Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
Kağan Canerik

Student Number:
b2210765021

1 Problem Definition

The aim of this project is to analyze the efficiency of some algorithms based on their time and space complexity according to the various number of inputs. At the end of the project, students are expected to have developed their algorithmic thinking skills through comparative analysis and expanded their understanding of evaluating algorithms in the field of computer science.

2 Solution Implementation

2.1 Merge Sort

```java
import java.util.Arrays;
public class MergeSorting {
    public static int[] sort(int[] array){
        int n = array.length;
        if (n <= 1) {
            return array;
        }
        int[] left = Arrays.copyOfRange(array, 0, n / 2);
        int[] right = Arrays.copyOfRange(array, n / 2, n);
        sort(left);
        sort(right);
        return merge(left, right);
    }
    public static int[] merge(int[] left, int[] right){
        int[] mergedArray = new int[left.length + right.length];
        int leftIndex = 0, rightIndex = 0, mergedIndex = 0;
        while (leftIndex < left.length && rightIndex < right.length) {
            if (left[leftIndex] <= right[rightIndex]) {
                mergedArray[mergedIndex++] = left[leftIndex++];
            } else {
                mergedArray[mergedIndex++] = right[rightIndex++];
            }
        }
        while (leftIndex < left.length) {
            mergedArray[mergedIndex++] = left[leftIndex++];
        }
        while (rightIndex < right.length) {
            mergedArray[mergedIndex++] = right[rightIndex++];
        }
        return mergedArray;
    }
}
```

This Java code implements the **Merge Sort** algorithm. Merge Sort is a divide-and-conquer sorting algorithm that operates by recursively dividing the array into smaller subarrays until each

subarray contains only one element. Then, it merges adjacent subarrays in a sorted manner, combining them into larger sorted subarrays until the entire array is sorted. It guarantees a time complexity of $O(n \log n)$, making it efficient for sorting large datasets. However, it requires additional space proportional to the size of the input array for the merging process. The **sort** method splits the array into two halves and recursively calls itself to sort each half. Then, the **merge** method merges the two sorted halves.

2.2 Insertion Sort

```
35 public class InsertionSorting {
36     public static void sort(int[] array) {
37         for (int i = 1; i < array.length; i++) {
38             int key = array[i];
39             int j = i - 1;
40             while (j >= 0 && array[j] > key) {
41                 array[j + 1] = array[j];
42                 j = j - 1;
43             }
44             array[j + 1] = key;
45         }
46     }
47 }
```

This Java code implements the **Insertion Sort** algorithm.Insertion Sort is a comparison-based sorting algorithm that iterates through an array, gradually building up a sorted portion of the array by inserting each unsorted element into its correct position.

2.3 Counting Sort

```
50 import java.util.Arrays;
51 public class CountingSorting {
52     public static int[] sort(int[] array, int k){
53         int[] count = new int[k+1];
54         Arrays.fill(count, 0);
55         int[] output = new int[array.length];
56         int size = array.length;
57 
58         for(int i = 0; i < size; i++) {
59             int j = array[i];
60             count[j] = count[j] + 1;
61         }
62 
63         for(int i = 1; i < k + 1; i++){
64             count[i] = count[i] + count[i-1];
65         }
66 
```

```
        for(int i = size - 1; i >= 0 ; i--){
            int j = array[i];
            count[j] = count[j] - 1;
            output[count[j]] = array[i];
        }

        return output;
    }
}
```

This Java code demonstrates the **Counting Sort** algorithm, which is effective when the input values have a known range. It works by counting the frequency of each element, then reconstructing a sorted array based on these counts. It achieves linear time complexity $O(n + k)$, where n is the number of elements and k is the range of input values. While efficient for small ranges, it requires additional space proportional to the range, making it less practical for larger ranges or sparse datasets. The algorithm first finds the maximum value, counts element occurrences, and computes the correct position for each element to produce the sorted array as output.

2.4 Linear Search

```
public class LinearSearching {
    public static int search(int[] array, int x){
        int size = array.length;
        for(int i=0; i<size-1;i++){
            if(array[i] == x){
                return i;
            }
        }
        return -1;
    }
}
```

This Java code implements **Linear Search** algorithm. It sequentially checks each element in the data structure until it finds the desired element or reaches the end of the structure. It has a time complexity of $O(n)$, where n is the number of elements in the array. While it is straightforward and easy to implement, it is not efficient for large datasets or when the element being searched for is located towards the end of the array. However, it is useful for unsorted arrays or when a sorted array is not available.

2.5 Binary Search

```
public class BinarySearching {
    public static int search(int[] array, int x){
        int low = 0;
        int high = array.length - 1;

```

```
93          while (low <= high) {
94              int mid = low + (high - low) / 2;
95
96              if (array[mid] == x) {
97                  return mid;
98              }
99              else if (array[mid] < x) {
100                 low = mid + 1;
101             }
102             else {
103                 high = mid - 1;
104             }
105         }
106
107         return -1;
108     }
109 }
```

This Java code impelents **Binary Search** algorithm. It works by repeatedly dividing the search interval in half until the target element is found or the interval is empty. It begins by comparing the target element with the middle element of the array. If they match, the search is complete. If the target element is less than the middle element, the search continues in the lower half of the array; otherwise, it continues in the upper half. This process halves the search space with each iteration, resulting in a time complexity of $O(\log n)$, where n is the number of elements in the array. Binary Search is significantly faster than linear search for large datasets, but it requires the array to be sorted beforehand.

3 Results, Analysis, Discussion

Your explanations, results, plots go in this section...

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

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

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

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									
Insertion sort	0.5	0.2	0.3	1.4	6.6	25.6	101.1	406.6	1637.5	6291.7
Merge sort	0.5	0.0	0.1	0.2	0.8	0.9	1.6	3.6	10.4	24.2
Counting sort	145.0	133.3	133.2	133.6	133.7	133.2	134.1	134.5	139.7	140.3
	Sorted Input Data Timing Results in ms									
Insertion sort	0.1	0.0	0.0	0.0	0.0	0.1	0.0	0.1	0.2	0.2
Merge sort	2.2	0.1	0.1	0.2	0.4	0.9	1.8	3.6	8.8	19.4
Counting sort	141.8	133.3	133.7	134.1	134.4	133.8	134.3	135.1	136.1	136.2
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.1	0.1	0.4	1.3	5.1	20.4	82.0	328.5	1266.7
Merge sort	0.1	0.6	0.1	0.3	0.5	0.8	1.7	3.9	13.9	26.1
Counting sort	135.4	133.6	134.7	133.7	134.0	134.2	133.9	139.4	135.5	137.4

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)	4016.704	2554.159	208.478	367.459	648.074	1031.458	2108.194	4615.671	8723.996	15172.824
Linear search (sorted data)	1042.204	155.751	200.283	383.007	632.031	1080.164	2076.121	4366.333	8235.284	14807.634
Binary search (sorted data)	1059.925	93.672	100.672	83.037	66.87	66.497	74.629	121.921	79.132	178.583

Table 3: Computational complexity comparison of the given algorithms.

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Insertion sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Merge sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$
Linear Search	$\Omega(1)$	$\Theta(n)$	$O(n)$
Binary Search	$\Omega(1)$	$\Theta(\log n)$	$O(\log n)$

Table 4: Auxiliary space complexity of the given algorithms.

Algorithm	**Auxiliary Space Complexity**
Insertion sort	$O(1)$
Merge sort	$O(n)$
Counting sort	$O(n+k)$
Linear Search	$O(1)$
Binary Search	$O(1)$

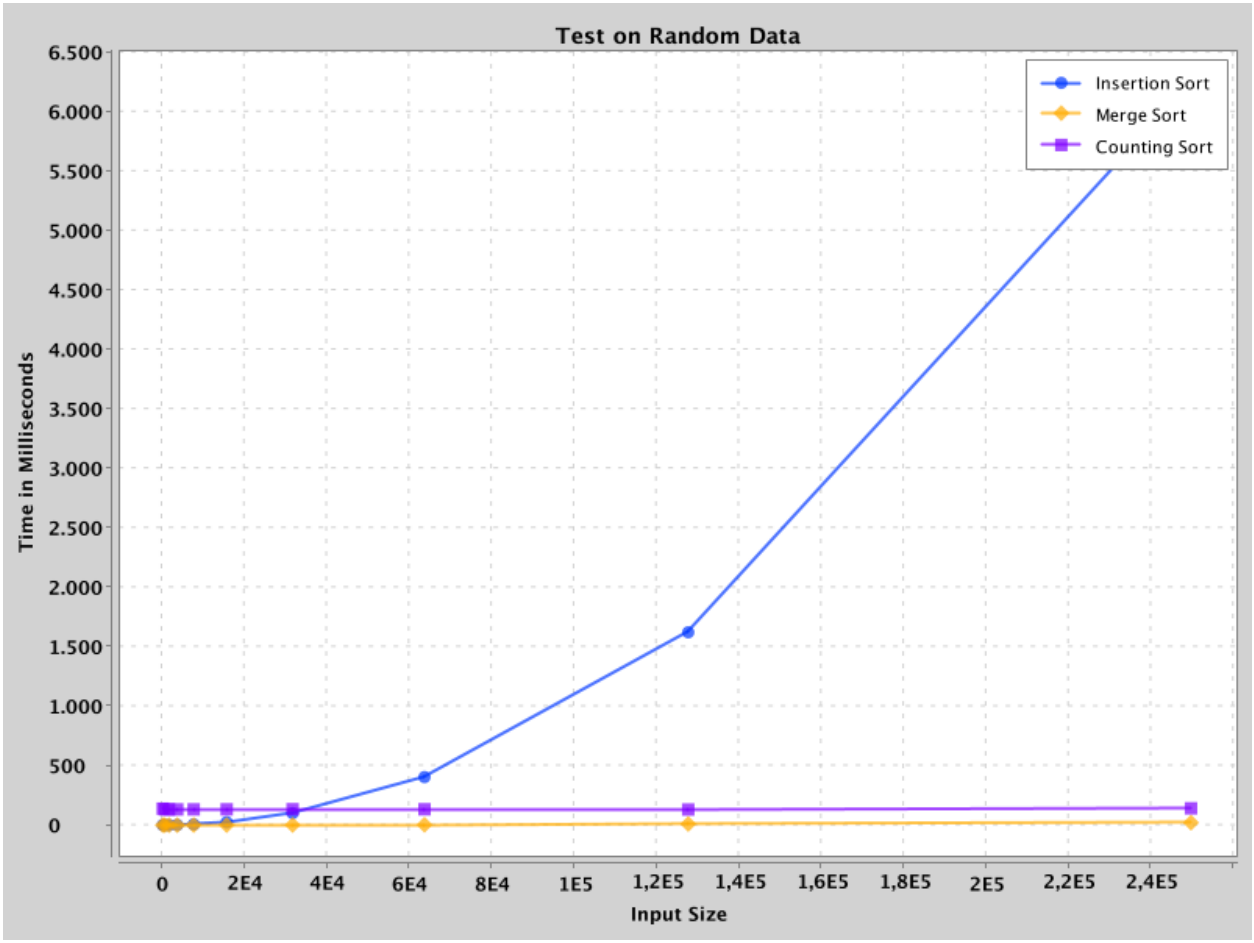

Figure 1: Sorting with the random data.

In this test case, the complexity of the algorithms was examined when they had the average case scenario. The theoretical $\Theta(n^2)$ time complexity for the Insertion Sort algorithm was reached as a result of the experiment. The theoretical $\Theta(n \log n)$ time complexity for the Merge Sort algorithm was achieved as a result of the experiment. Since the $\Theta(n + k)$ time complexity in the theory for the Counting Sort algorithm is a variable that depends on the k value, the graph progressed horizontally in a straight direction, as in theory. As a result of this experiment, when the size of our data set increases, the Merge Sort or Counting Sort algorithm can be selected according to the k value. If the k value is very large, as in this experiment, the Merge Sort algorithm can be chosen, otherwise the Counting Sort algorithm can be chosen.

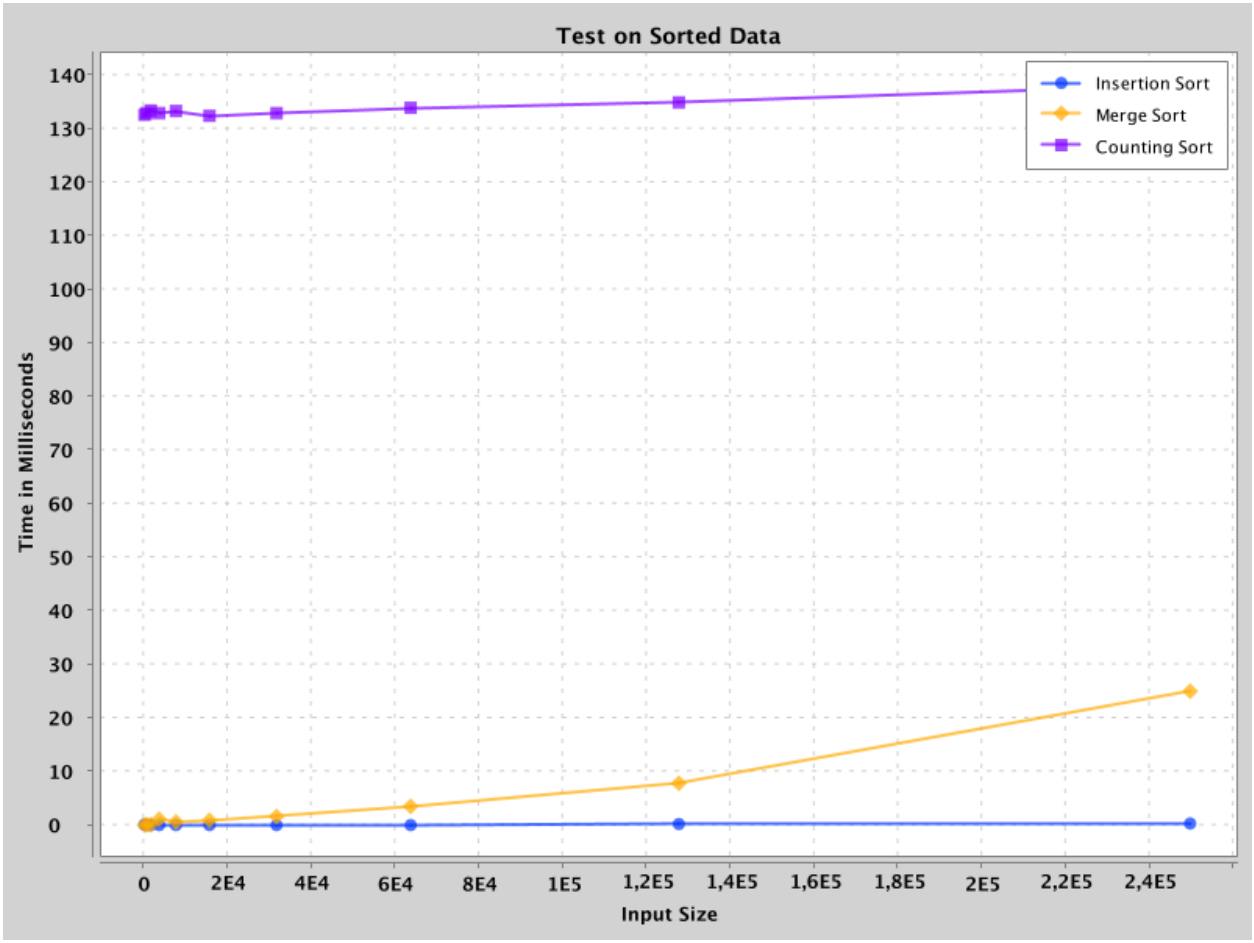

Figure 2: Sorting with the sorted data.

In this test case, the complexity of the algorithms was examined when they had the best case scenario. Although the theoretical $\Omega(n)$ time complexity for the Insertion Sort algorithm seems not to have been fully achieved as a result of the experiment, it is not understood because the time unit is milliseconds, but when examined in smaller time units such as nanoseconds, the theoretical time complexity for the insertion sort is reached. The theoretical $\Omega(n \log n)$ time complexity for the Merge Sort algorithm was achieved as a result of the experiment. Since the $\Omega(n + k)$ time complexity in the theory for the Counting Sort algorithm is a variable that depends on the k value, the graph progressed horizontally in a straight direction, as in theory. As a result of this experiment, choosing the Insertion Sort algorithm in the best scenario allows us to obtain more efficient results.

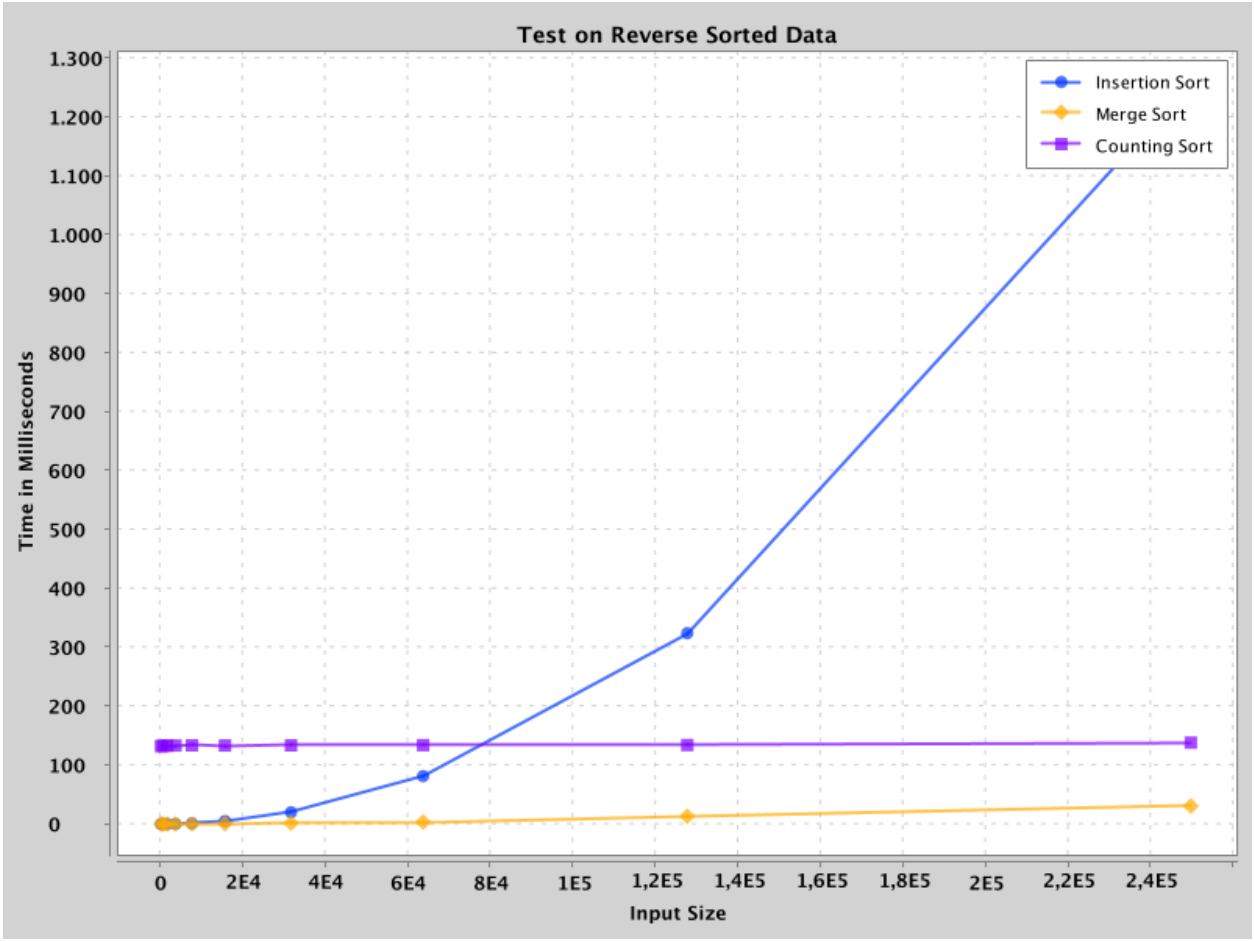

Figure 3: Sorting with the reverse sorted data.

In this test case, the complexity of the algorithms was examined when they had the worst case scenario. As a result of the experiment, theoretical $O(n^2)$ time complexity was reached for the Insertion Sort algorithm. As a result of the experiment, theoretical $O(n \log n)$ time complexity was achieved for the Sort by Merge Sort algorithm. Since the theoretically $O(n + k)$ time complexity for the Counting Sort algorithm is a variable that depends on the k value, the graph progressed in a horizontally straight direction as in theory. As a result of this experiment, when the size of our data set increases, the Merge Sort or Counting Sort algorithm can be selected according to the k value. It is understood from the movement of the lines that when the k value is extremely large, the time complexity of the Merge Sort algorithm will be more than the time complexity of the Counting Sort algorithm. That is, if the k value is extremely large, counting sort can be chosen.

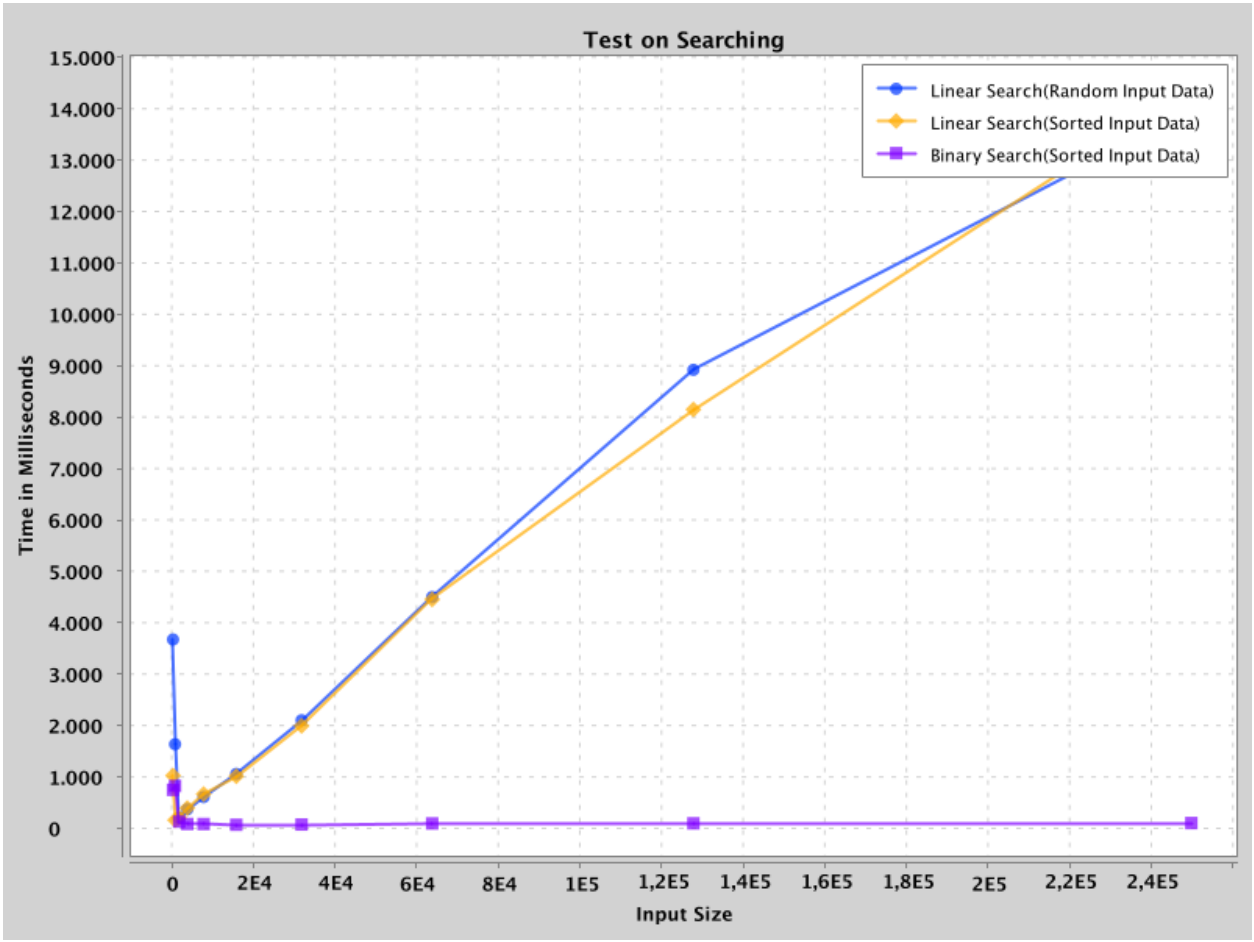

Figure 4: Searching Graphs.

In this test case, the time complexity performances of the algorithms were examined with random and sorted data sets for Linear Search and with sorted data sets for Binary Search. The algorithms were run 1000 times in all test cases. In each study, a random element from the array was selected and searched. Then their averages were taken and a graph was drawn.