Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 20, 2024

Student name:
Gazi Kağan Soysal

Student Number:
b2210356050

1 Problem Definition

Our goal when writing our codes is not only to ensure that the code works correctly, but also to make it take as little time as possible. Because when we process large data sets, it can take a very long time. If it takes too long, its usefulness in daily life applications decreases. As qualified engineers of the future, we must take this aspect of our projects into consideration.

2 Solution Implementation

After implementing some popular sort and search algorithms in our project, we compare the average times we obtained after running these algorithms with data sets of different sizes by creating a graph.

2.1 Merge Sort Algorithm

```
1  public class SortAlgorithms {
2      static int[] mergeSort(int[] datas) {
3          int size = datas.length;
4
5          if (size <= 1) return datas;
6          int[] leftArray = Arrays.copyOfRange(datas, 0, size/2);
7          int[] rightArray = Arrays.copyOfRange(datas, size/2, size);
8
9          leftArray = mergeSort(leftArray);
10         rightArray = mergeSort(rightArray);
11
12         return merge(leftArray, rightArray);
13
14     }
15
16     static int[] merge(int[] left, int[] right) {
17         int[] mergeArray = new int[left.length + right.length];
18
19         int leftIndex = 0, rightIndex = 0, mergeIndex = 0;
20
21         while (leftIndex < left.length || rightIndex < right.length) {
22             if (leftIndex == left.length || (rightIndex < right.length && left
                   [leftIndex] > right[rightIndex])) {
23                 mergeArray[mergeIndex++] = right[rightIndex++];
24             } else if (rightIndex == right.length || (leftIndex < left.length
                   && left[leftIndex] <= right[rightIndex])) {
25                 mergeArray[mergeIndex++] = left[leftIndex++];
26             }
27         }
28
29         return mergeArray;
30     }}
```

2.2 Insertion Sort Algorithm

```
public class SortAlgorithms {
    static int[] insertionSort(int[] datas) {
        for (int j=1; j < datas.length; j++) {
            int key = datas[j];
            int i = j-1;

            while (i >= 0 && datas[i] > key) {
                datas[i+1] = datas[i];
                i--;
            }
            datas[i+1] = key;
        }
        return datas;
    }
}
```

2.3 Counting Sort Algorithm

```
public class SortAlgorithm {
    static  int[] countingSort(int[] datas) {
        int maxValue = Utilities.findMax(datas);

        int[] count = new int[maxValue + 1];
        int[] output = new int[datas.length];

        for (int i : datas) count[i]++;
        for (int i = 1; i < maxValue+1; i++)
            count[i] = count[i] + count[i - 1];

        for (int i = datas.length-1; i >= 0; i--) {
            int j = datas[i];
            count[j]--;
            output[count[j]] = datas[i];
        }
        return output;
    }
}
```

2.4 Linear Search Algorithm

```
public class SearchAlgorithms {
    static int linearSearch(int[] datas, int value) {
        for (int i = 0; i < datas.length; i++) {
            if (datas[i] == value) return i;
        }
        return -1;
    }
}
```

2.5 Binary Search Algorithm

```
public class SortAlgorithms {
    static int binarySearch(int[] datas, int value) {
        int low = 0;
        int high = datas.length - 1;

        while (high - low > 1) {
            int mid = (high + low) / 2;

            if (datas[mid] < value) low = mid + 1;
            else high = mid;
        }

        if (datas[low] == value) return low;
        else if (datas[high] == value) return high;

        return -1;
    }
}
```

3 Results, Analysis, Discussion

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	0.3	0.1	0.2	0.7	2.6	11.2	41.8	162.4	672.9	2774.4
Merge sort	0.1	0.0	0.1	0.3	0.6	1.2	2.5	5.6	11.7	25.0
Counting sort	117.3	93.5	91.8	96.6	92.9	92.6	92.5	94.7	102.3	96.9
	Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.2	0.4
Merge sort	0.0	0.0	0.0	0.2	0.3	0.6	1.1	2.6	5.2	10.6
Counting sort	92.9	93.9	92.8	92.7	93.2	91.7	92.2	94.8	93.4	95.3
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.1	0.1	0.3	1.2	5.3	20.2	81.9	328.6	1338.7	4906.8
Merge sort	0.0	0.0	0.1	0.2	0.3	0.5	1.2	2.4	4.8	11.2
Counting sort	93.8	94.2	93.6	95.8	93.2	94.1	92.5	96.3	96.3	97.2

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)	1749.0	245.2	180.6	248.3	420.1	988.5	1052.9	2410.7	4999.7	12301.6
Linear search (sorted data)	718.5	120.4	171.6	275.8	556.8	1079.5	2118.4	4159.3	8676.8	16662.0
Binary search (sorted data)	162.3	71.3	96.6	51.7	40.4	45.4	49.9	41.0	45.5	46.1

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	$\Omega(1)$	$\Theta(\log n)$	$O(n \log n)$

- Since the best case of sort algorithms is sorted cases, the best case comes out better in insertion sort, but the complexity does not change since other algorithms will perform the same operation even if they are sorted.

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	$O(1)$

- Insertion Sort: No extra space is assigned in this algorithm. It operates in-place, thereby resulting in an auxiliary space complexity of O(1).
- Merge Sort: These lines demonstrate the creation of supplementary arrays (left, right, C) to store intermediate results throughout the merge sort procedure. Consequently, the auxiliary space complexity amounts to O(n).
  Line 6: left ← A[1]...A[n/2]
  Line 7: right ← A[n/2+1]...A[n]
  Line 13: C: array
- Counting Sort: These lines reserve extra space for the count array and the output array. The dimensions of these arrays are contingent on the range of elements within the input array, represented by k. As a result, the auxiliary space complexity becomes O(k + n).
  Line 2: count ← array of k + 1 zeros
  Line 3: output ← array of the same length as A
- Linear Search: This algorithm does not assign any extra space. It operates solely using a fixed amount of additional space for storing loop counters and temporary variables. Hence, the auxiliary space complexity remains O(1).
- Binary Search: No extra space is assigned in this algorithm. It operates solely with a constant amount of additional space for storing loop counters and temporary variables. Hence, the auxiliary space complexity remains O(1).

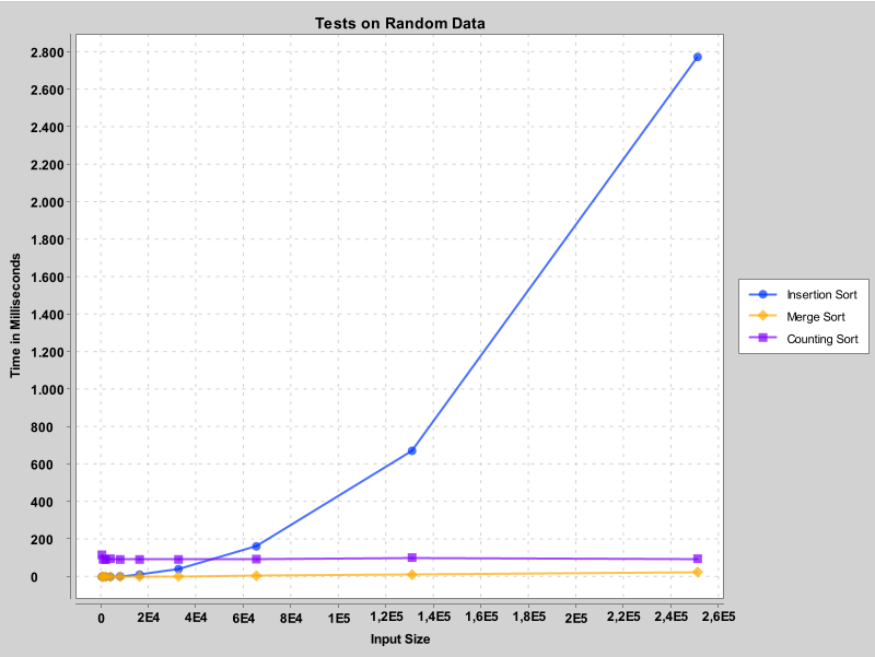

Figure 1: Tests on Random Data

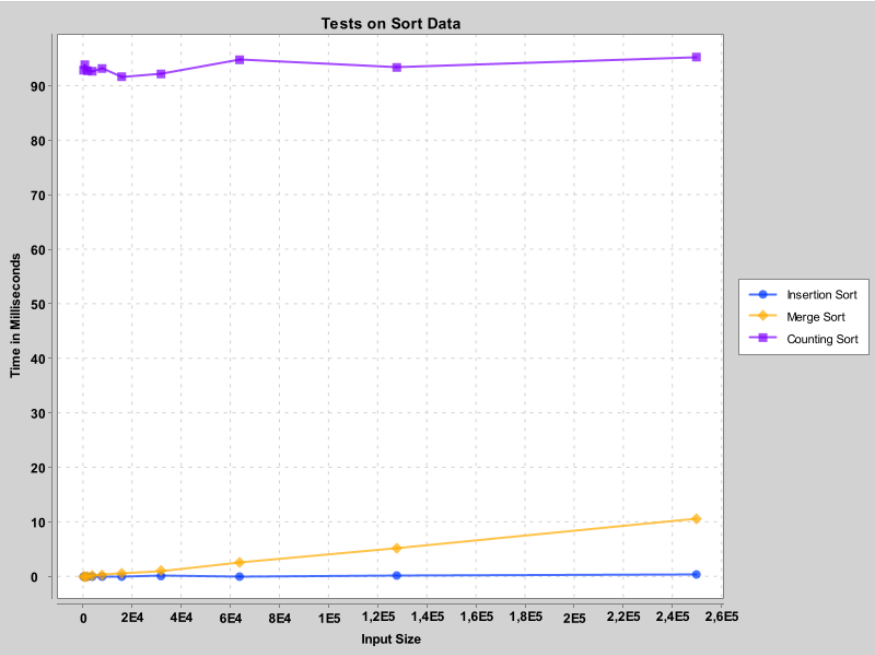

Figure 2: Tests on Sorted Data

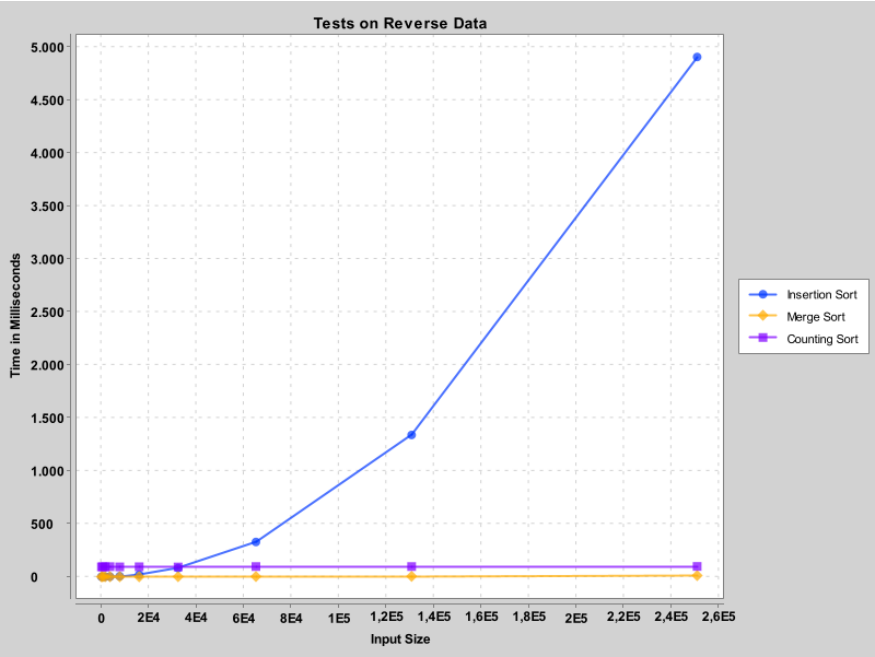

Figure 3: Tests on Reverse Data

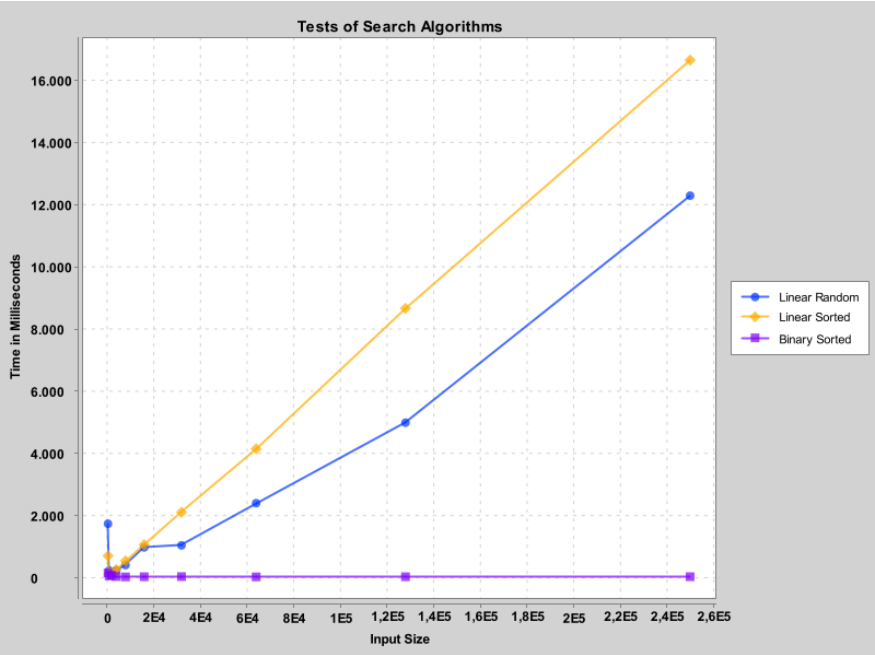

Figure 4: Tests of Search Algorithms

4 Notes

- When working with random data or reverse data, working with the insertion sort algorithm takes a lot of time for large data sizes. Therefore, it would be more useful to use merge sort or counting sort algorithms to improve performance when working with this data.
- Algorithms will take much less time when working with already sorted data. However, in our chart, the counting sort algorithm takes longer than others. I guess this may be due to the data range we are working with being too high.
- While analyzing the search algorithms, I tried to choose the value to be searched in a balanced way so that the results would be more consistent. For this reason, I chose 10 values to search and these values are evenly distributed in the data. Additionally, as can be seen from our graph, the binary search algorithm is a much more efficient algorithm compared to linear.