Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
Dursun Zahid Korkmaz

Student Number:
b2210356020

1 Problem Definition

The main aim of this assignment is to show the relationship between the running time of the algorithm implementations with their theoretical asymptotic complexities.

2 Solution Implementation

Firstly, we implement the sorting and searching algorithms we intend to analyze.

2.1 Insertion Sort

The implementation:

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

2.2 Merge Sort

The implementation:

```
15 public class Algorithms {
16     private static void merge(int arr[], int l, int m, int r) {
17         int n1 = m - l + 1;
18         int n2 = r - m;
19
20         int leftArr[] = new int[n1];
21         int rightArr[] = new int[n2];
22
23         for (int i = 0; i < n1; ++i)
24             leftArr[i] = arr[l + i];
25         for (int j = 0; j < n2; ++j)
26             rightArr[j] = arr[m + 1 + j];
27
28         int i = 0, j = 0;
29
```

```
        int k = l;
        while (i < n1 && j < n2) {
            if (leftArr[i] <= rightArr[j]) {
                arr[k] = leftArr[i];
                i++;
            } else {
                arr[k] = rightArr[j];
                j++;
            }
            k++;
        }

        while (i < n1) {
            arr[k] = leftArr[i];
            i++;
            k++;
        }

        while (j < n2) {
            arr[k] = rightArr[j];
            j++;
            k++;
        }
    }

    public static void mergeSort(int arr[], int left, int right) {
        if (left < right) {
            int mid = left + (right - left) / 2;

            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);

            merge(arr, left, mid, right);
        }
    }
}
```

2.3 Count Sort

The implementation:

```
66 public class Algorithms {
67     public static void countSort(int[] inputArray) {
68         int maxElement = 0;
69 
70         for (int i = 0; i < inputArray.length; i++) {
71             maxElement = Math.max(maxElement, inputArray[i]);
72         }
73 
74         int[] countArray = new int[maxElement + 1];
75 
76         for (int i = 0; i < inputArray.length; i++) {
77             countArray[inputArray[i]]++;
78         }
79 
80         int index = 0;
81         for (int i = 0; i <= maxElement; i++) {
82             for (int j = countArray[i]; j > 0; j--) {
83                 inputArray[index++] = i;
84             }
85         }
86     }
87 }
```

2.4 Linear Search

The implementation:

```
88  public class Algorithms {
89      public static int linearSearch(int arr[], int x) {
90          for (int i = 0; i < arr.length; i++) {
91              if (arr[i] == x)
92                  return i;
93          }
94          return -1;
95      }
96  }
```

2.5 Binary Search

The implementation:

```
97  public class Algorithms {
98      public static int binarySearch(int arr[], int x) {
99          int low = 0;
100         int high = arr.length - 1;
101         while (low <= high) {
102             int mid = low + (high - low) / 2;
103
104             if (arr[mid] == x)
105                 return mid;
106
107             if (arr[mid] < x)
108                 low = mid + 1;
109
110             else
111                 high = mid - 1;
112         }
113
114         return -1;
115     }
116 }
```

3 Results, Analysis, Discussion

We have run each test in sorting section 10 times and each test in searching section 1000 times. After that we averaged the results.

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	0	0	0	3	13	47	184	758	3147
Merge sort	0	1	0	0	0	0	2	4	9	18
Counting sort	84	68	66	66	68	66	68	77	74	95
	Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	0	0	0	0	0	0	0
Merge sort	0	0	0	0	0	0	1	4	7	8
Counting sort	0	0	0	0	0	0	0	0	0	78
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	1	6	24	96	389	1498	5566
Merge sort	0	0	0	0	0	0	0	1	3	6
Counting sort	71	71	78	77	75	74	73	74	72	75

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)	2184	1519	290	468	800	1450	2573	4782	8505	9839
Linear search (sorted data)	183	203	311	545	1005	1863	3530	6790	13735	15968
Binary search (sorted data)	188	94	101	84	109	148	159	173	222	511

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	$\Omega(1)$	$\Theta(logn)$	$O(logn)$

Table 4: Auxiliary space complexity of the given algorithms.

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

Counting Sort's Auxilary Space Complexity is decided by this line in the psuedo code: for i $\leftarrow$ 2, . . . , k + 1 do

because it repeats a step k times

Linear Search doesn't have Auxilary Space necessity.

Binary Search doesn't have Auxilary Space necessity.

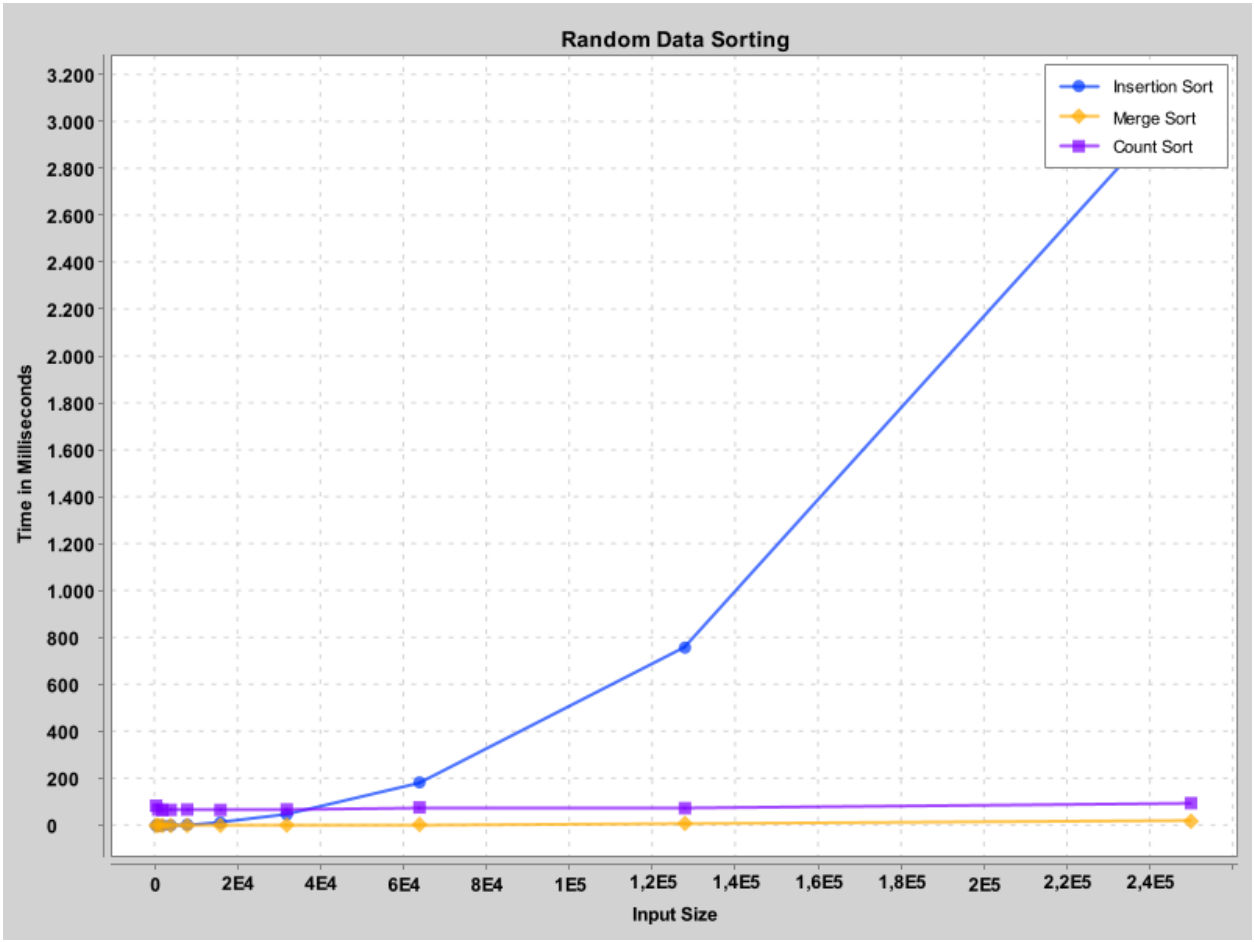

Figure 1: The graph comparison of our sorting algorithms tested on random data.

4 Notes

The results we achieved are not perfect because of some probable errors in the code so we can not directly analyze the theoretical complexity reflect on reality in these graphs. However when analyzed carefully, it can be seen that Insertion Sort is the slowest on average case as stated in theoretical analysis.

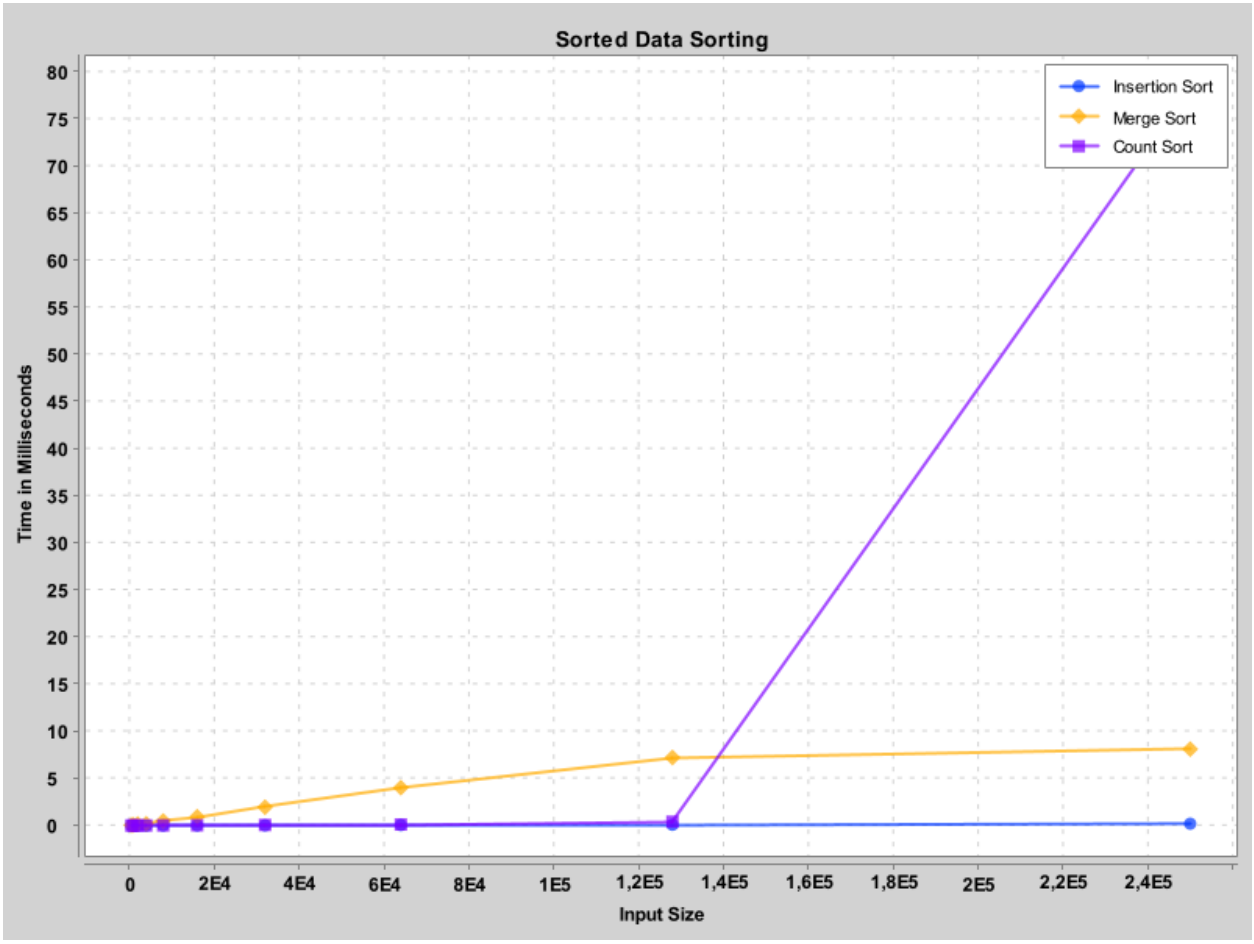

Figure 2: The graph comparison of our sorting algorithms tested on sorted data.

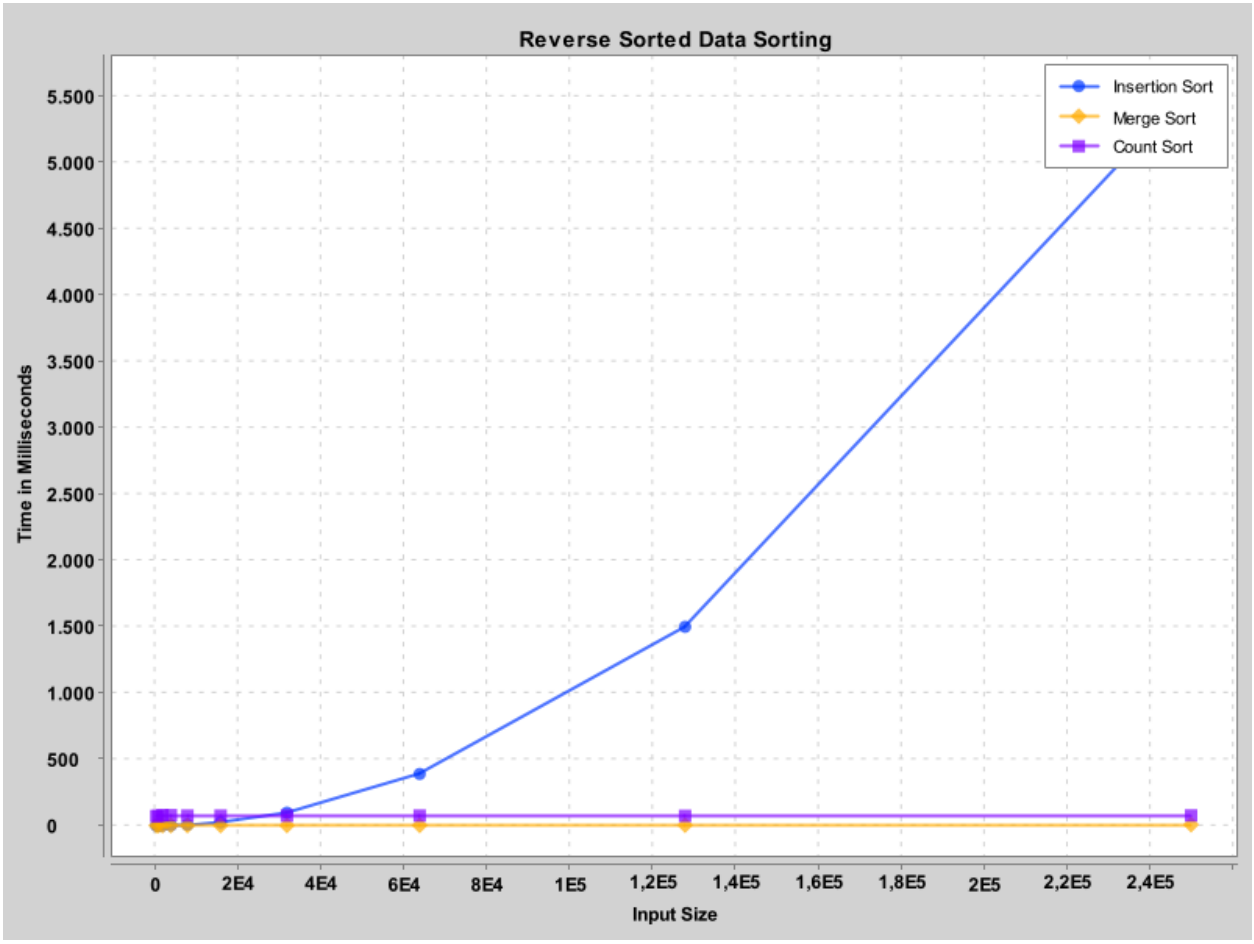

Figure 3: The graph comparison of our sorting algorithms tested on reversely sorted data.

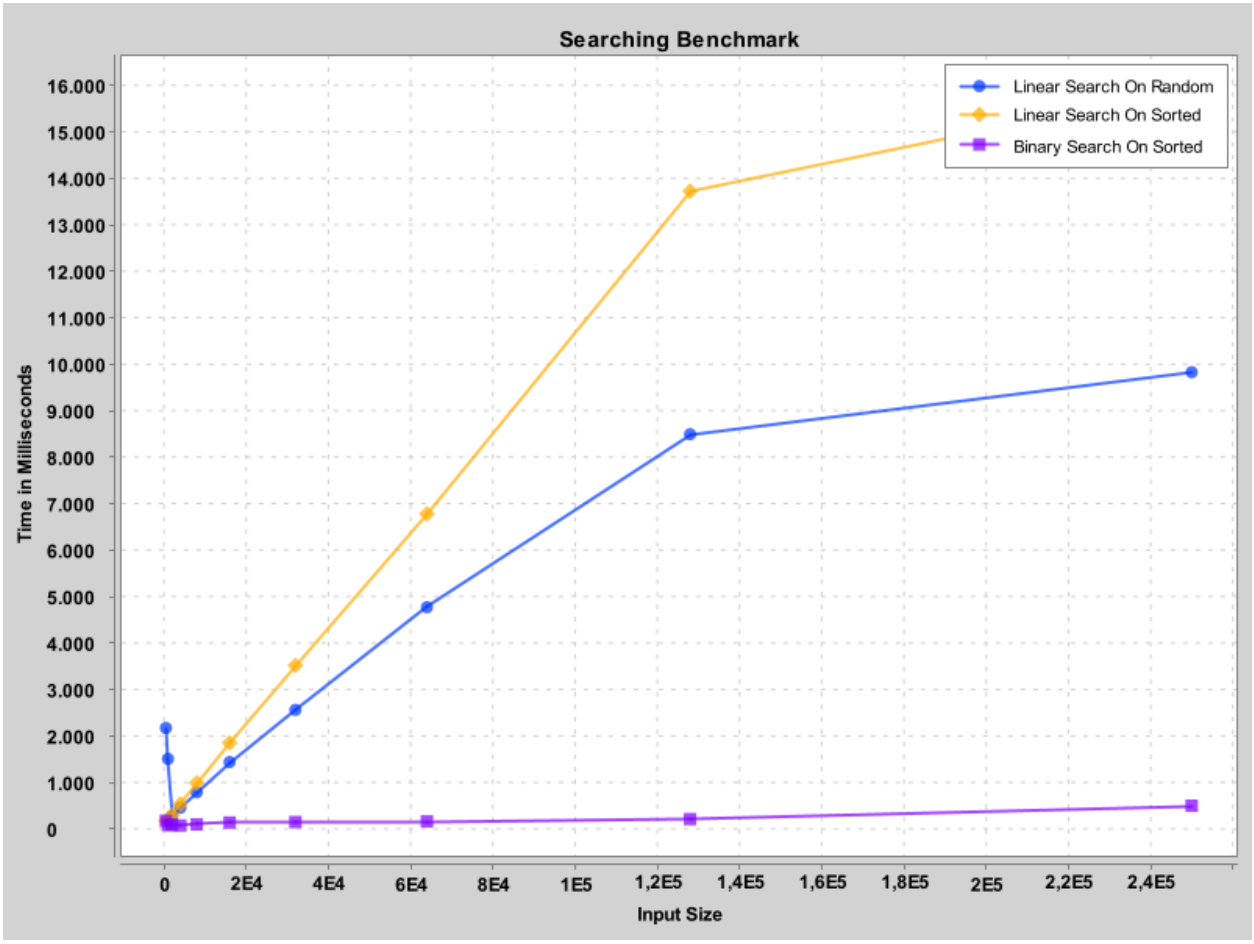

Figure 4: The graph comparison of our search algorithms tested on different datasets.