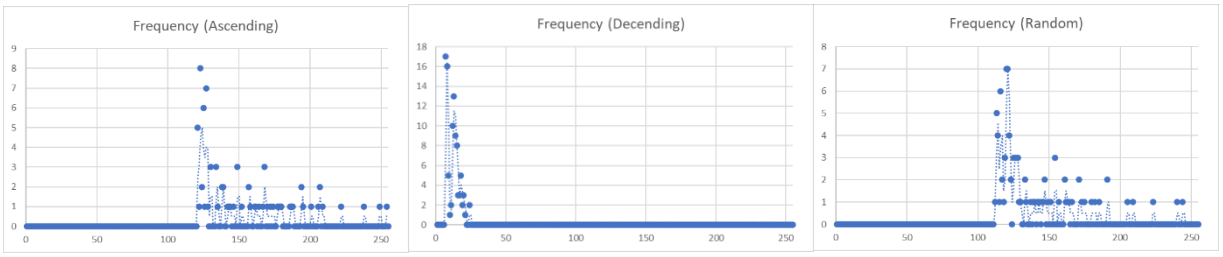
Merge Sort 10



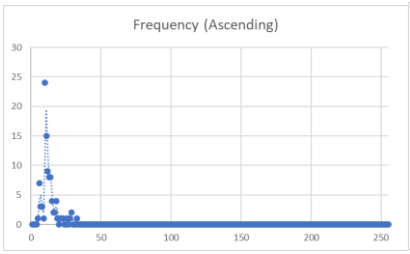
Merge Sort 100

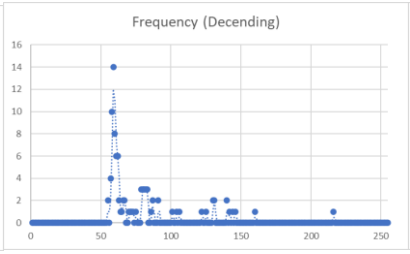


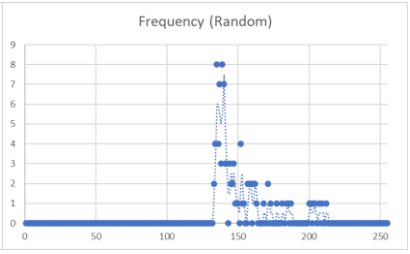
Merge Sort 1 000



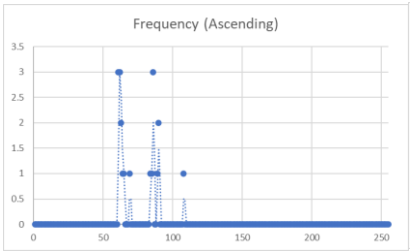
Merge Sort 10 000

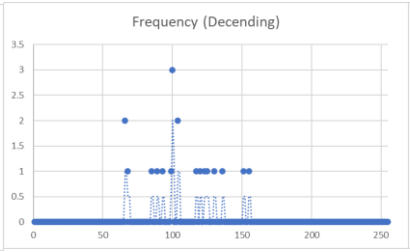


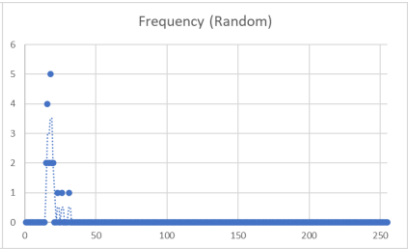




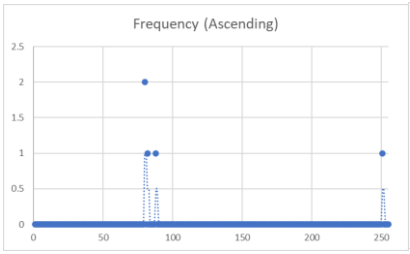
Merge Sort 100 000

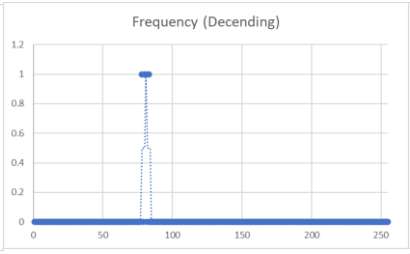


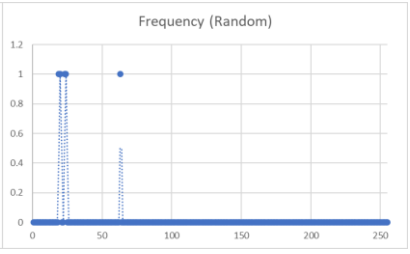




Merge Sort 1 000 000

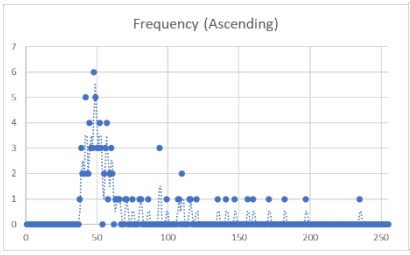


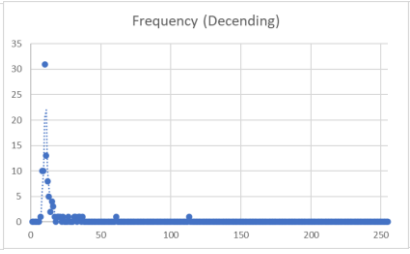


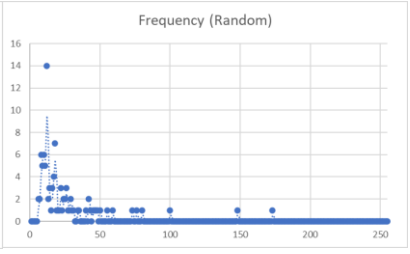


Quick Sort

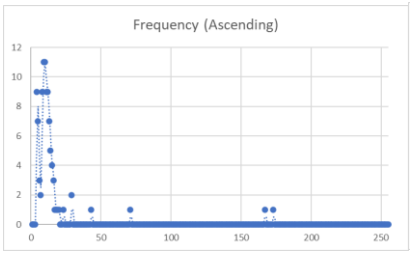
Quick Sort 10

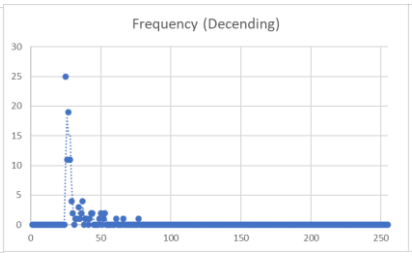


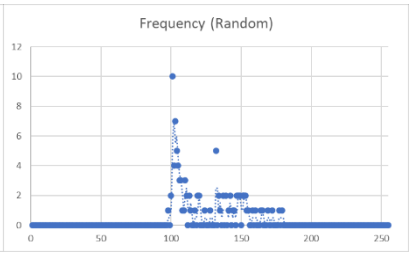




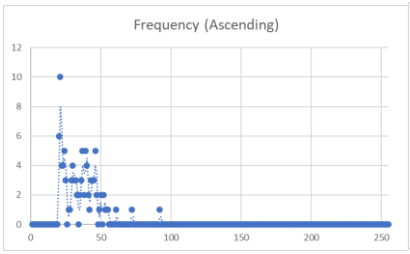
Quick Sort 100

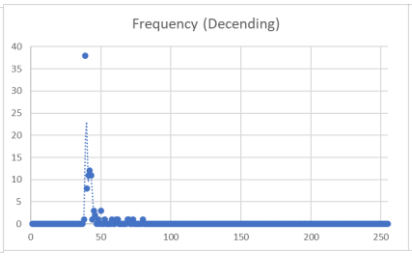


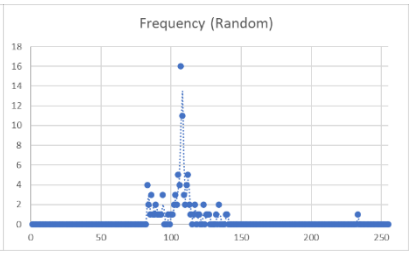




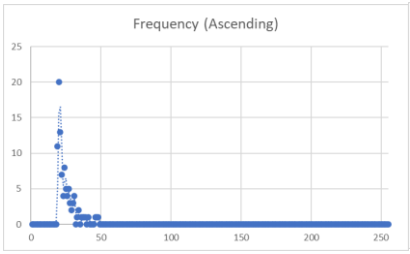
Quick Sort 1 000

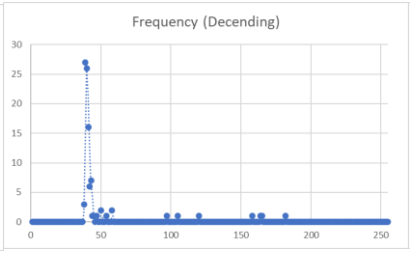


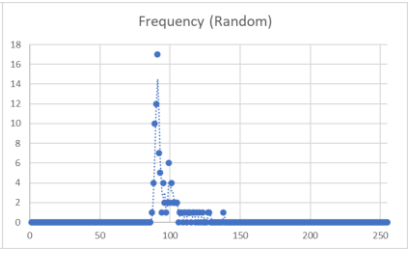




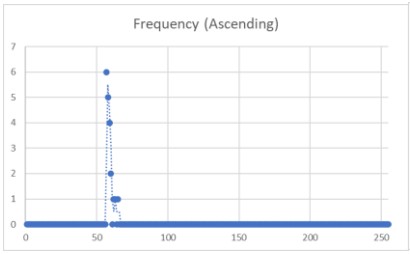
Quick Sort 10 000

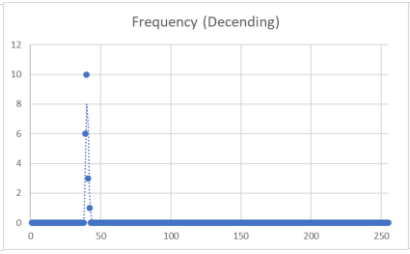


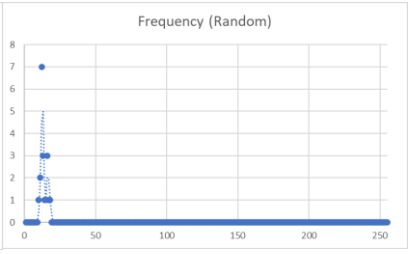




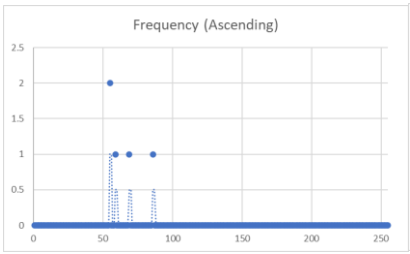
Quick Sort 100 000

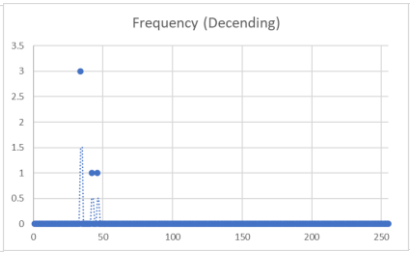


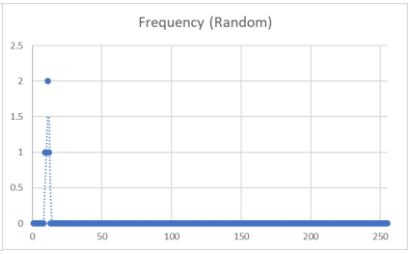




Quick Sort 1 000 000

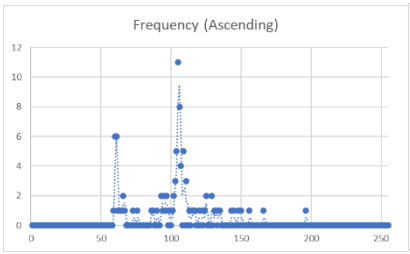


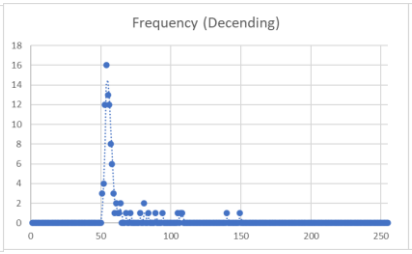


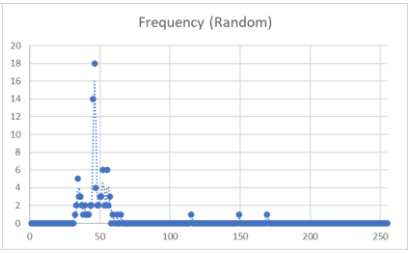


Selection Sort

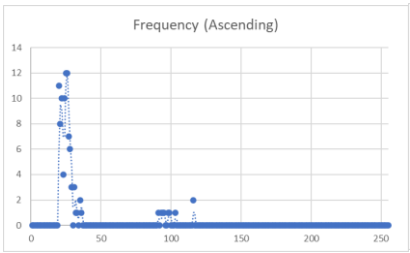
Selection Sort 10

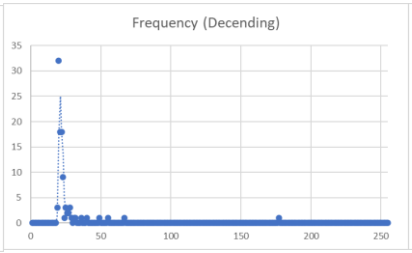


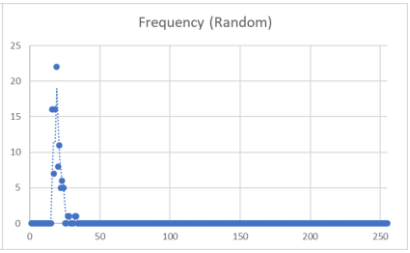




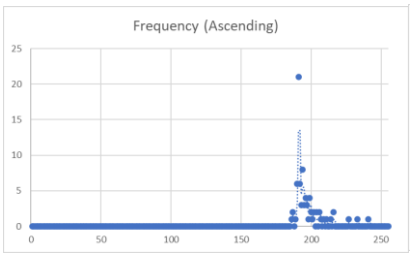
Selection Sort 100

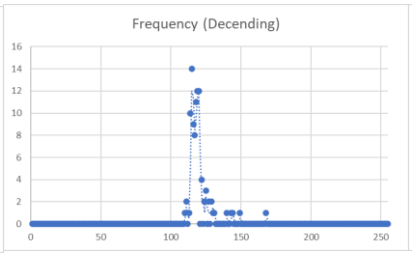


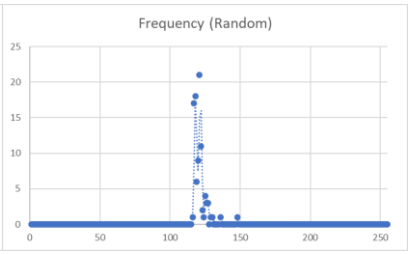




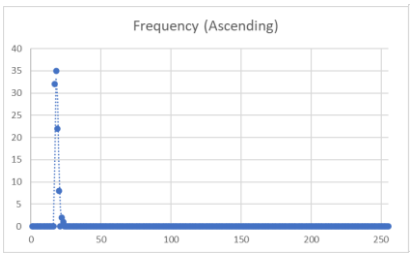
Selection Sort 1 000

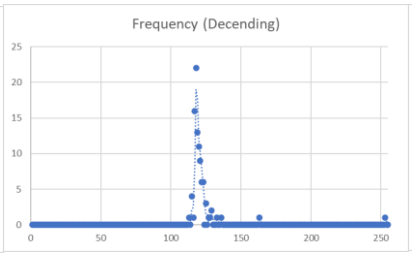


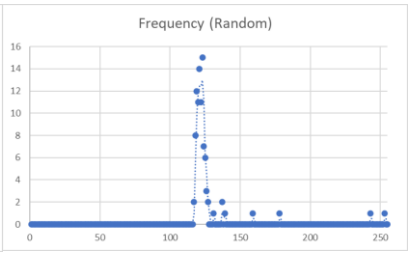




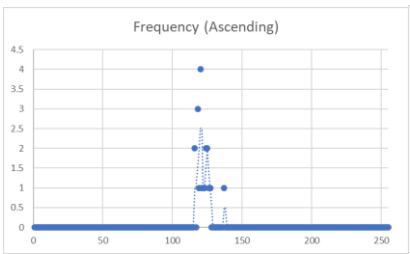
Selection Sort 10 000

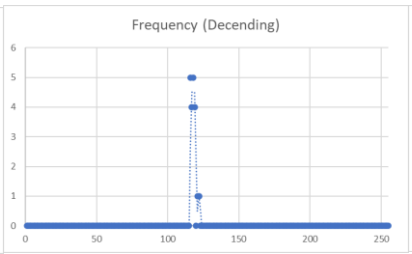


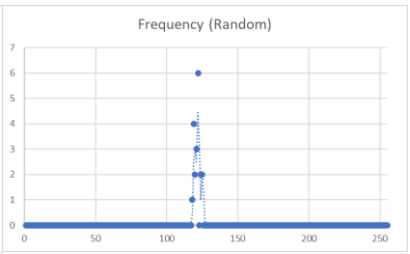




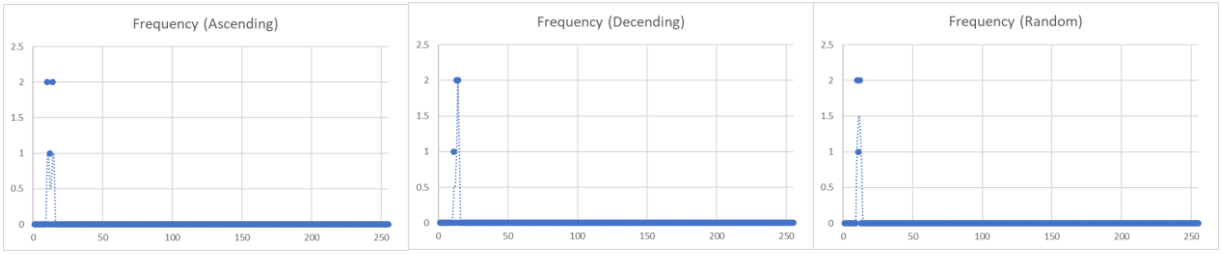
Selection Sort 100 000







Selection Sort 1 000 000



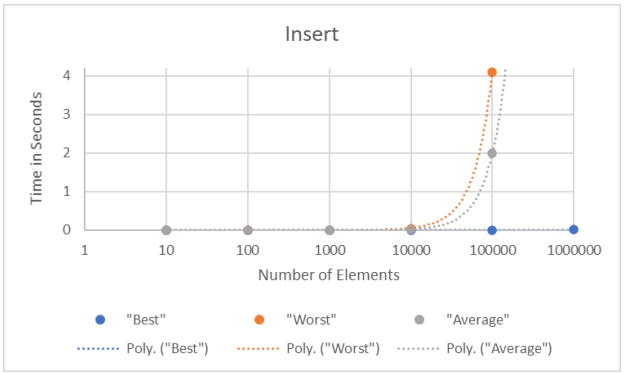
Tables of most Frequent Run Times

Based on the above scatter plots, times were chosen for the tables below that most accurately reflected the experiment. Below are the tables of the data sorted by the sorting algorithm. Please note the graphs often don't look like they change until the final two values (100 000 and 1 000 000) due to how big the jump in time was between these results and the first four values of 10, 100, 1 000 and 10 000.

Insertion Sort

Insert

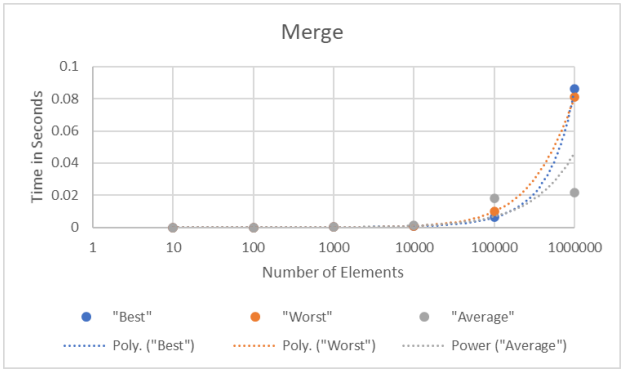
Elements	"Best"	"Worst"	"Average"
10	0.0000047	0.000009	0.0000048
100	0.0000049	0.00004	0.0000112
1000	0.000015	0.00039	0.00022
10000	0.000025	0.039	0.021
100000	0.00023	4.1	2
1000000	0.0031	390	195



Merge Sort

Merge

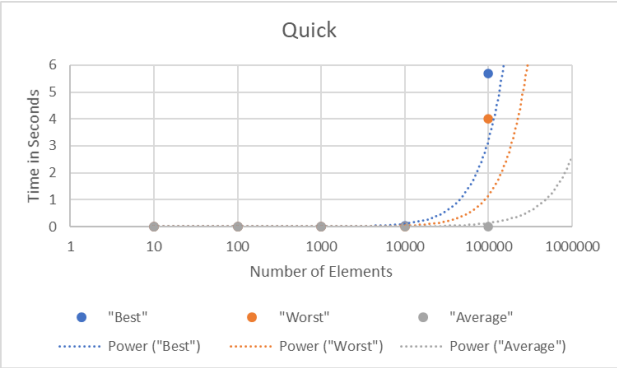
Elements	"Best"	"Worst"	"Average"
10	0.00001	0.000006	0.0000039
100	0.0000185	0.000013	0.0000119
1000	0.000123	0.00007	0.00012
10000	0.001	0.00059	0.00135
100000	0.0062	0.01	0.018
1000000	0.086	0.081	0.0215



Quick Sort

Quick

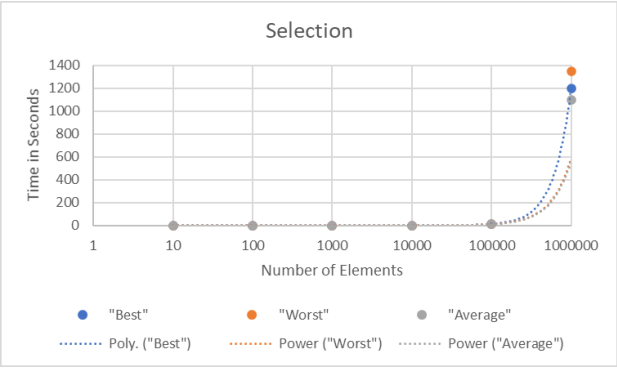
Elements	"Best"	"Worst"	"Average"
10	0.000048	0.00001	0.000012
100	0.000095	0.000025	0.0000101
1000	0.00021	0.00039	0.000107
10000	0.02	0.00039	0.00091
100000	5.7	4	0.012
1000000	550	340	110



Selection Sort

Selection

Elements	"Best"	"Worst"	"Average"
10	0.0000105	0.0000054	0.0000046
100	0.0000255	0.00002	0.000019
1000	0.000191	0.00115	0.00121
10000	0.018	0.118	0.123
100000	12	11.75	12.2
1000000	1200	1350	1100



Data Analysis

Insertion Sort

For arrays in ascending order, insertion sort was consistently the fastest. However, for arrays in descending and random order insertion sort was the 2nd slowest algorithm. The trendline for ascending order suggests a best time complexity of O(n) due to the time consistently being small, whereas the trendlines for random order and descending order both suggest a $O(n^2)$ or higher time complexity due to their run time's exponential growth starting at around 10 000 items. To summarize, the data collected seems to imply the best-case being O(n) and worst-case being $O(n^2)$.

The trend lines from the insertion sort graph reflect these findings for both the best and worst case.

Merge Sort

For arrays in ascending order merge sort started off as the worst algorithm for small array lengths, but then became the fastest algorithm for bigger array lengths. However, for arrays in descending and random order merge sort was substantially faster than all other algorithms. Both of these observations imply a logarithmic growth for both the best and worst case, implying both cases to be O(n lg n).

The trend lines from the merge sort graph do not reflect these findings for the worst case, although they do reflect the findings for the best case.

Quick Sort

For arrays in ascending order, quick sort was consistently the 2nd slowest algorithm by a substantial margin. However, for arrays in descending and random order merge sort started off as the worst algorithm for small array lengths, but then became the second fastest algorithm for bigger array lengths and was the 2nd fastest of all the algorithms, only behind merge sort. This implies a similar time complexity to merge sort of O(n lg n).

The trend lines from the quick sort graph not reflect these findings for both the best and worst case.

Selection Sort

For arrays in any order, selection sort was either the fastest or second fastest for small data sets but was substantially worse the bigger data sets got. It performed the worst for all array orders for data sets bigger than 1 000, and not by a small margin. This implies a best and worst case of $O(n^2)$ at the least.

The trend lines from the selection sort graph reflect these findings for both the best and worst case.

Complexity Analysis

Assumptions

During this complexity analysis, 2 assumptions were made:

- All assignments, declarations, arithmetic operations, value fetching, *if* statements, and initializations will count as 1 operation
    - i.e. int i = 0; has a complexity of O(1)
    - i.e. var j, k = 5; has a complexity of O(2) based pm the summation rule
- Single loops have a complexity of O(n)
    - i.e. A single *for* or *while* loop will have a complexity of O(n)
    - i.e. Two nested *while* or *for* loops have a complexity of $O(n^2)$ based on the product rule
    - i.e. Three *while* or *for* loops outside of each other have a complexity of O(3n) based on the summation rule

Selection Sort

```
Line 1   public void selectionSort(){
Line 2       int n = myArray.length;
Line 3
Line 4       for (int i = 0; i < n-1; i++) {
Line 5          int min_idx = i;
Line 6
Line 7          for (int j = i+1; j < n; j++)
Line 8             if (myArray[j] < myArray[min_idx])
Line 9                min_idx = j;
Line 10
Line 11         int temp = myArray[min_idx];
Line 12         myArray[min_idx] = myArray[i];
Line 13         myArray[i] = temp;
Line 14      }
Line 15   }
```

- **Best Case:** $O(n^2)$
  Happens when the array is in ascending order because the algorithm doesn't need to swap any array elements. The double, nested *for* loops on line 4 and Line 7 mean a complexity of $O(n^2)$ due to each loop having a complexity of O(n) and the product rule.
- **Worst Case:** $O(n^2)$
  Happens when the array is in descending order because the algorithm needs to swap elements the maximum amount of times possible, which is *n-1* due to the condition on the outer *for* loop of *i < n-1*. As with the "Best Case", the double, nested *for* loops on line 4 and Line 7 mean a complexity of $O(n^2)$.

Insertion Sort

```
Line 1   public void insertionSort(){
Line 2       int n = myArray.length;
Line 3
Line 4       for (int i = 1; i < n; ++i) {
Line 5          int key = myArray[i];
Line 6          int j = i - 1;
Line 7
Line 8          while (j >= 0 && myArray[j] > key) {
Line 9             myArray[j + 1] = myArray[j];
Line 10            j = j - 1;
Line 11         }
Line 12         myArray[j + 1] = key;
Line 13      }
Line 14   }
```

- **Best Case:** O(n)
  Happens when the array is in ascending order because the algorithm only needs to make one comparison per position, resulting in *n-1* comparisons and *2(n-1)* redundant comparisons. The *for* loop on Line 4 is used and the *while* loop on Line 8 is not used, resulting in a complexity of O(n).
- **Worst Case:** $O(n^2)$
  Happens when the array is in descending order because the algorithm each element needs to be swapped with the element before it one position at a time until the array is in proper order. This means for the *i* iterations of the outer *for* loop on Line 4 there are *i* comparisons by the *while* loop on Line 8, resulting in a complexity of $O(n^2)$.

Merge Sort

```
Line 1   public int[] merge(int[] arrayOne, int[] arrayTwo) {
Line 2       int[] returnArray = new int[arrayOne.length + arrayTwo.length];
Line 3       int arrayOneIndex = 0;
Line 4       int arrayTwoIndex = 0;
Line 5
Line 6       for (int i = 0; i < returnArray.length; i++) {
Line 7          if (arrayOneIndex < arrayOne.length && arrayTwoIndex < arrayTwo.length) {
Line 8             if (arrayOne[arrayOneIndex] > arrayTwo[arrayTwoIndex]) {
Line 9                returnArray[i] = arrayTwo[arrayTwoIndex];
Line 10               arrayTwoIndex++;
Line 11            } else {
Line 12               returnArray[i] = arrayOne[arrayOneIndex];
Line 13               arrayOneIndex++;
Line 14            }
Line 15         } else if (arrayOneIndex < arrayOne.length) {
Line 16            returnArray[i] = arrayOne[arrayOneIndex];
Line 17            arrayOneIndex++;
Line 18         } else {
```

```
Line 19             returnArray[i] = arrayTwo[arrayTwoIndex];
Line 20             arrayTwoIndex++;
Line 21           }
Line 22       }
Line 23       return returnArray;
Line 24    }
Line 25
Line 26 public int[] mergeSortAlg(int[] sortArray){
Line 27       if(sortArray.length == 1){
Line 28          return sortArray;
Line 29       }
Line 30
Line 31       int [] arrayOne = Arrays.copyOfRange(sortArray, 0, sortArray.length/2);
Line 32       int [] arrayTwo = Arrays.copyOfRange(sortArray, sortArray.length/2,
          sortArray.length);
Line 33
Line 34       arrayOne = mergeSortAlg(arrayOne);
Line 35       arrayTwo = mergeSortAlg(arrayTwo);
Line 36
Line 38       return merge(arrayOne, arrayTwo);
Line 39    }
```

- **Best Case:** O(n lg n)
  Happens when the array is in ascending or descending order because the algorithm is able to divide arrays into two subarrays of equal length, meaning each subarray would be n / 2 in length. The algorithm would then create four new subarrays from the two previously made by splitting each in half, meaning each subarray would now be n / 4 in length. This logarithmic method of dividing arrays combined with then iterating over each subarray while combining them results in a complexity of O(n lg n).
- **Worst Case:** O(n lg n)
  Happens when the array is in a random order when the maximum number of comparisons. However, the time complexity of merge sort is always O(n lg n) because it always halves the arrays with a complexity of O(1) and then takes linear time of O(n) to merge the halves, and does this O(lg n) times for a total of O(n lg n).

Quick Sort

```
Line 1  public void quickSortAlg(int arr[], int l, int h)
Line 2     {
Line 3        int[] stack = new int[h - l + 1];
Line 4        int top = -1;
Line 5
Line 6        stack[++top] = l;
Line 7        stack[++top] = h;
Line 8
Line 9        while (top >= 0) {
Line 10          h = stack[top--];
```

```
Line 11          l = stack[top--];
Line 12          int p = partition(arr, l, h);
Line 13
Line 14          if (p - 1 > l) {
Line 15             stack[++top] = l;
Line 16             stack[++top] = p - 1;
Line 18          }
Line 19
Line 20          if (p + 1 < h) {
Line 21             stack[++top] = p + 1;
Line 22             stack[++top] = h;
Line 23          }
Line 24       }
Line 25    }
Line 26
Line 27    static int partition(int arr[], int low, int high)
Line 28    {
Line 29       int pivot = arr[high];
Line 30
Line 31       int i = (low - 1);
Line 32       for (int j = low; j <= high - 1; j++) {
Line 33          if (arr[j] <= pivot) {
Line 34             i++;
Line 35
Line 36             int temp = arr[i];
Line 37             arr[i] = arr[j];
Line 38             arr[j] = temp;
Line 39          }
Line 40       }
Line 41       int temp = arr[i + 1];
Line 42       arr[i + 1] = arr[high];
Line 43       arr[high] = temp;
Line 44       return i + 1;
Line 45    }
```

- **Best Case:** O(n lg n)
  Happens when the array is in ascending order because the algorithm is able to divide arrays into two subarrays of equal length, meaning each subarray would be n / 2 in length. If the pivot is correctly chosen the algorithm would then create four new subarrays from the two previously made by splitting each in half, meaning each subarray would now be n / 4 in length. This logarithmic method of dividing arrays combined with iterating over each array results in a complexity of O(n lg n).
- **Worst Case:** $O(n^2)$
  Happens when the array is in descending order because the algorithm splits the array into subarrays of length *n-1* and length 1 due to the pivot being either the smallest or largest array element. If this continues, the program will create arrays O(n) times. Multiplying this by iterating over the array gives a worst case of $O(n^2)$.

Interpretation

What does the empirical data and complexity analysis tell us about the algorithms?

The data analysis and the complexity analysis for each algorithm matched for almost every algorithm.

As predicted insertion sort's best-case was O(n) while the worst case was $O(n^2)$. This implies insertion sort should only be used for small data sets as it will either out perform or match every other algorithm in time complexity, but for bigger data sets it is considerably worse than O(n lg n) algorithms.

As predicted merge sort's best-case and worst case were both O(n lg n). This implies merge sort should be used for large data sets as it will either out perform or match every other algorithm in time complexity, but for medium to small data sets it will be worse than O(n) algorithms.

As predicted quick sort's best-case was O(n lg n), however the data analysis and complexity analysis for quick sort differed. Complexity analysis predicted a worst case of $O(n^2)$ while empirical data showed a worse cast of O(n lg n). This could be due to bigger array lengths only being tested a few times, therefore skewing the empirical data. The combination of data and complexity analysis suggests quick sort should be used for medium to large data sets, although it could possible perform worse than merge sort due to a worse time complexity in the worse case scenario.

As predicted selection sort's best and worse case were both $O(n^2)$. This implies selection sort should only be used for small data sets as it will usually out perform or match every other algorithm in time complexity, but for bigger data sets it is considerably worse than O(n lg n) and O(n) algorithms.

How do the algorithms compare with each other?

For small data sets, insertion sort is the best algorithm to use because at best it will be exponentially faster than selection sort and logarithmically faster than both merge and quick sort, and at worst the same speed as selection sort and quick sort, yet still faster than merge sort.

For large data sets, merge sort is the best if the data is not in ascending order. In the both the best and worst cases it is faster or equal to quick sort and selection sort, only being slower than insertion sort during unlikely best-case scenarios.

Does the order of input matter?

Input order does matter for algorithm speed. Ordered inputs tend to allow the algorithm to run faster due to less comparisons and swaps needing to be done. Therefore, inputs in reverse order cause the most comparisons and swaps to be done. Quick sort highlights this because when inputs are ordered it is O(n lg n) complexity, but in reverse order the time complexity jumps all the way to $O(n^2)$, which is much slower.

How do algorithms within the same big-O classification compare to each other?

Selection sort and insertion sort both have the same worst-case time complexity. However, selection sort took more time in almost every case (shown by the bottom right table), showing that big-O classification is not the only factor that matters when analyzing algorithms. This is further proven by merge and quick sort as they both have the same best-case time complexity, however merge sort outperformed quick sort by a massive margin (shown by the bottom left table).

Best				
Elements	Insert	Merge	Quick	Selection
10	0.0000047	0.00001	0.000048	1.05E-05
100	0.0000049	1.85E-05	0.000095	2.55E-05
1000	0.000015	0.000123	0.00021	0.000191
10000	0.000025	0.001	0.02	0.018
100000	0.00023	0.0062	5.7	12
1000000	0.0031	0.086	550	1200

Worst				
Elements	Insert	Merge	Quick	Selection
10	0.000009	0.000006	0.00001	5.4E-06
100	0.00004	0.000013	0.000025	0.00002
1000	0.00039	0.00007	0.00039	0.00115
10000	0.039	0.00059	0.00039	0.118
100000	4.1	0.01	4	11.75
1000000	390	0.081	340	1350

Conclusions

What algorithms are appropriate for practical use when sorting either small or large amounts of data?

For small data sets under 100 elements, this analysis has empirically proven select sort to be the quickest algorithm regardless of algorithm order. For small data sets over 100 elements, insertion sort is the best algorithm to use because at best it will be exponentially faster than selection sort and logarithmically faster than both merge and quick sort, and at worst the same speed as selection sort and quick sort, yet still faster than merge sort; this conclusion is also empirically proven by the collected data.

For large data sets, merge sort is the best if the data is not in ascending order. In the both the best and worst cases it is faster or equal to quick sort and selection sort, only being slower than insertion sort during unlikely best-case scenarios. All empirical data collected further supports this statement.

Are there any limitations with any of the algorithms?

Recursive algorithms can run into memory problems as array size becomes bigger. An example being the implementation of quick sort in this experiment was originally implemented as a recursive algorithm, however due to it taking up too much memory the algorithm ran into stack overflow errors, cause the algorithm to be switched to an iterative implementation.

Further potential limitations are if the data is usually given in a worst-case scenario, which would negatively affect algorithms that have a more complex worse-case time complexity than their best-case time complexity.

How does your analysis support these conclusions?

The tables below showing the best-case and worst-case for each algorithm supports these conclusions.

For small data sets under 100 elements, the empirical data below proves select sort to be the quickest algorithm regardless of algorithm order. For small data sets over 100 elements, insertion sort is shown to be the best algorithm For large data sets, the data collected supports merge sort is the best if the data is not in ascending order.

Best				
Elements	Insert	Merge	Quick	Selection
10	0.0000047	0.00001	0.000048	1.05E-05
100	0.0000049	1.85E-05	0.000095	2.55E-05
1000	0.000015	0.000123	0.00021	0.000191
10000	0.000025	0.001	0.02	0.018
100000	0.00023	0.0062	5.7	12
1000000	0.0031	0.086	550	1200

Worst				
Elements	Insert	Merge	Quick	Selection
10	0.000009	0.000006	0.00001	5.4E-06
100	0.00004	0.000013	0.000025	0.00002
1000	0.00039	0.00007	0.00039	0.00115
10000	0.039	0.00059	0.00039	0.118
100000	4.1	0.01	4	11.75
1000000	390	0.081	340	1350

Bibliography

1. G. Case, “How do I generate random integers within a specific range in Java?,” Stack Overflow, 12-Dec-2008. [Online]. Available: https://stackoverflow.com/questions/363681/how-do-i-generate-random-integers-within-a-specific-range-in-java.
2. “Insertion sort,” GeeksforGeeks, 08-Jul-2021. [Online]. Available: https://www.geeksforgeeks.org/insertion-sort/.
3. “Iterative quick sort,” GeeksforGeeks, 06-Sep-2021. [Online]. Available: https://www.geeksforgeeks.org/iterative-quick-sort/.
4. “Java Create and Write To Files,” w3schools. [Online]. Available: https://www.w3schools.com/java/java_files_create.asp.
5. KidUncertainty , JoTheKhan, and Evulrabbitz, “R/learnprogramming - [java] quicksort giving Stackoverflowerror for large lists and I have no clue why,” reddit, 2014. [Online]. Available: https://www.reddit.com/r/learnprogramming/comments/1yzs5z/java_quicksort_giving_stackoverflowerror_for/.
6. M. Sambol, “Insertion sort in 2 minutes,” YouTube, 17-Jul-2016. [Online]. Available: https://www.youtube.com/watch?v=JU767SDMDvA.
7. M. Sambol, “Merge sort in 3 minutes,” YouTube, 30-Jul-2016. [Online]. Available: https://www.youtube.com/watch?v=4VqmGXwpLqc.
8. M. Sambol, “Quick sort in 4 minutes,” YouTube, 14-Aug-2016. [Online]. Available: https://www.youtube.com/watch?v=Hoixgm4-P4M&t=49s.
9. M. Sambol, “Selection sort in 3 minutes,” YouTube, 19-Jul-2016. [Online]. Available: https://www.youtube.com/watch?v=g-PGLbMth_g.
10. “Quicksort,” GeeksforGeeks, 25-Jan-2022. [Online]. Available: https://www.geeksforgeeks.org/quick-sort/.
11. “Selection sort,” GeeksforGeeks, 21-Jan-2022. [Online]. Available: https://www.geeksforgeeks.org/selection-sort/.