|  |  |
| --- | --- |
| **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

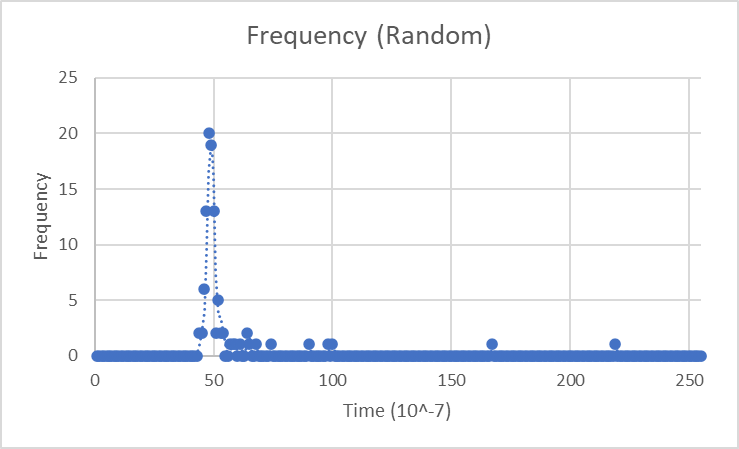
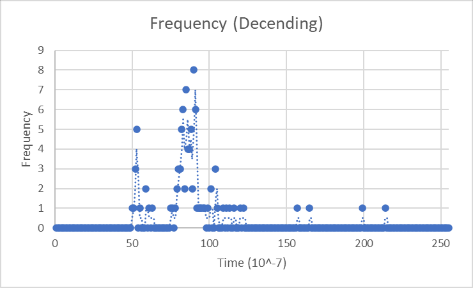
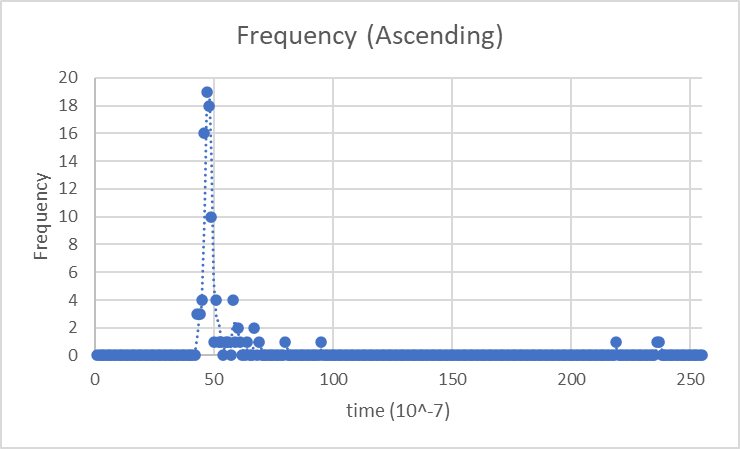
Frequency scatter plots were made by first importing all data for a specific configuration, as the Insertion sort for an ascending array with length 10 000. Then the data was divided by a multiple of 10 to allow for a generalization of data points. Finally, the frequency of each data point was counted 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:

Table

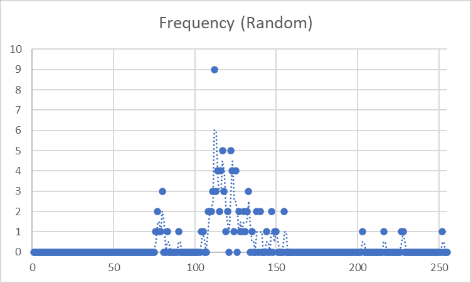
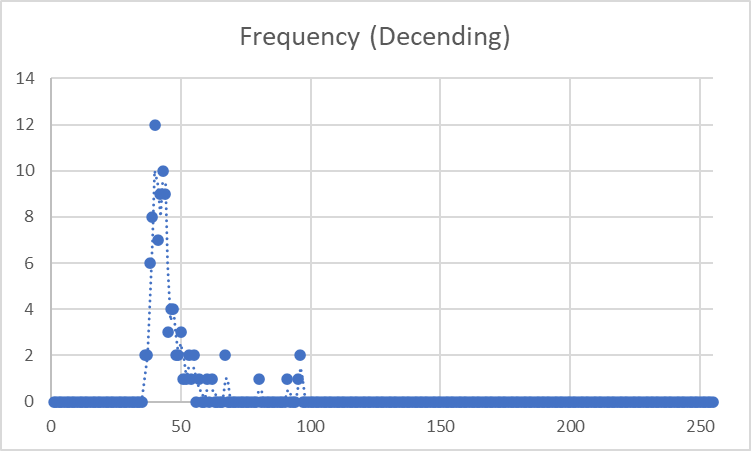
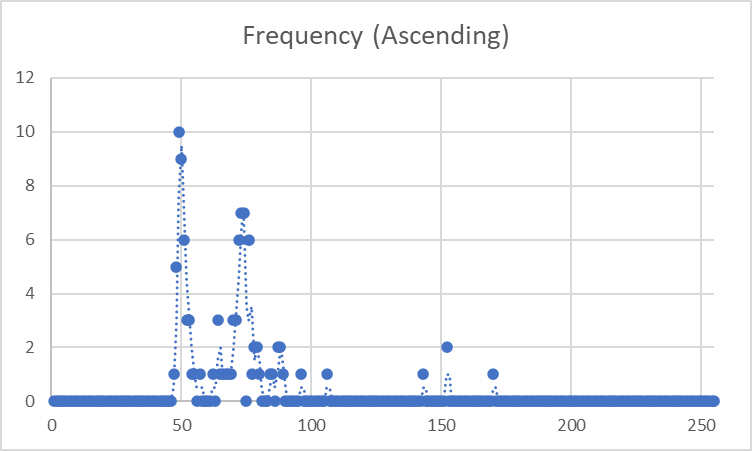
Description automatically generated

### Insertion Sort

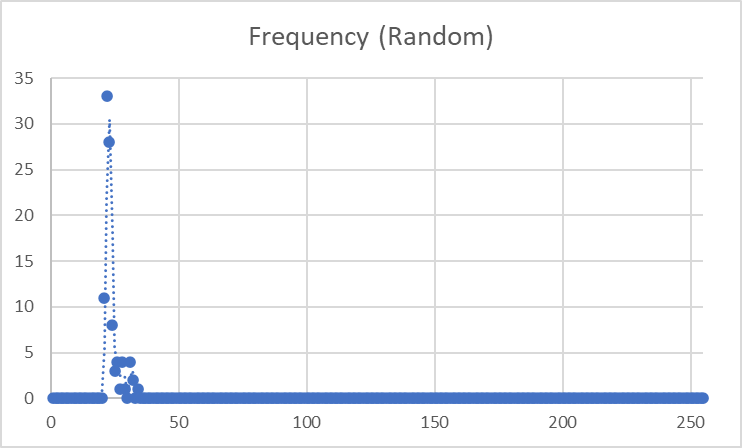
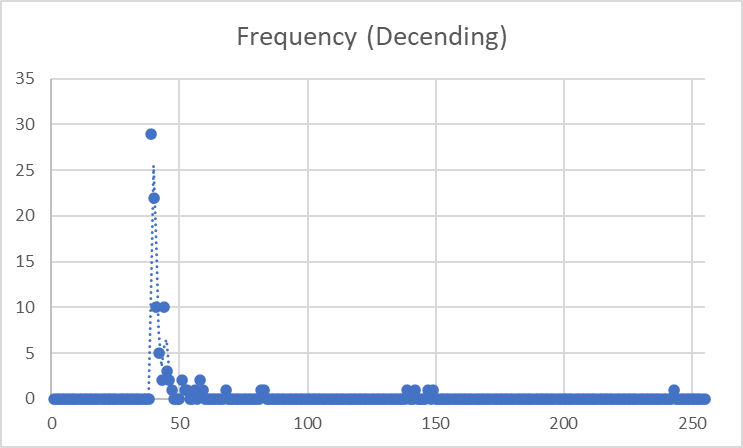
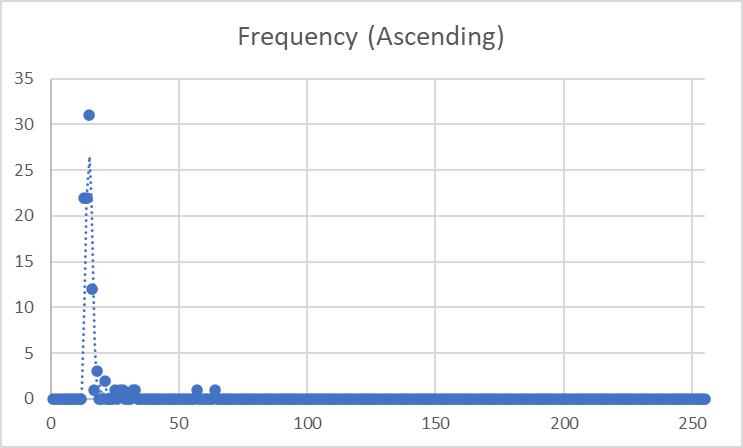
Insertion Sort 10



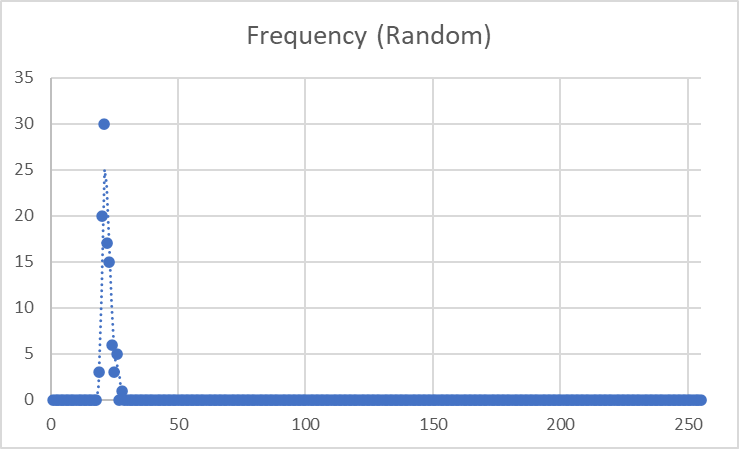
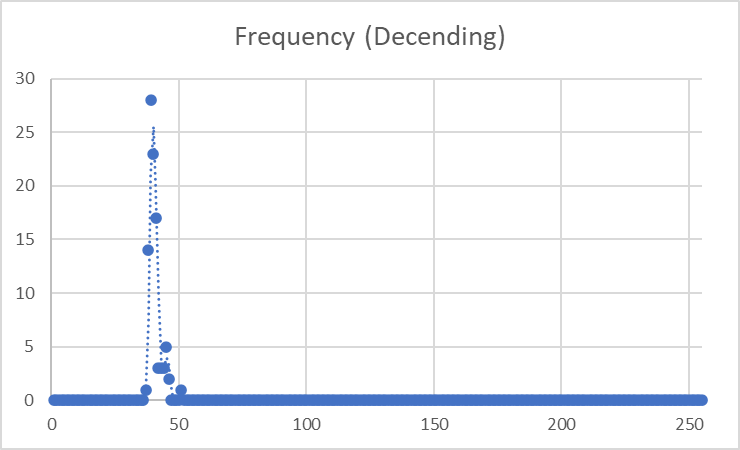
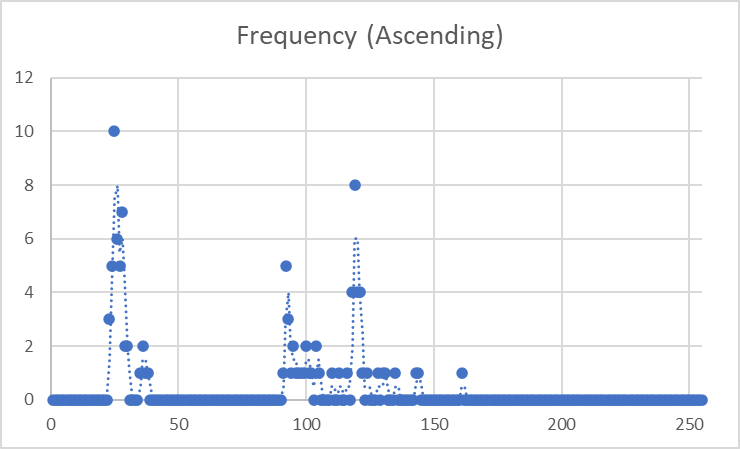
Insertion Sort 100



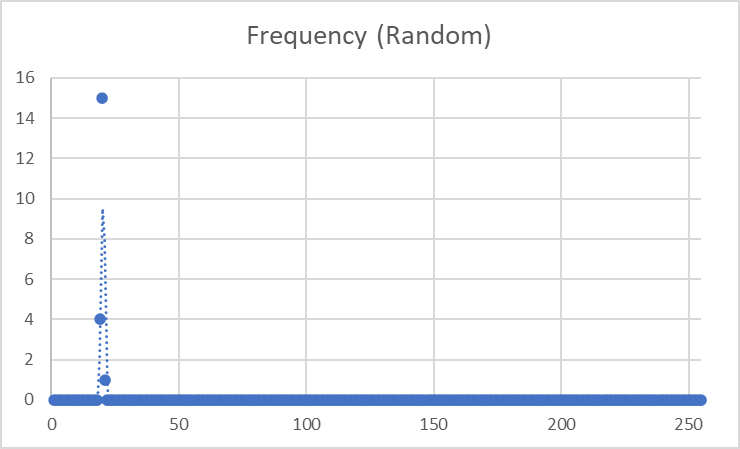
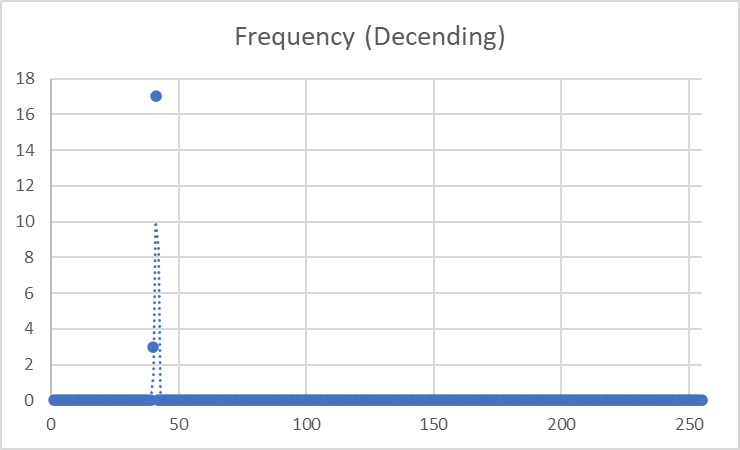
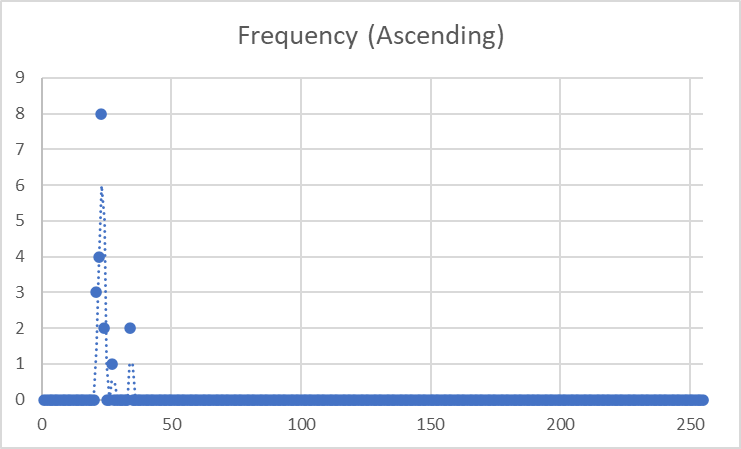
Insertion Sort 1 000



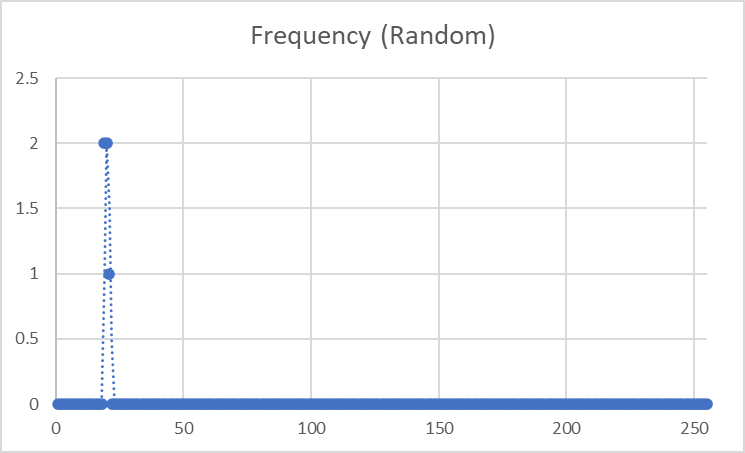
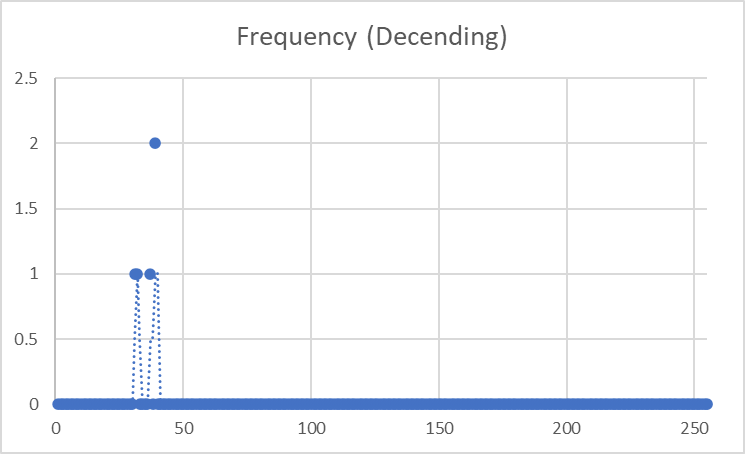
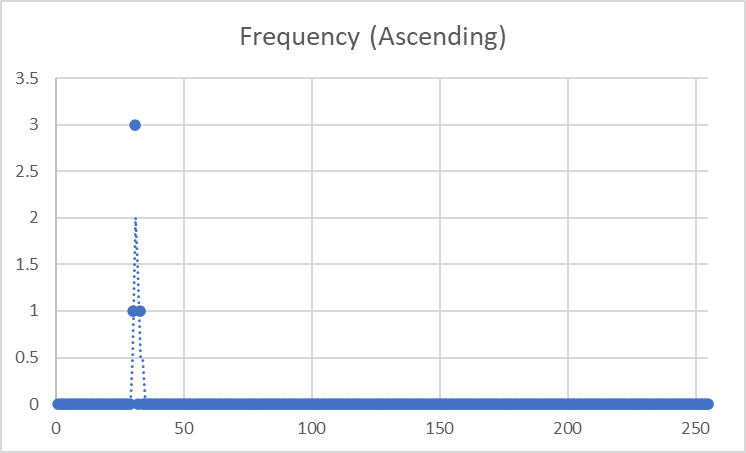
Insertion Sort 10 000



Insertion Sort 100 000

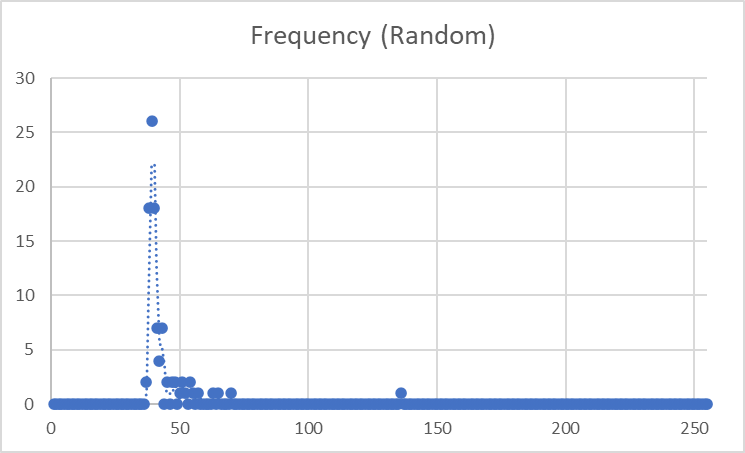
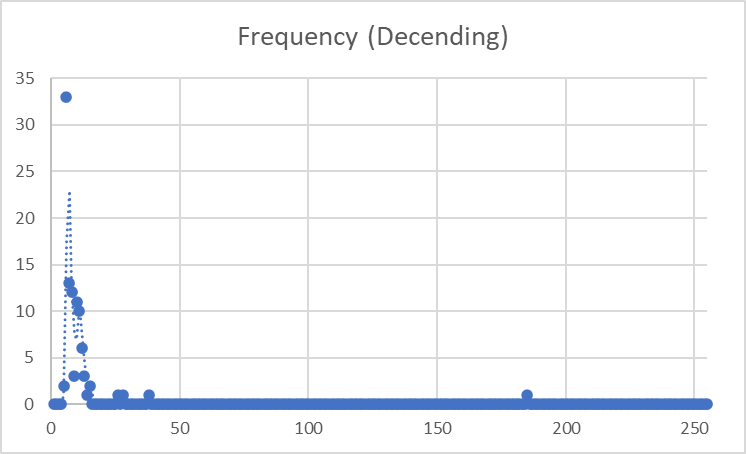
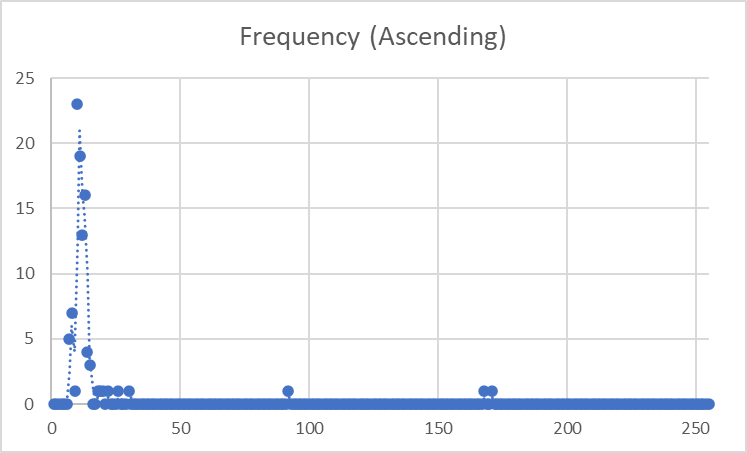


Insertion Sort 1 000 000

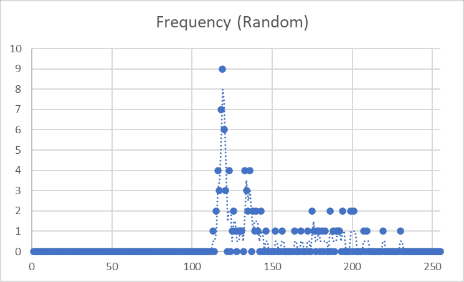
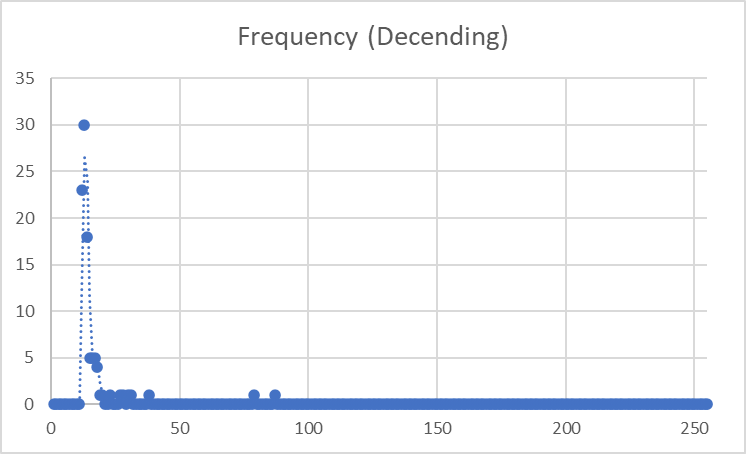
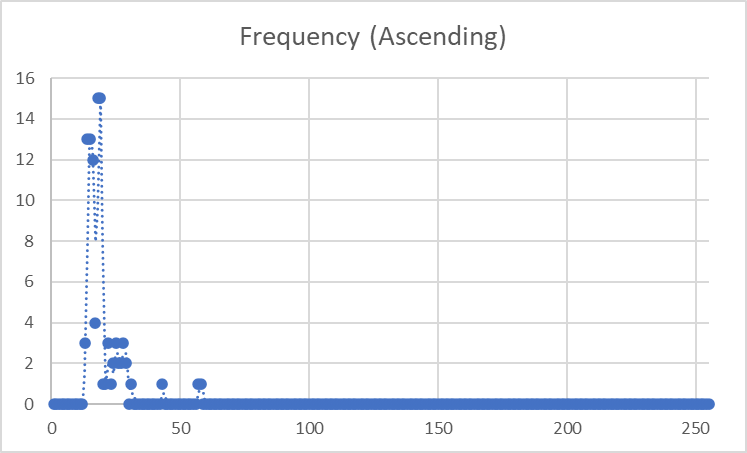


### Merge Sort

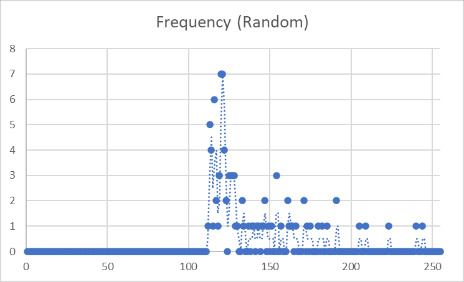
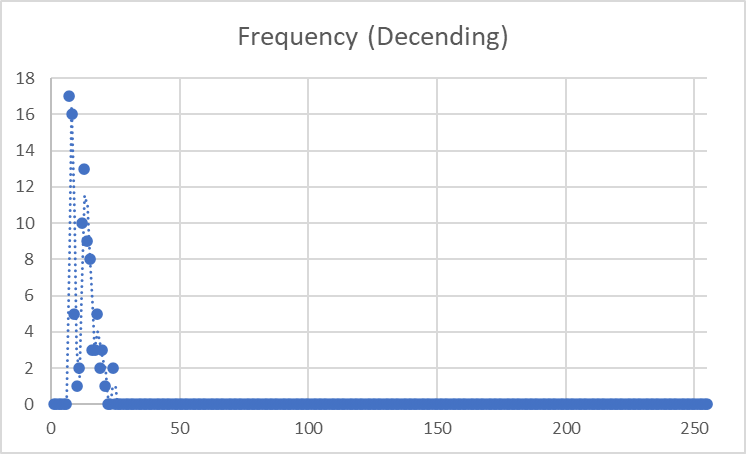
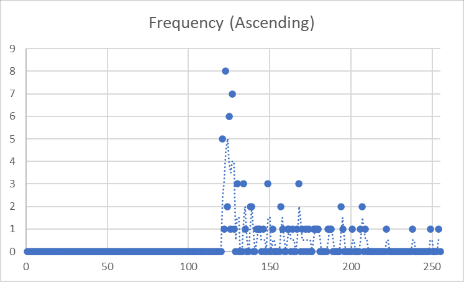
Merge Sort 10



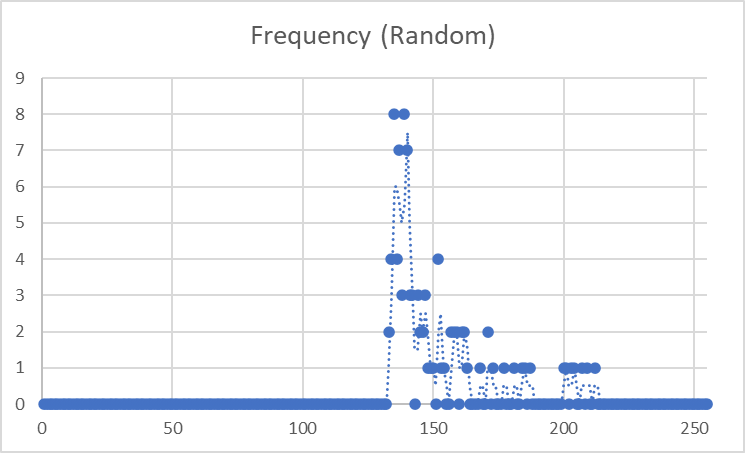
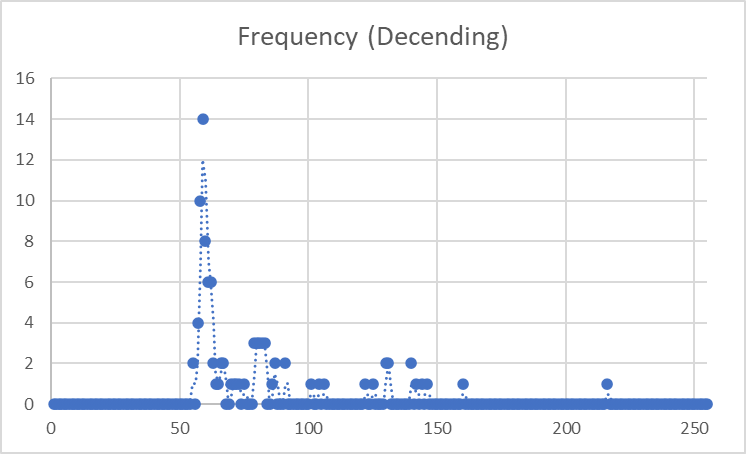
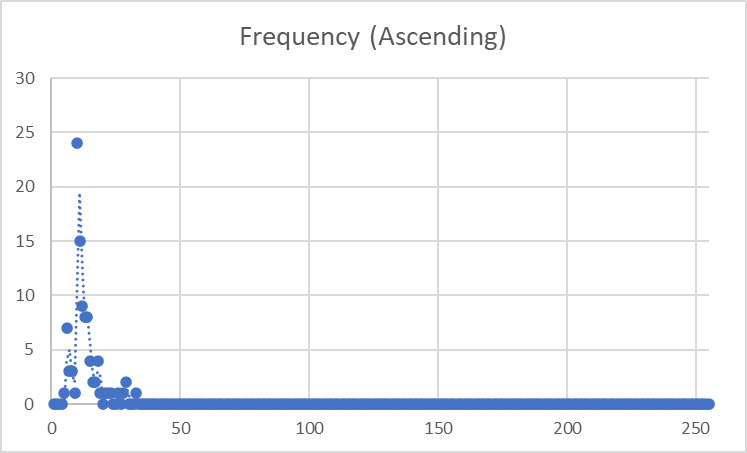
Merge Sort 100



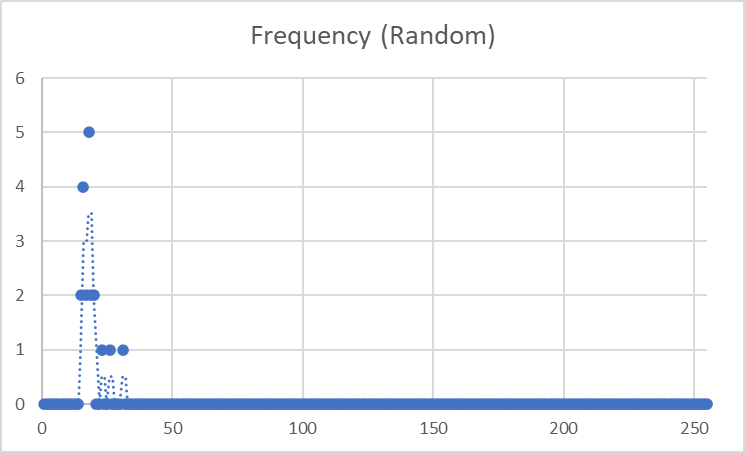
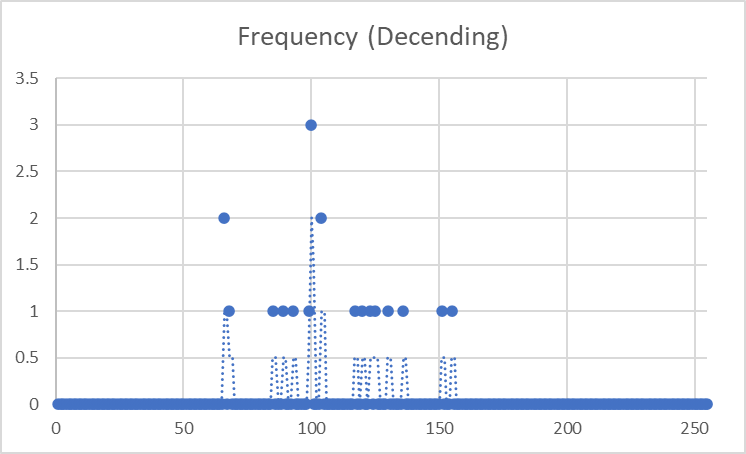
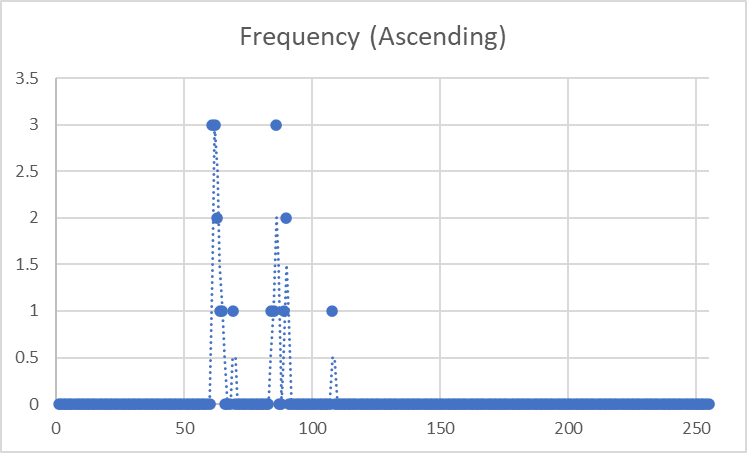
Merge Sort 1 000



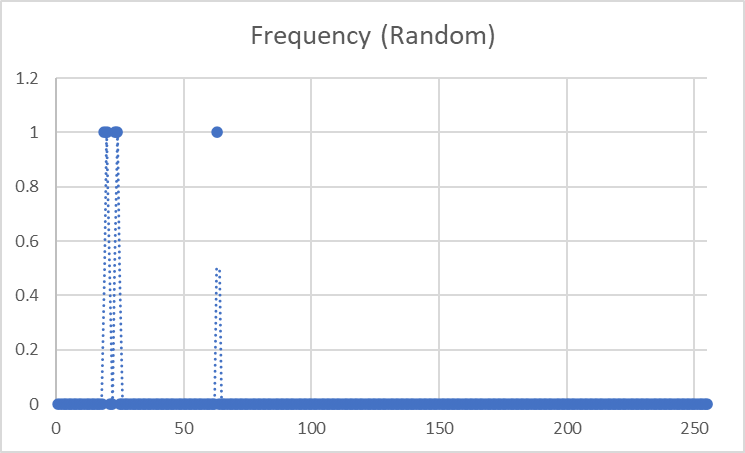
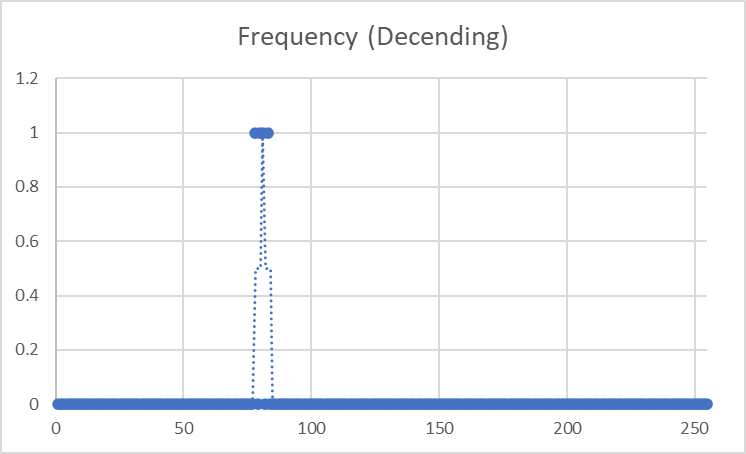
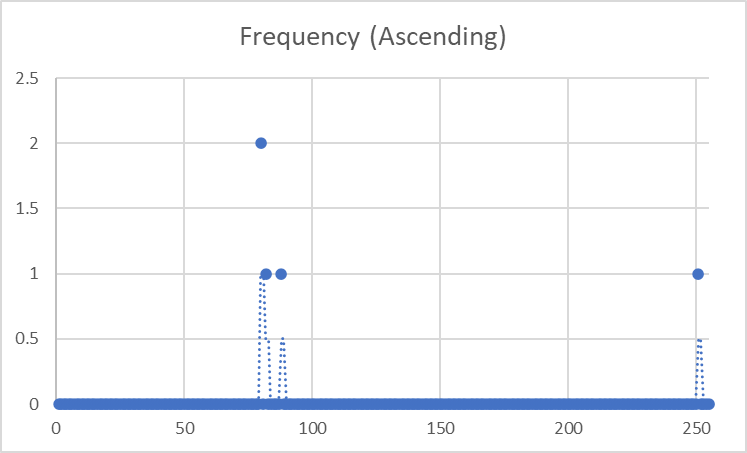
Merge Sort 10 000



Merge Sort 100 000

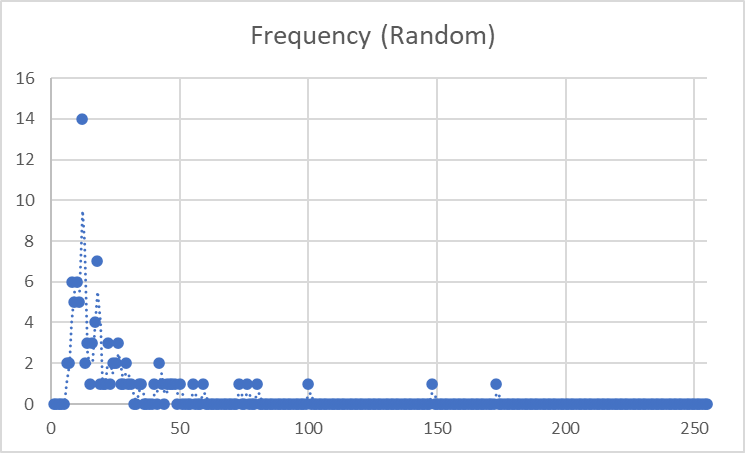
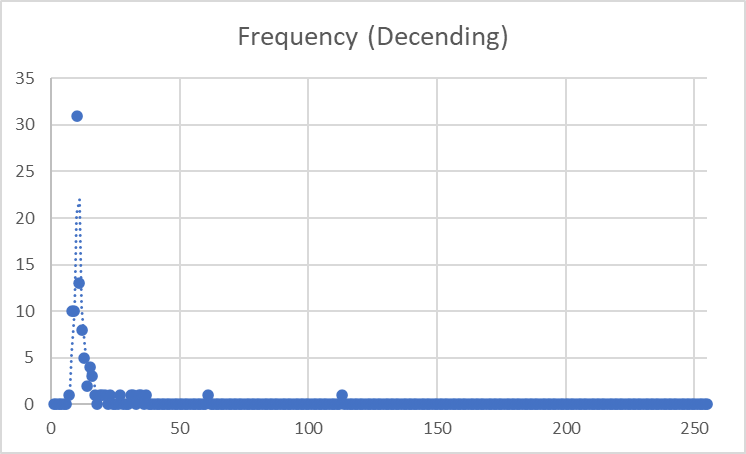
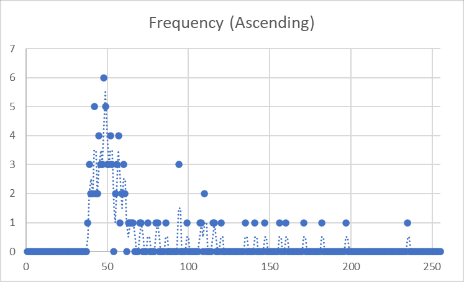


Merge Sort 1 000 000

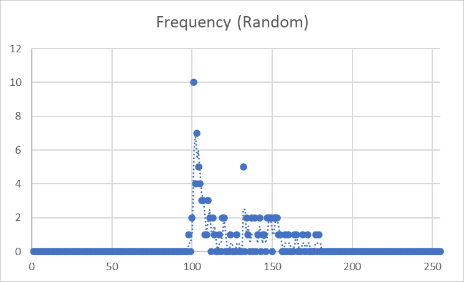
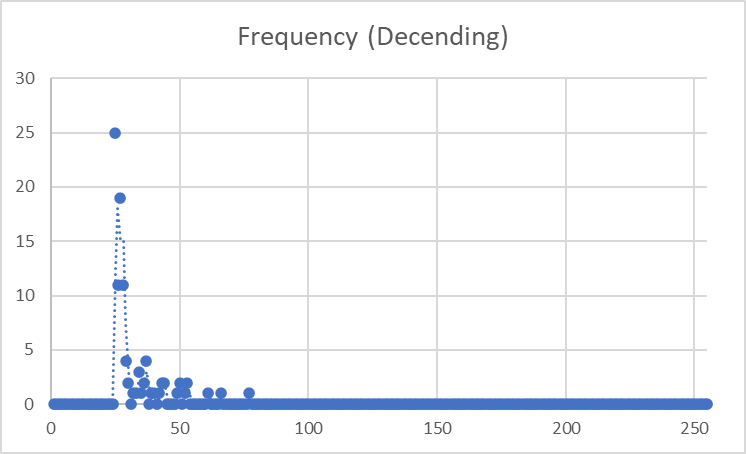
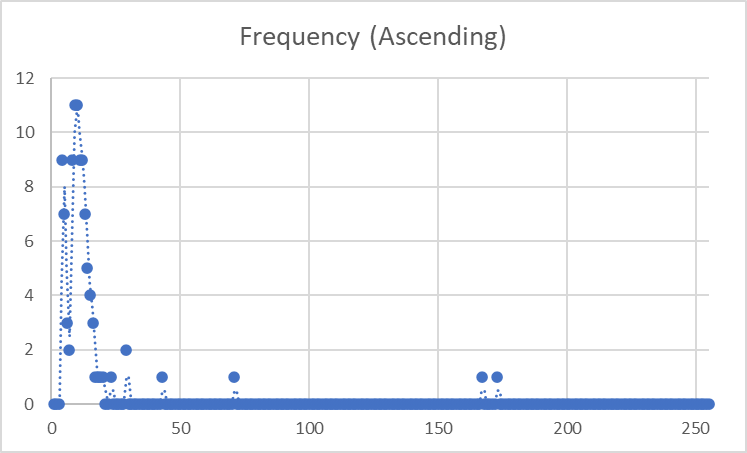


### Quick Sort

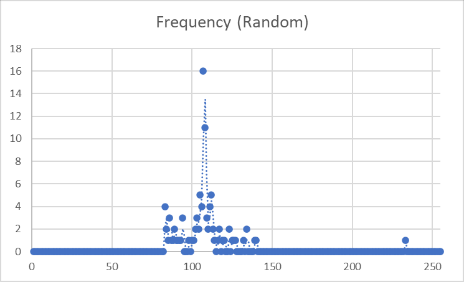
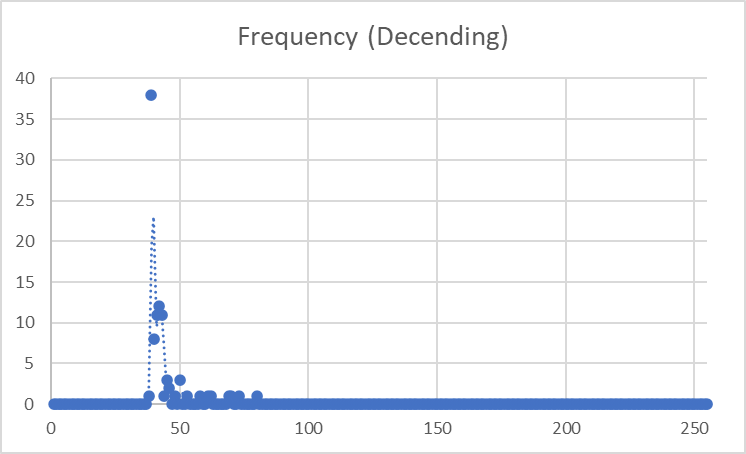
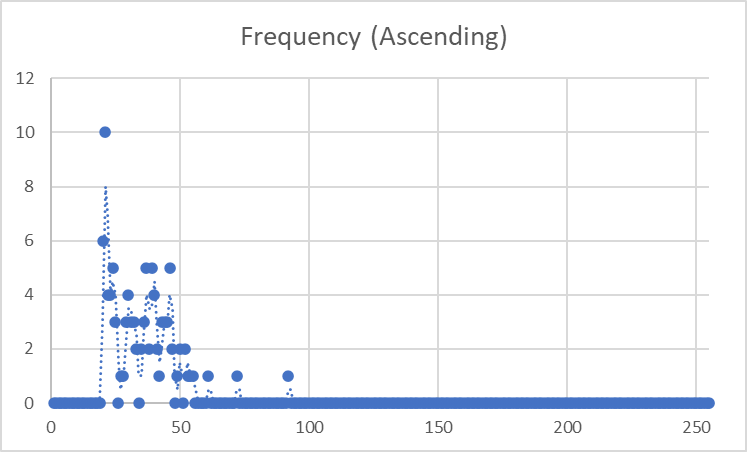
Quick Sort 10



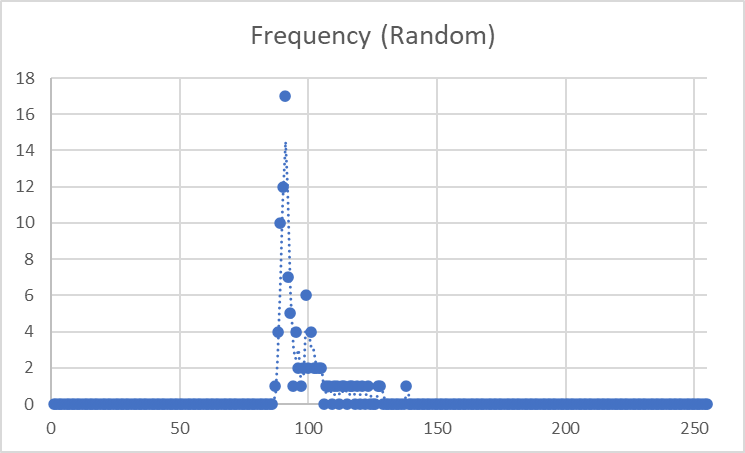
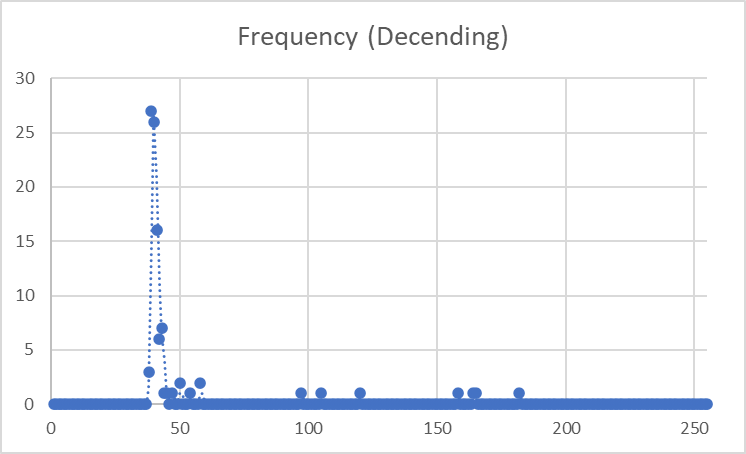
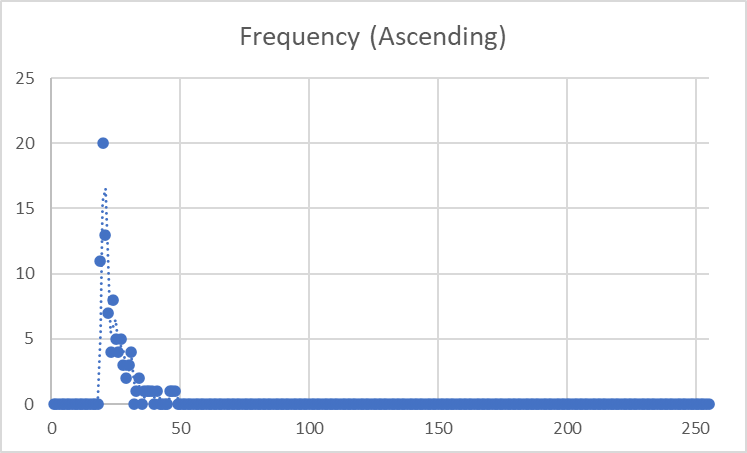
Quick Sort 100



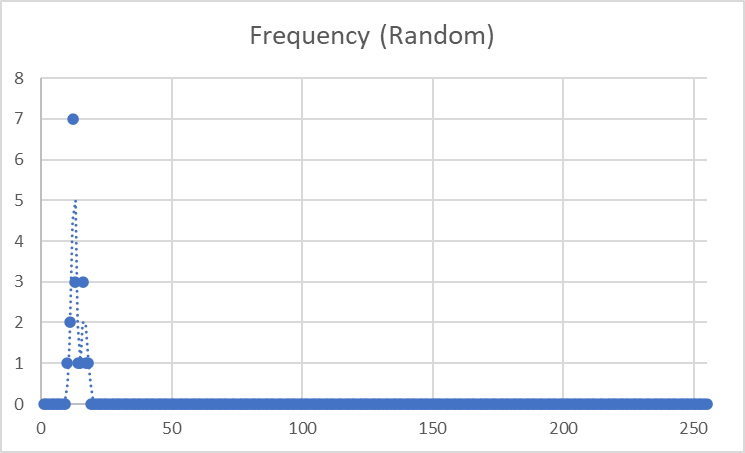
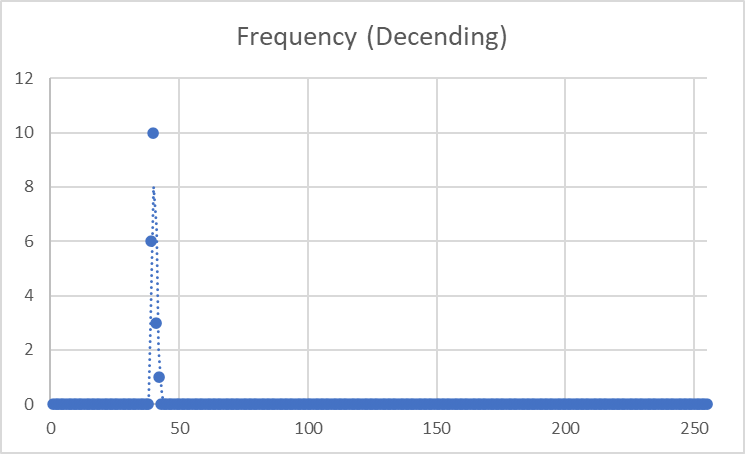
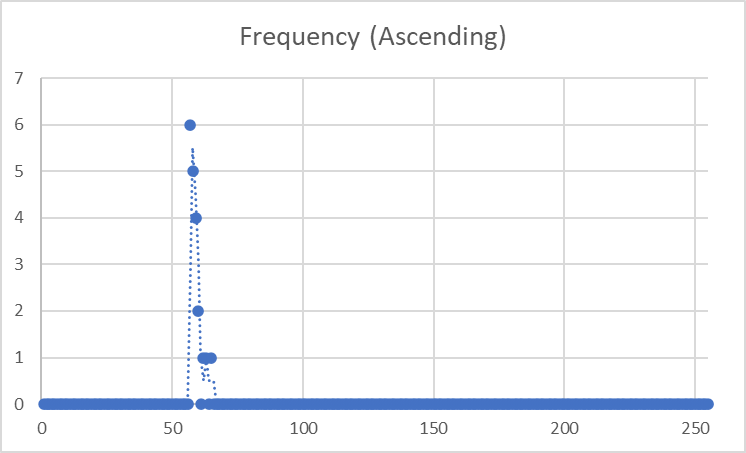
Quick Sort 1 000



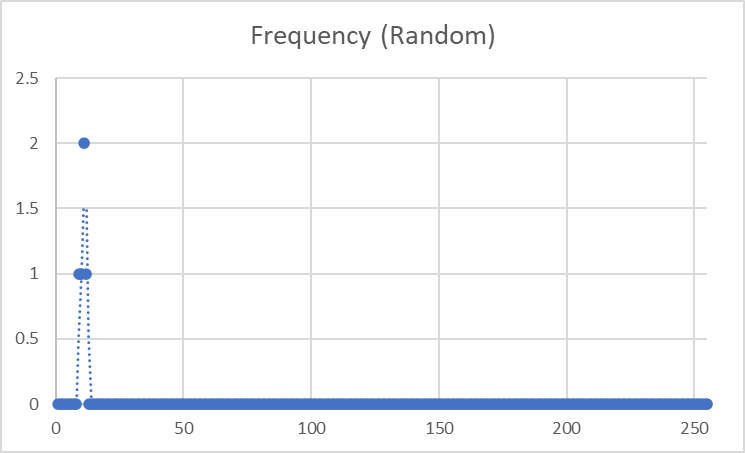
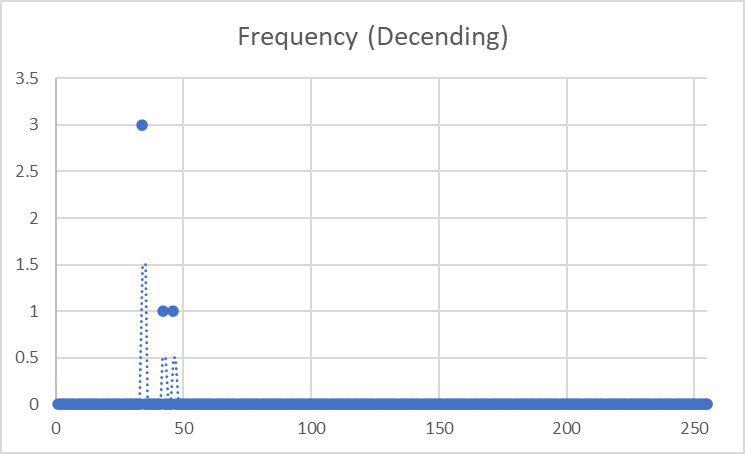
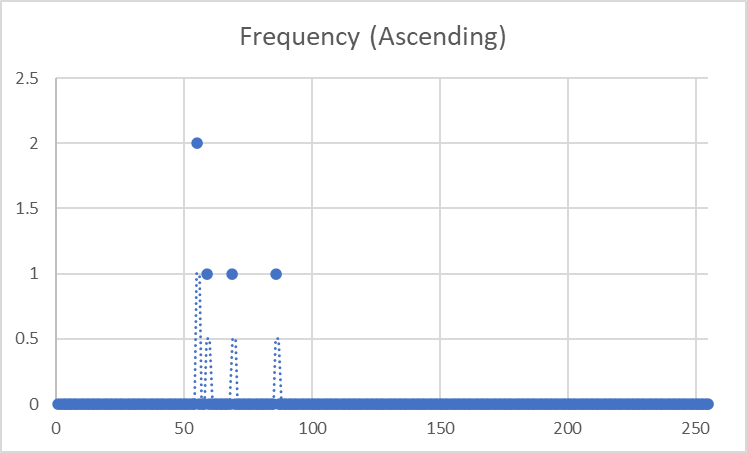
Quick Sort 10 000



Quick Sort 100 000

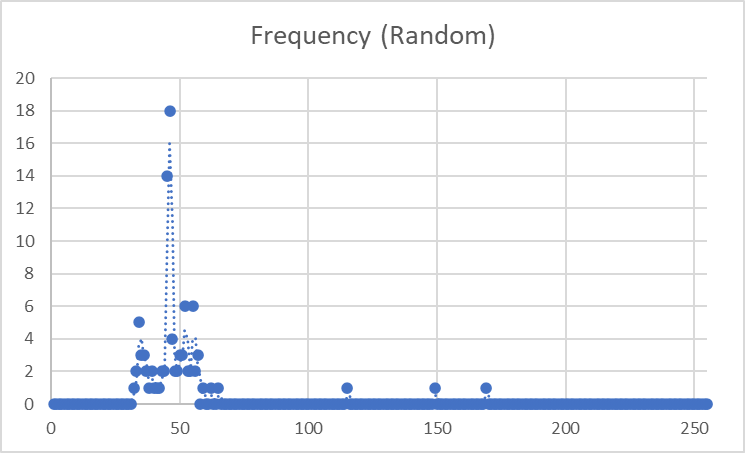
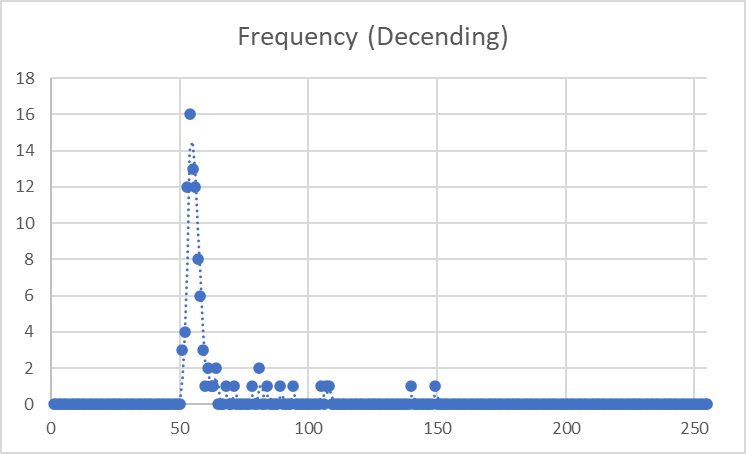
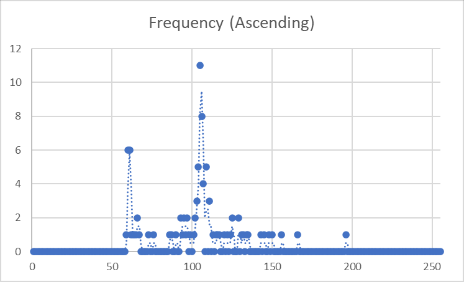


Quick Sort 1 000 000

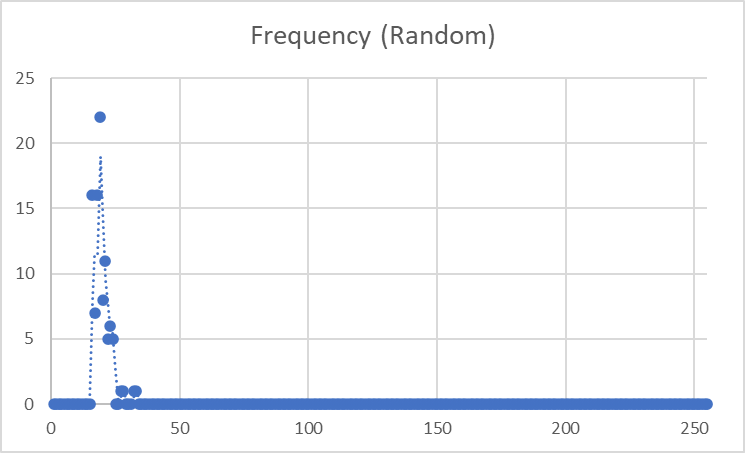
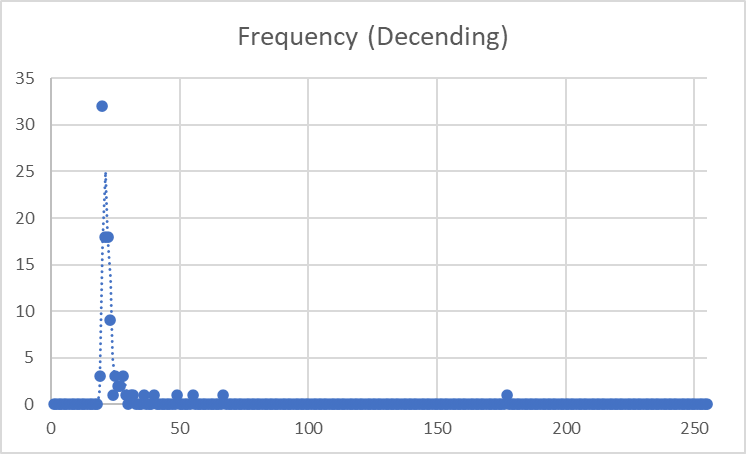
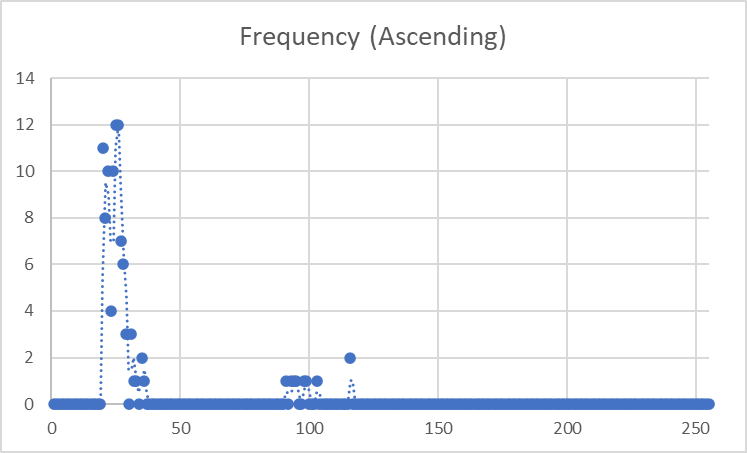


### Selection Sort

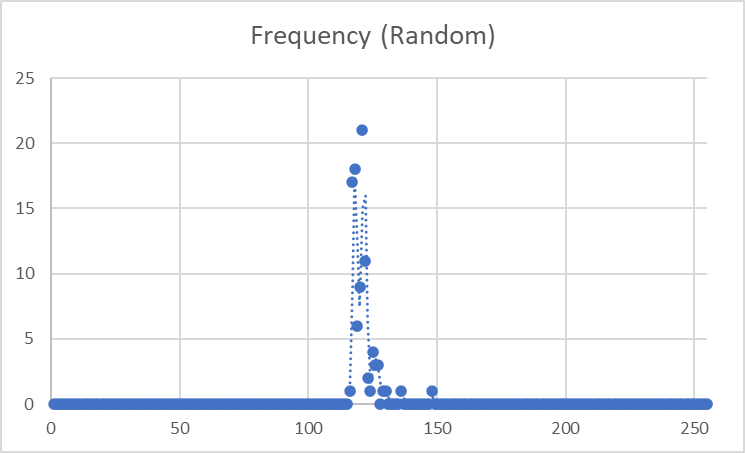
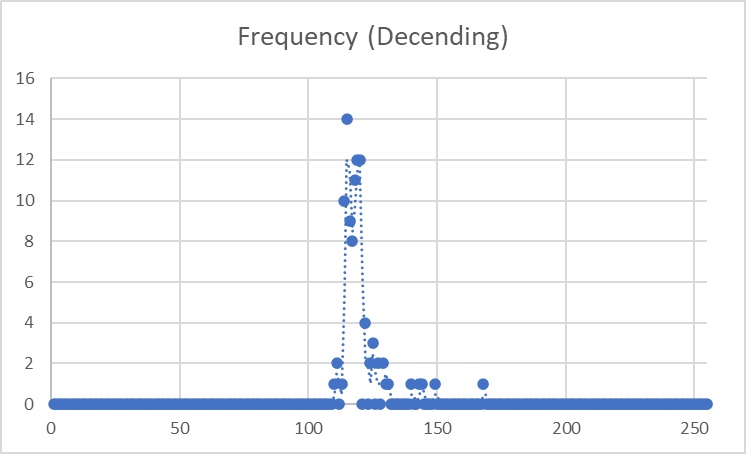
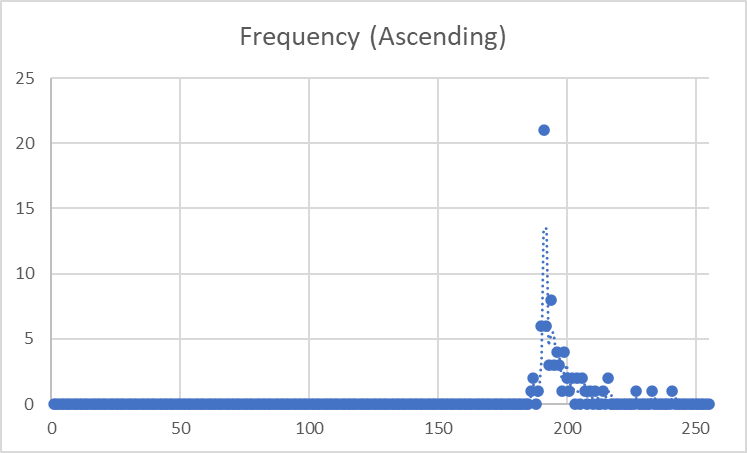
Selection Sort 10



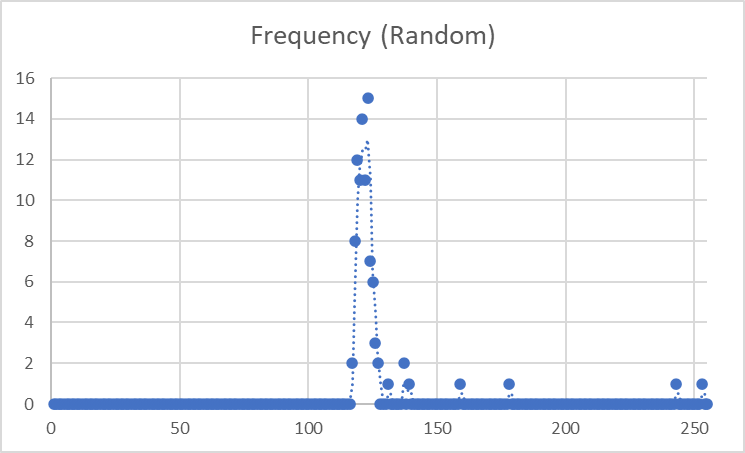
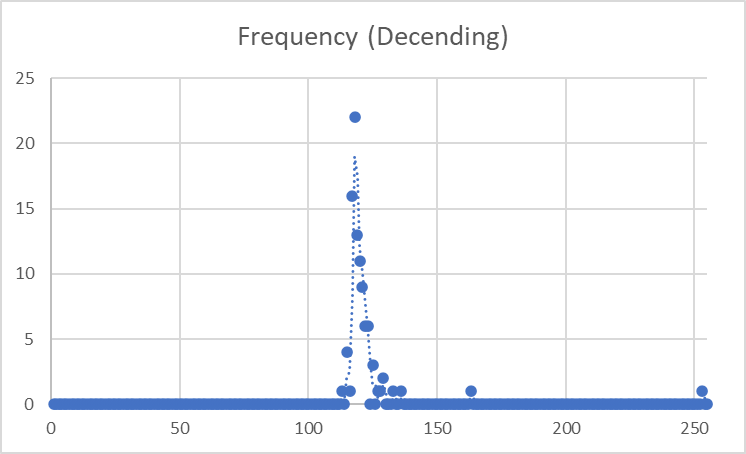
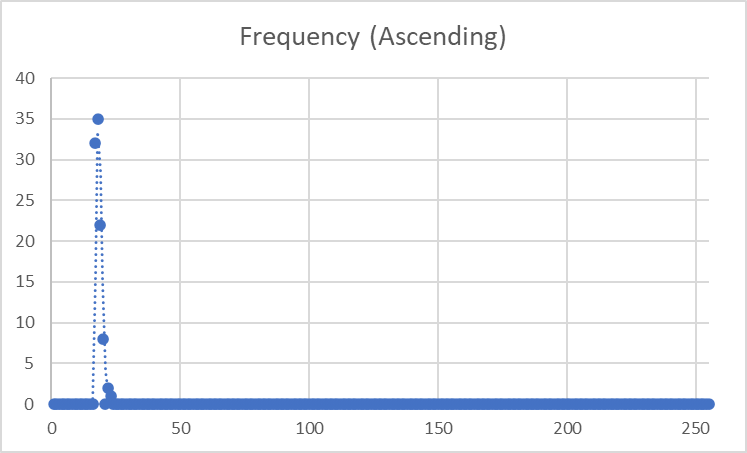
Selection Sort 100



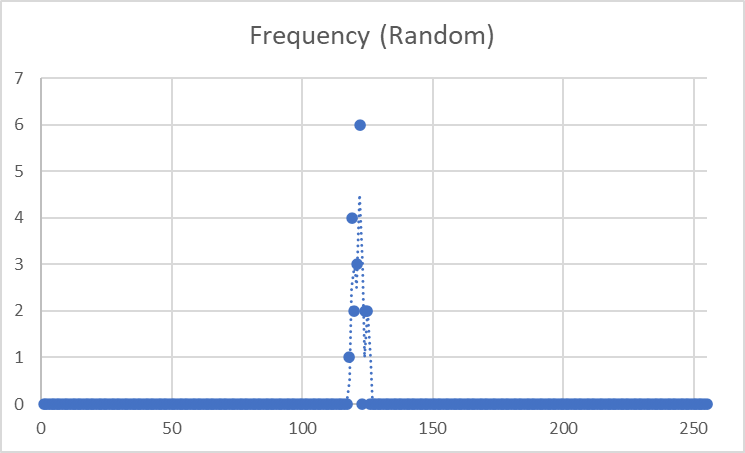
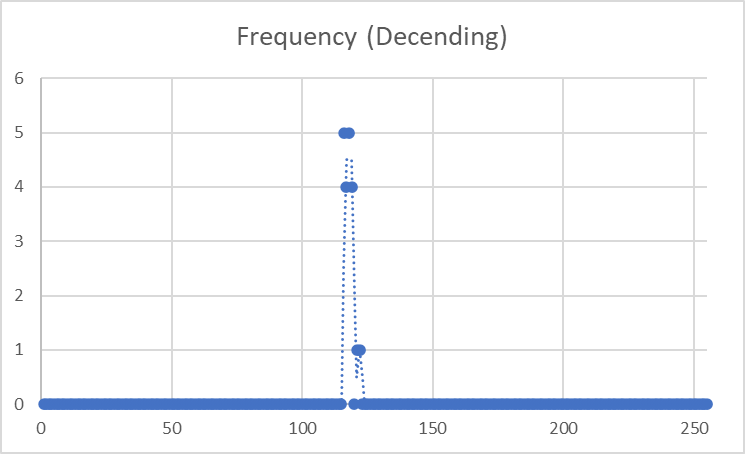
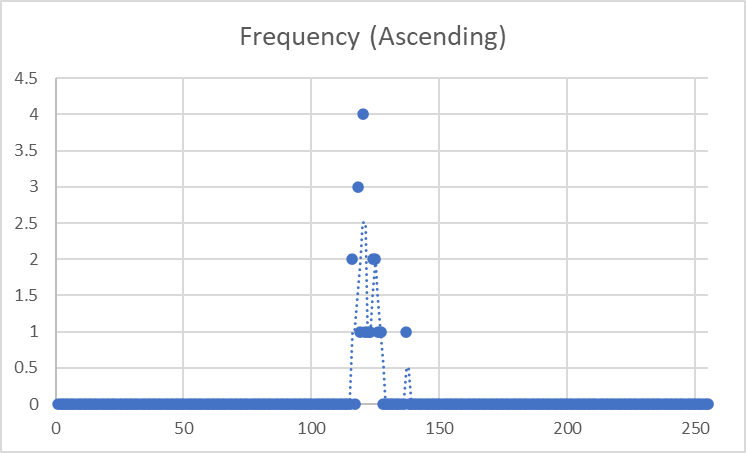
Selection Sort 1 000



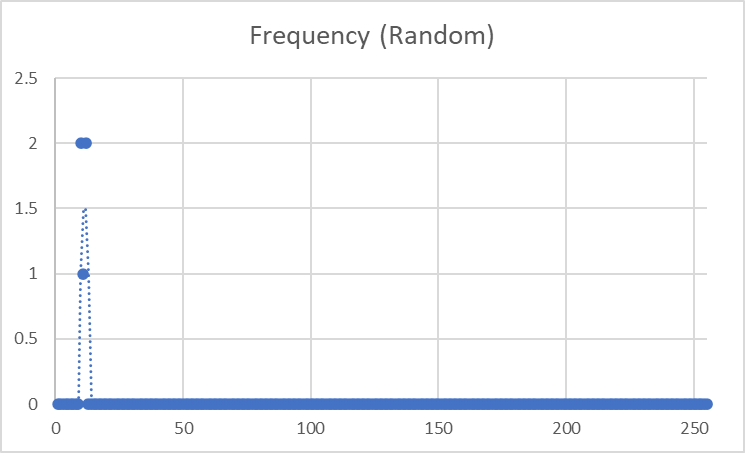
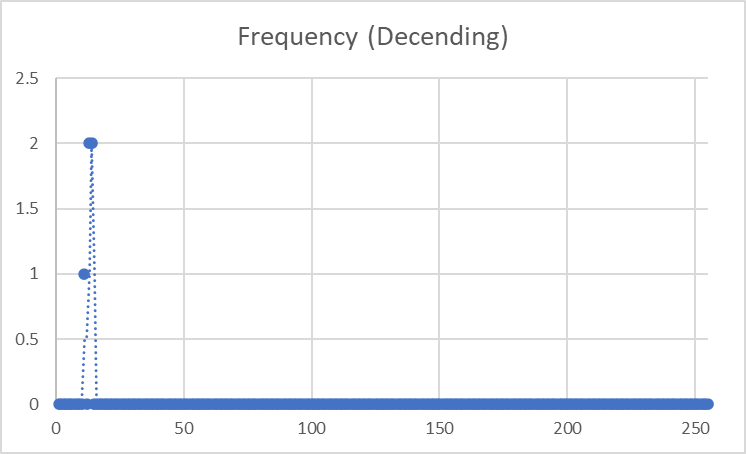
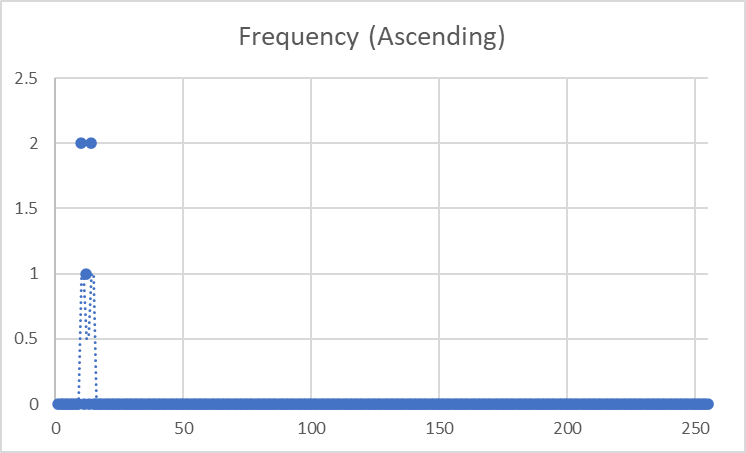
Selection Sort 10 000



Selection Sort 100 000

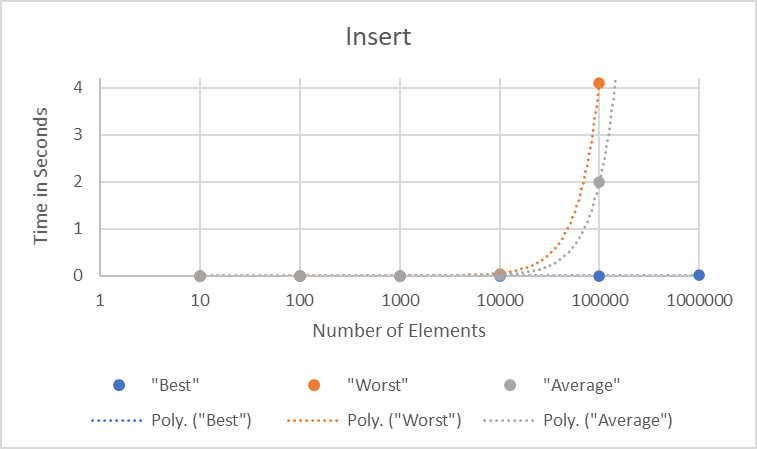


Selection Sort 1 000 000

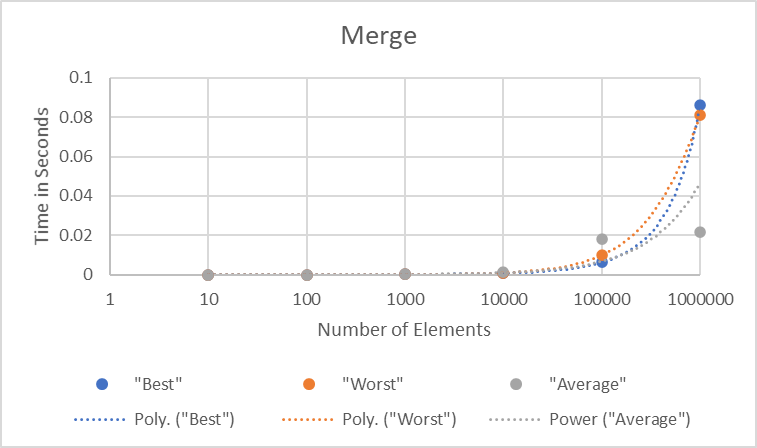


## Tables of most Frequent Run Times

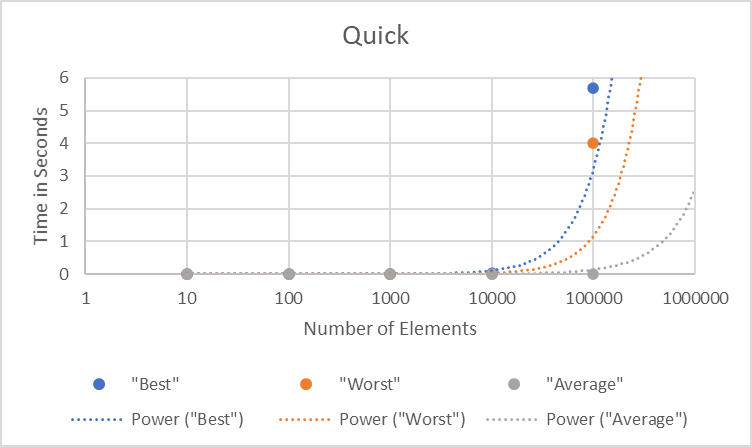
Based on the above scatter plots, times were chosen for the tables below that most accurately reflected the experiment. Below are the tables of the data sorted by the sorting algorithm. Please note the graphs often don’t look like they change until the final two values (100 000 and 1 000 000) due to 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

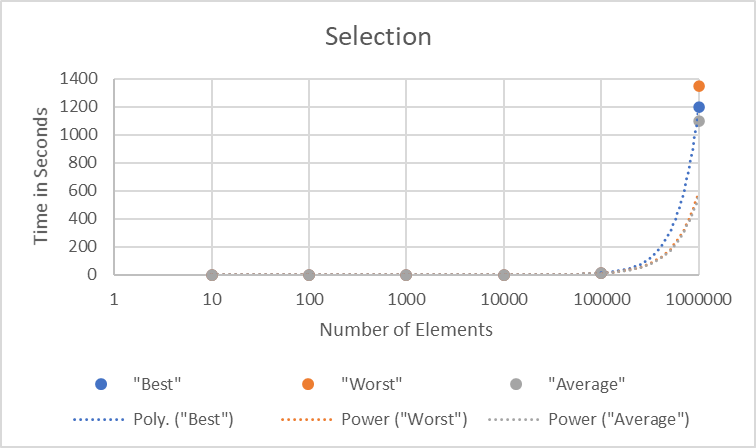
|  |  |  |  |
| --- | --- | --- | --- |
| Insert | | | |
| Elements | "Best" | "Worst" | "Average" |
| 10 | 0.0000047 | 0.000009 | 0.0000048 |
| 100 | 0.0000049 | 0.00004 | 0.0000112 |
| 1000 | 0.000015 | 0.00039 | 0.00022 |
| 10000 | 0.000025 | 0.039 | 0.021 |
| 100000 | 0.00023 | 4.1 | 2 |
| 1000000 | 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(n2) 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(n2).

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(n2) 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(n2) 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  Line 2  Line 3  Line 4  Line 5  Line 6  Line 7  Line 8  Line 9  Line 10  Line 11  Line 12  Line 13  Line 14  Line 15 | public void selectionSort(){  int n = myArray.length;    for (int i = 0; i < n-1; i++) {  int min\_idx = i;    for (int j = i+1; j < n; j++)  if (myArray[j] < myArray[min\_idx])  min\_idx = j;  int temp = myArray[min\_idx];  myArray[min\_idx] = myArray[i];  myArray[i] = temp;  }  } |

* **Best Case:** O(n2)

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(n2) due to each loop having a complexity of O(n) and the product rule.

* **Worst Case:** O(n2)

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(n2).

## Insertion Sort

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

* **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(n2)

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(n2).

## Merge Sort

|  |  |
| --- | --- |
| Line 1  Line 2  Line 3  Line 4  Line 5  Line 6  Line 7  Line 8  Line 9  Line 10  Line 11  Line 12  Line 13  Line 14  Line 15  Line 16  Line 17  Line 18  Line 19  Line 20  Line 21  Line 22  Line 23  Line 24  Line 25  Line 26  Line 27  Line 28  Line 29  Line 30  Line 31  Line 32  Line 33  Line 34  Line 35  Line 36  Line 38  Line 39 | public int[] merge(int[] arrayOne, int[] arrayTwo) {  int[] returnArray = new int[arrayOne.length + arrayTwo.length];  int arrayOneIndex = 0;  int arrayTwoIndex = 0;  for (int i = 0; i < returnArray.length; i++) {  if (arrayOneIndex < arrayOne.length && arrayTwoIndex < arrayTwo.length) {  if (arrayOne[arrayOneIndex] > arrayTwo[arrayTwoIndex]) {  returnArray[i] = arrayTwo[arrayTwoIndex];  arrayTwoIndex++;  } else {  returnArray[i] = arrayOne[arrayOneIndex];  arrayOneIndex++;  }  } else if (arrayOneIndex < arrayOne.length) {  returnArray[i] = arrayOne[arrayOneIndex];  arrayOneIndex++;  } else {  returnArray[i] = arrayTwo[arrayTwoIndex];  arrayTwoIndex++;  }  }  return returnArray;  }  public int[] mergeSortAlg(int[] sortArray){  if(sortArray.length == 1){  return sortArray;  }  int [] arrayOne = Arrays.copyOfRange(sortArray, 0, sortArray.length/2);  int [] arrayTwo = Arrays.copyOfRange(sortArray, sortArray.length/2, sortArray.length);  arrayOne = mergeSortAlg(arrayOne);  arrayTwo = mergeSortAlg(arrayTwo);  return merge(arrayOne, arrayTwo);  } |

* **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  Line 2  Line 3  Line 4  Line 5  Line 6  Line 7  Line 8  Line 9  Line 10  Line 11  Line 12  Line 13  Line 14  Line 15  Line 16  Line 18  Line 19  Line 20  Line 21  Line 22  Line 23  Line 24  Line 25  Line 26  Line 27  Line 28  Line 29  Line 30  Line 31  Line 32  Line 33  Line 34  Line 35  Line 36  Line 37  Line 38  Line 39  Line 40  Line 41  Line 42  Line 43  Line 44  Line 45 | public void quickSortAlg(int arr[], int l, int h)  {  int[] stack = new int[h - l + 1];  int top = -1;  stack[++top] = l;  stack[++top] = h;  while (top >= 0) {  h = stack[top--];  l = stack[top--];  int p = partition(arr, l, h);  if (p - 1 > l) {  stack[++top] = l;  stack[++top] = p - 1;  }  if (p + 1 < h) {  stack[++top] = p + 1;  stack[++top] = h;  }  }  }  static int partition(int arr[], int low, int high)  {  int pivot = arr[high];  int i = (low - 1);  for (int j = low; j <= high - 1; j++) {  if (arr[j] <= pivot) {  i++;  int temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }  }  int temp = arr[i + 1];  arr[i + 1] = arr[high];  arr[high] = temp;  return i + 1;  } |

* **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(n2)

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(n2).

# 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(n2). 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(n2) 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(n2). 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(n2), 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).

# 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.

# 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&amp;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/.