Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2023 Spring

# Programming Assignment 1

March 30, 2023

*Student name:*
Hüseyin Eren DOĞAN

*Student Number:*
b2210765009

# 1 Problem Definition

In this assignment we take "Traffic Flow Dataset" and tried to implement three different sorting and two searching algorithms to sort and search in dataset with "Flow Duration" feature. Our sorting algorithms are "Selection Sort", "Quick Sort" and "Bucket Sort"; our searching algorithms are "Linear Search" and "Binary Search".

# 2 Solution Implementation

Here are code implementations of sort and searching algorithms.

## 2.1 Selection Sort

```
public static void selectionSort(int[] list) {
        int n = list.length;
        for (int i = 0; i < n - 1; i++) {
            int min = i;
            for (int j = i + 1; j < n; j++) {
                if (list[j] < list[min]) {
                    min = j;
                }
            }
            if (min != i) {
                int temp = list[i];
                list[i] = list[min];
                list[min] = temp;
            }
        }
    }
```

Selection sort is a simple comparison based sorting algorithm that works by repeatedly finding the minimum element from an unsorted list and putting it at the beginning of the list. This process is repeated for each subsequent element until the entire list is sorted.

## 2.2 Quick Sort

```
17 public static void quickSort(int[] array, int low, int high) {
18         int stackSize = high - low + 1;
19         int[] stack = new int[stackSize];
20         int top = -1;
21 
22         stack[++top] = low;
23         stack[++top] = high;
24 
25         while (top >= 0) {
26             high = stack[top--];
27             low = stack[top--];
28             int pivot = partition(array, low, high);
29             if (pivot - 1 > low) {
30                 stack[++top] = low;
31                 stack[++top] = pivot - 1;
32             }
33             if (pivot + 1 < high) {
34                 stack[++top] = pivot + 1;
35                 stack[++top] = high;
36             }
37         }
38     }
39 
40     public static int partition(int[] array, int low, int high) {
41         int pivot = array[high];
42         int i = low - 1;
43         for (int j = low; j <= high - 1; j++) {
44             if (array[j] <= pivot) {
45                 i++;
46                 int temp = array[i];
47                 array[i] = array[j];
48                 array[j] = temp;
49             }
50         }
51         int temp = array[i + 1];
52         array[i + 1] = array[high];
53         array[high] = temp;
54         return i + 1;
55     }
```

Quick sort is a popular comparison based sorting algorithm that uses a divide-and-conquer approach to sort an array. It works by selecting a pivot element from the array, and then partitioning the array into two sub-arrays, one containing elements smaller than the pivot, and the other containing elements larger than the pivot. The process is then repeated recursively on each sub-array until the entire array is sorted.

### 2.3 Bucket Sort

```
56 public static void bucketSort(int[] A, int n) {
57         int numberOfBuckets = (int) Math.ceil(Math.sqrt(n));
58         List<Integer>[] buckets = new List[numberOfBuckets];
59         for (int i = 0; i < numberOfBuckets; i++) {
60             buckets[i] = new ArrayList<Integer>();
61         }
62         int max = A[0];
63         for (int i = 1; i < n; i++) {
64             if (A[i] > max) {
65                 max = A[i];
66             }
67         }
68         for (int i = 0; i < n; i++) {
69             int index = hash(A[i], max, numberOfBuckets);
70             buckets[index].add(A[i]);
71         }
72         for (int i = 0; i < numberOfBuckets; i++) {
73             Collections.sort(buckets[i]);
74         }
75         int index = 0;
76         for (int i = 0; i < numberOfBuckets; i++) {
77             for (int j = 0; j < buckets[i].size(); j++) {
78                 A[index++] = buckets[i].get(j);
79             }
80         }
81     }
82 
83     public static int hash(int i, int max, int numberOfBuckets) {
84         return (int) Math.floor((double) i / max * (numberOfBuckets - 1));
85     }
```

Bucket sort is a non-comparison based sorting algorithm that divides an input array into a number of smaller "buckets" or "bins", and then sorts each bucket individually using another sorting algorithm (usually insertion sort). Bucket sort works particularly well when the input values are uniformly distributed over a range.

### 2.4 Linear Search

```
86  public static int linearSearch(int[] A, int x) {
87          int size = A.length;
88          for (int i = 0; i < size - 1; i++) {
89              if (A[i] == x) {
90                  return i;
91              }
92          }
93          return -1;
94      }
```

Linear search is a sequential searching algorithm where we start from one end and check every element of the list until the desired element is found. It is the simplest searching algorithm and works by sequentially checking each element in the list or array until the target value is found, or until the end of the list is reached.

### 2.5 Binary Search

```
95  public static int binarySearch(int[] A, int x) {
96          int low = 0;
97          int high = A.length - 1;
98          while (high - low > 1) {
99              int mid = (high + low) / 2;
100             if (A[mid] < x) {
101                 low = mid + 1;
102             } else {
103                 high = mid;
104             }
105         }
106         if (A[low] == x) {
107             return low;
108         } else if (A[high] == x) {
109             return high;
110         } else {
111             return -1;
112         }
113     }
```

Binary search is an algorithm used to find a particular value in a sorted list or array. It works by repeatedly dividing the search interval in half until the target value is found, or until it is determined that the target value is not in the array. It is very efficient for large data sets and is widely used in many programming applications. However, binary search only works on sorted lists or arrays and requires extra time to sort the input data beforehand.

## 3 Results, Analysis, Discussion

As a result of the experiment, we can observe that each algorithm has its own advantages and disadvantages on different input types and sizes.

For selection sort, we can say that it is the worst algorithm on running time when we're comparing it two other algorithms we used in this experiment. It takes too much time, especially when input size is too large, because it goes along the list until it finds the minimum element of the list and repeats this until the list is all sorted.

Quick sort algorithm is useful on random and reverse sorted data, but we can say that from the results it is worse than selection sort on sorted data. You can see it on Figure 3

Bucket sort algorithm has the best time complexity in this three sorting algorithms according to the results of experiment, especially in large input sizes.

Running time test results for sorting algorithms are given in Table 1.

Table 1: Results of the running time tests performed for varying input sizes (in ms).

| | **Input Size $n$** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Algorithm** | **500** | **1000** | **2000** | **4000** | **8000** | **16000** | **32000** | **64000** | **128000** | **250000** |
| | **Random Input Data Timing Results in ms** | | | | | | | | | |
| Selection sort | 0 | 0 | 0 | 3 | 12 | 53 | 206 | 810 | 3252 | 12434 |
| Quick sort | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 9 | 28 | 46 |
| Bucket sort | 0 | 0 | 0 | 1 | 1 | 3 | 3 | 6 | 15 | 31 |
| | **Sorted Input Data Timing Results in ms** | | | | | | | | | |
| Selection sort | 0 | 1 | 1 | 3 | 12 | 51 | 201 | 804 | 3251 | 12430 |
| Quick sort | 0 | 1 | 1 | 6 | 21 | 81 | 318 | 1270 | 5090 | 19428 |
| Bucket sort | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 4 |
| | **Reversely Sorted Input Data Timing Results in ms** | | | | | | | | | |
| Selection sort | 0 | 0 | 1 | 4 | 17 | 73 | 298 | 1193 | 4943 | 21868 |
| Quick sort | 0 | 0 | 0 | 1 | 7 | 12 | 42 | 182 | 699 | 1481 |
| Bucket sort | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 3 | 7 |

And our searching algorithms... We can say that linear search may take too much time on random data when input sizes are increasing because it checks every element from the beginning until finds it. Figure 5

And on a sorted data, this results show us linear search is slightly better than binary search on running time. But when input size increases, binary search's time decreases while linear search's increases too. So on large input sizes it may be better using binary search. Figure 6

Running time test results for search algorithms are given in Table 2.

Table 2: Results of the running time tests of search algorithms of varying sizes (in ns).

| | **Input Size $n$** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Algorithm** | **500** | **1000** | **2000** | **4000** | **8000** | **16000** | **32000** | **64000** | **128000** | **250000** |
| Linear search (random data) | 1862 | 292 | 1547 | 496 | 196 | 650 | 1996 | 5737 | 519 | 14956 |
| Linear search (sorted data) | 4 | 3 | 39 | 3 | 1 | 6 | 20 | 56 | 5 | 162 |
| Binary search (sorted data) | 362 | 132 | 183 | 148 | 133 | 176 | 203 | 86 | 131 | 65 |

Complexity analysis tables to complete (Table 3 and Table 4):

Table 3: Computational complexity comparison of the given algorithms.

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
|---|---|---|---|
| Selection Sort | $\Omega(n^2)$ | $\Theta(n^2)$ | $O(n^2)$ |
| Quick Sort | $\Omega(n \log n)$ | $\Theta(n \log n)$ | $O(n^2)$ |
| Bucket Sort | $\Omega(n + k)$ | $\Theta(n + k)$ | $O(n^2)$ |
| Linear Search | $\Omega(1)$ | $\Theta(n)$ | $O(n)$ |
| Binary Search | $\Omega(1)$ | $\Theta(n \log n)$ | $O(n \log n)$ |

Table 4: Auxiliary space complexity of the given algorithms.

| **Algorithm** | **Auxiliary Space Complexity** |
|---|---|
| Selection Sort | $O(1)$ |
| Quick Sort | $O(n)$ |
| Bucket Sort | $O(n + k)$ |
| Linear Search | $O(1)$ |
| Binary Search | $O(1)$ |

# References


- https://www.geeksforgeeks.org/selection-sort/
- https://www.geeksforgeeks.org/quick-sort/
- https://www.javatpoint.com/bucket-sort
- https://www.programiz.com/dsa/linear-search
- https://www.programiz.com/dsa/binary-search

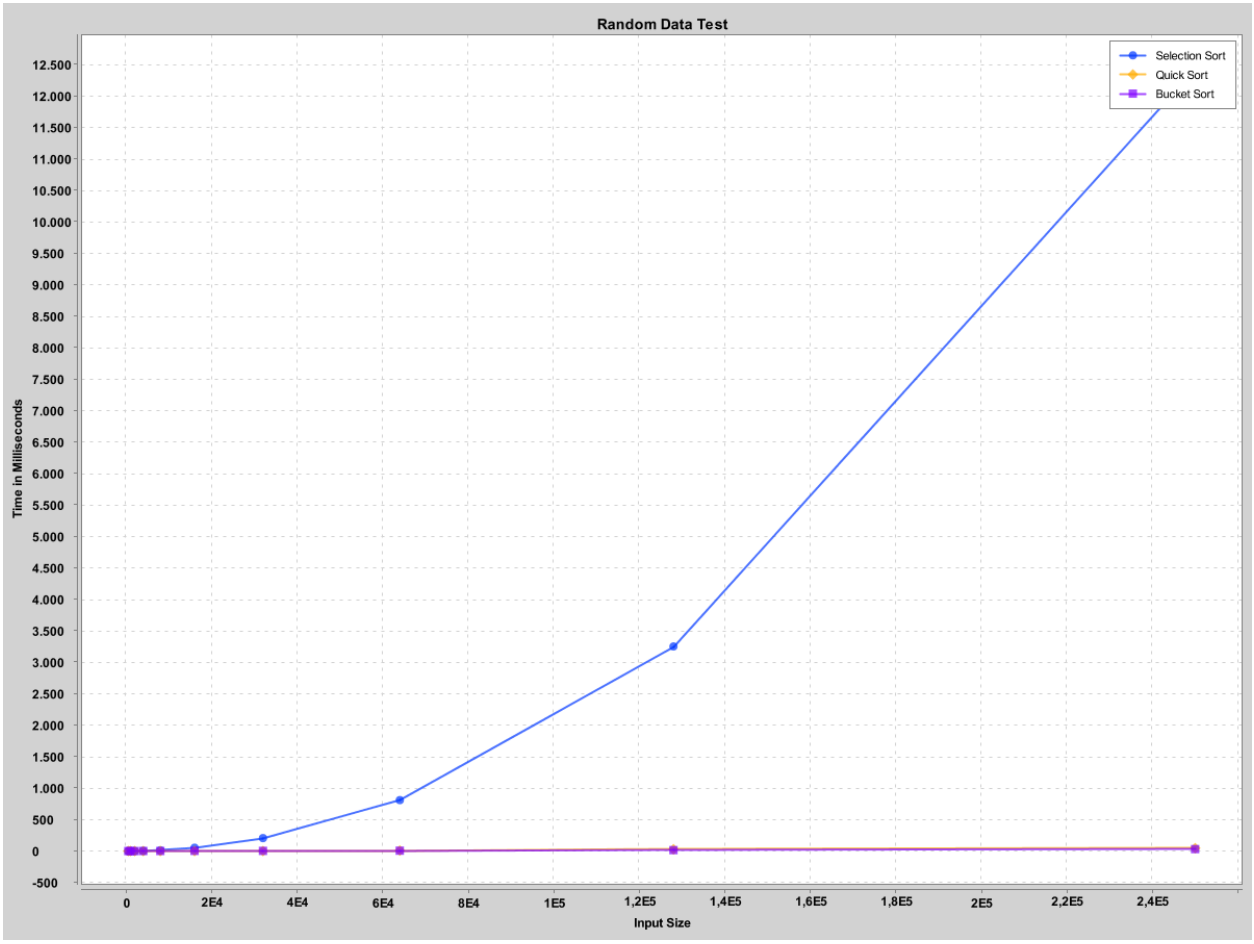


Figure 1: Testing our 3 sorting algorithms on random data.

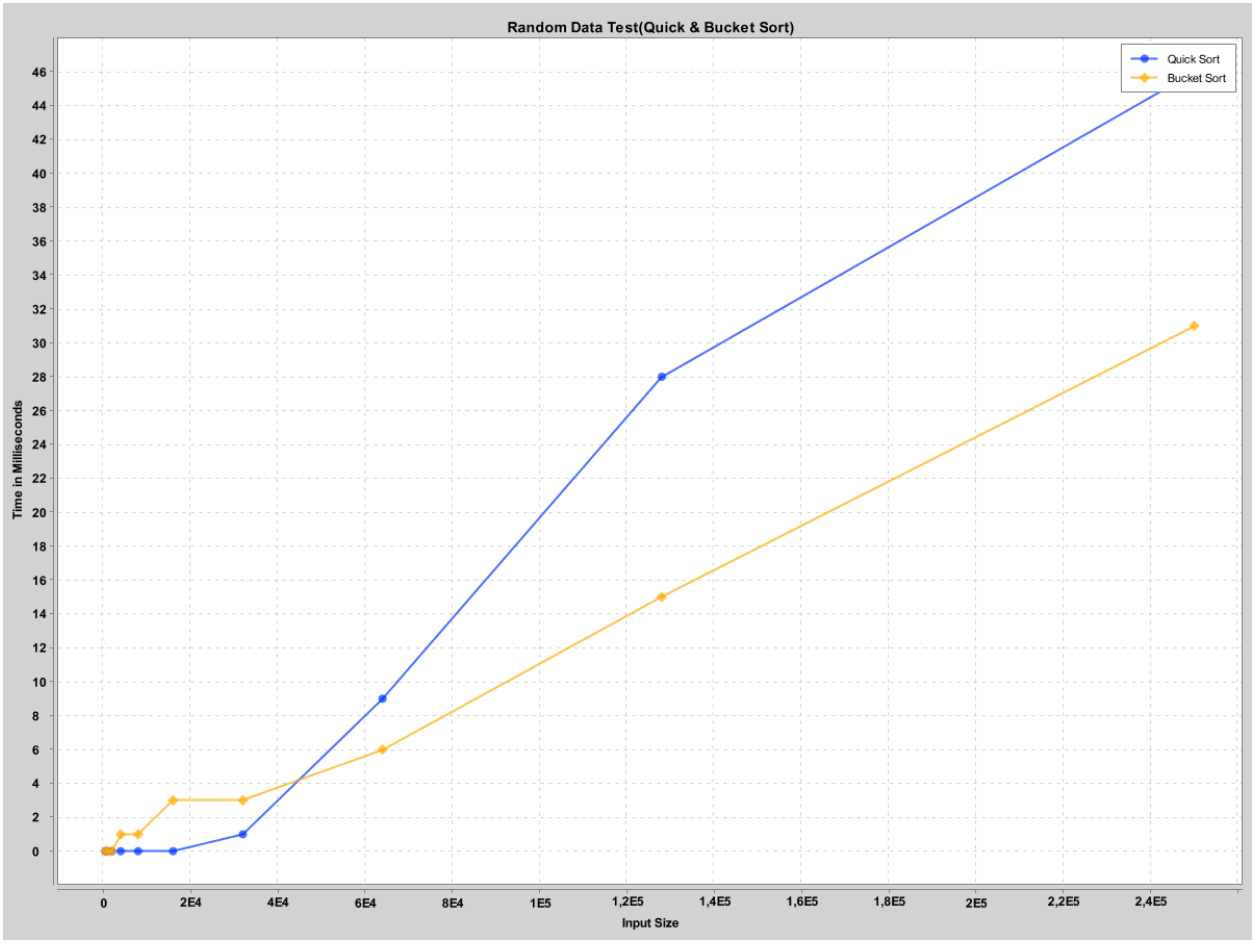


Figure 2: Another plot of testing random data because Quicksort and Bucketsort was overlapping in the first one.

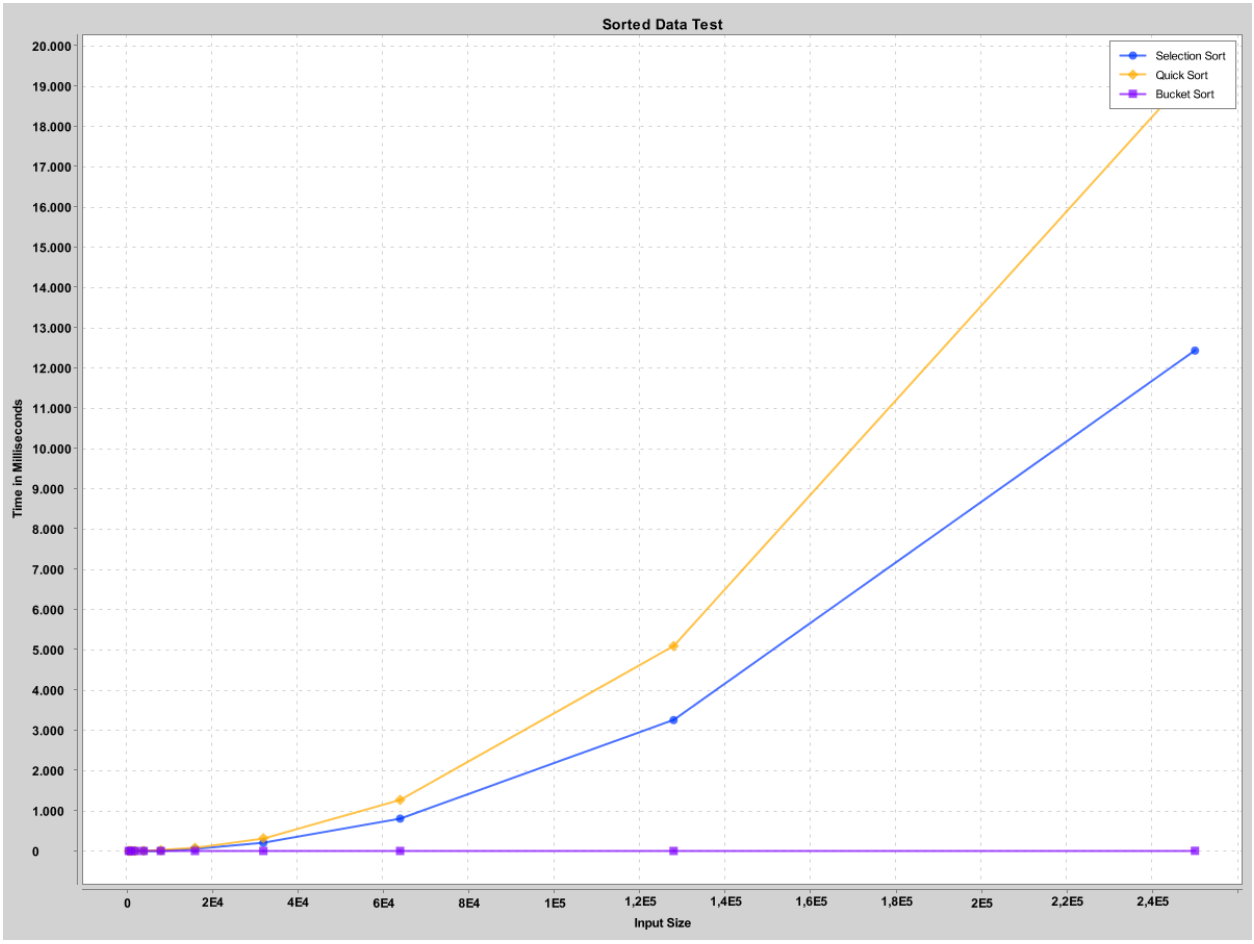


Figure 3: Testing on sorted data.

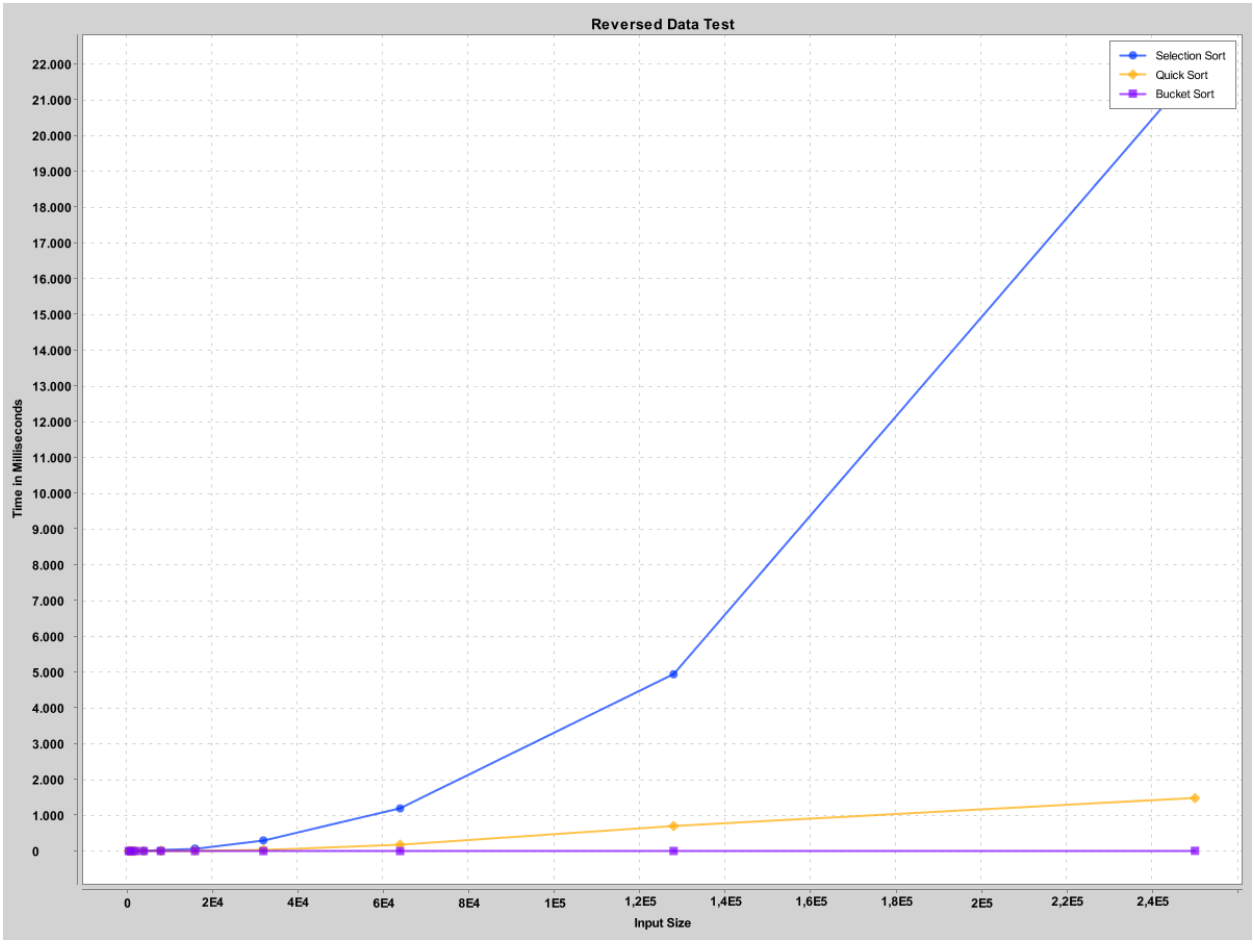


Figure 4: Testing on reverse sorted data.

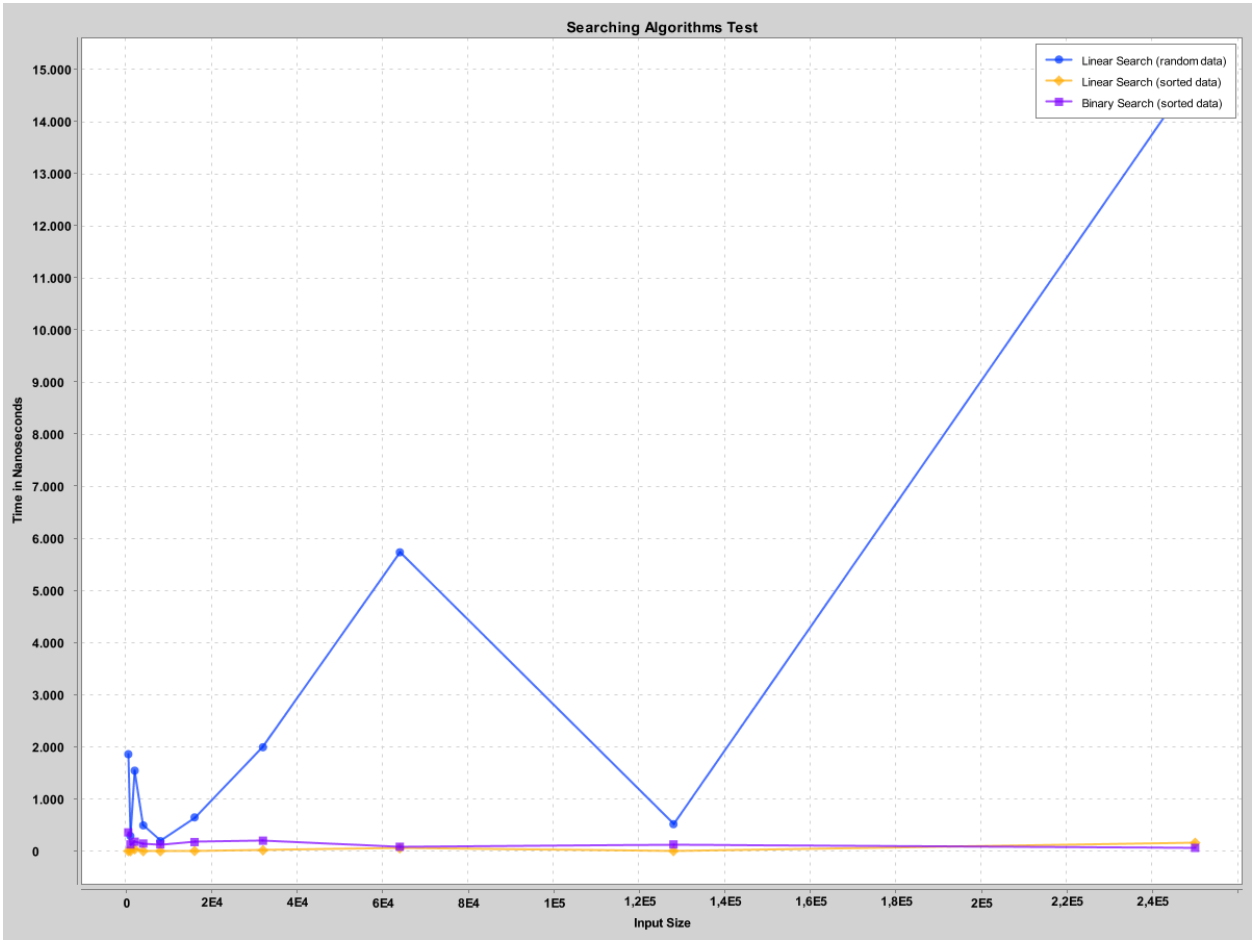


Figure 5: Testing our searching algorithms.

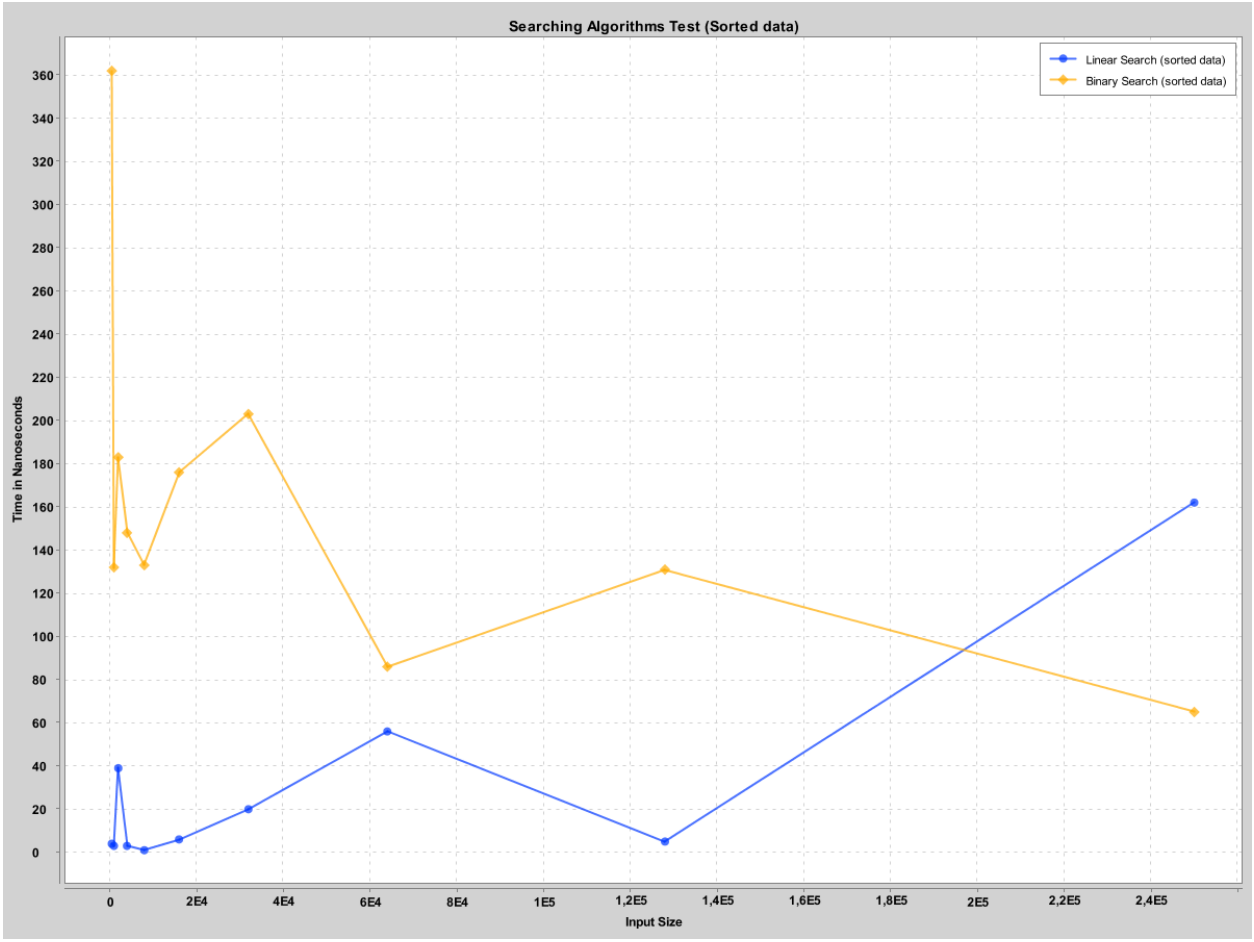


Figure 6: Testing our searching algorithms on sorted data because of overlapping in first graph again.