Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
Bora Dere

Student Number:
2220765021

1 Problem Definition

In the fast-paced world of software development, where every millisecond counts, optimizing performance is paramount. With a plethora of algorithms available for accomplishing the same task, determining the most efficient one for our specific use case can be a daunting task.

This assignment serves as a crucial step in our journey to identify and implement the algorithm that best aligns with our performance requirements and computational constraints. By thoroughly understanding the complexities and nuances of each algorithm, we can make informed decisions to ensure our code runs swiftly and smoothly, delivering optimal results to our users.

2 Solution Implementation

Procedures for algorithms are given in the assignment explanation. Those procedures were followed in the process of implementing the algorithms.

2.1 Insertion Sort

```
public class InsertionSort {
    public static void insertionSort(int[] A) {
        for (int j = 0; j < A.length; j++) {
            int key = A[j];
            int i = j - 1;

            while (i > -1 && A[i] > key) {
                A[i + 1] = A[i];
                i--;
            }

            A[i + 1] = key;
        }
    }
}
```

2.2 Counting Sort

```
public class CountingSort {
    public static int[] countingSort(int[] A, int k) {
        int[] count = new int[k + 1];
        int size = A.length;
        int[] output = new int[size];

        for (int j : A) {
            count[j]++;
        }
        for (int i = 1; i <= k; i++) {
            count[i] += count[i - 1];
        }
        for (int i = size - 1; i >= 0; i--) {
            output[count[A[i]] - 1] = A[i];
            count[A[i]]--;
        }
        return output;
    }
}
```

2.3 Merge Sort

Main (mergeSort) function:

```
public class MergeSort {
    public static int[] mergeSort(int[] array) {
        int n = array.length;

        if (n <= 1) {
            return array;
        }

        int mid = n / 2;
        int[] left = new int[mid];
        int[] right = new int[n - mid];

        System.arraycopy(array, 0, left, 0, mid);
        System.arraycopy(array, mid, right, 0, n - mid);

        left = mergeSort(left);
        right = mergeSort(right);

        return merge(left, right);
    }
}
```

Helper (merge) function:

```
public class MergeSort {
    static int[] merge(int[] A, int[] B) {
        int lenA = A.length;
        int lenB = B.length;

        int[] C = new int[lenA + lenB];
        int i = 0, j = 0, k = 0;

        while (i < lenA && j < lenB) {
            if (A[i] > B[j]) {
                C[k++] = B[j++];
            } else {
                C[k++] = A[i++];
            }
        }

        while (i < lenA) {
            C[k++] = A[i++];
        }

        while (j < lenB) {
            C[k++] = B[j++];
        }

        return C;
    }
}
```

3 Results, Analysis, Discussion

As instructed, sorting algorithms were ran 10 times and average duration of those 10 runs were taken, in milliseconds. For searching algorithms, 1000 runs were performed in a similar fashion, but in nanoseconds. For ease of readability, values are rounded to one decimal place.

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									
Insertion sort	0.1	0.1	0.2	0.2	0.8	2.2	6.8	29.2	110.7	482.1
Merge sort	0	0	0.1	0.1	0.5	0.9	1.6	3.5	7.1	52.9
Counting sort	196.8	97.1	96.8	104.5	96	97.2	94.8	99.2	117.1	309.5
	Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	0	0	0	0	0.1	0.1	0.4
Merge sort	0	0.3	0.3	0.9	1.4	3.2	1.9	3.9	7.2	15.6
Counting sort	275.1	129	119.1	160.2	137.9	119.1	98.4	102	105.3	100.2
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0.1	0.3	0.8	3.3	13.6	54.6	218.9	838.3
Merge sort	0	0.1	0.1	0.2	0.4	1	1.9	4.3	7.1	15.5
Counting sort	113.9	97.5	99.6	105.4	103.6	106.7	103.7	96.8	101.3	109.1

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)	2000	3000	2000	0	0	2000	2000	5000	12000	18000
Linear search (sorted data)	0	0	0	1000	0	2000	2000	5000	9000	17000
Binary search (sorted data)	0	0	1000	0	0	0	0	0	0	1000

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

Explanation for Table 3:
Insertion Sort:

- **Best case:** When the array is already sorted, and each element is compared only once with its preceding elements before finding its correct position.
- **Average case:** On average, each element is compared roughly half of the sorted portion of the array before finding its correct position.
- **Worst case:** When the array is sorted in reverse order, and each element has to be compared with all elements in the sorted portion of the array before finding its correct position.

Merge Sort:

- **Best case:** Regardless of the input distribution, Merge Sort divides the array into halves recursively until each sub-array contains only one element, and then merges them back together.
- **Average case:** Same as the best case.
- **Worst case:** Same as others, input distribution does not do any change.

Counting Sort:

- **Best case:** When the range of the input is smaller compared to the number of elements, and the elements are uniformly distributed.
- **Average case:** Same as the best case.
- **Worst case:** The worst-case complexity of Counting Sort is also O(n + k), where n is the number of elements in the input array and k is the range of the input. Even in the worst case, Counting Sort's time complexity remains linear. This is one of the advantages of Counting Sort over comparison-based sorting algorithms.

Linear Search:

- **Best case:** When the element being searched for is found at the beginning of the array.
- **Average case:** On average, the element being searched for is found halfway through the array.
- **Worst case:** When the element being searched for is either not present in the array or present at the end of the array.

Binary Search:

- **Best case:** When the element being searched for is found at the beginning of the array.
- **Average case:** On average, the element being searched for is found halfway through the array.
- **Worst case:** When the element being searched for is either not present in the array or present at the end of the array.

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	$O(1)$

Explanation for Table 4:
Insertion Sort:

- Insertion Sort operates in-place, modifying the input array without requiring additional space.
- Lines contributing to auxiliary space complexity: None, as the algorithm sorts the array in place.

Merge Sort:

- Merge Sort typically requires additional space proportional to the size of the input array for the merging step.
- Lines contributing to auxiliary space complexity: Line 13, where the array C is initialized to store merged elements.

Counting Sort:

- Counting Sort requires additional space proportional to the size of the input array plus the range of the input.
- Lines contributing to auxiliary space complexity: Lines 2-3, where arrays count and output are initialized to store counts and sorted elements, respectively.

Linear Search:

- Linear Search doesn't require any additional space; it performs the search in place.
- Lines contributing to auxiliary space complexity: None, as the algorithm operates without any additional space.

Binary Search:

- Binary Search doesn't require any additional space; it operates on the given array without allocating extra memory.
- Lines contributing to auxiliary space complexity: None, as the algorithm operates without any additional space.

Plots of tests, for further investigation.

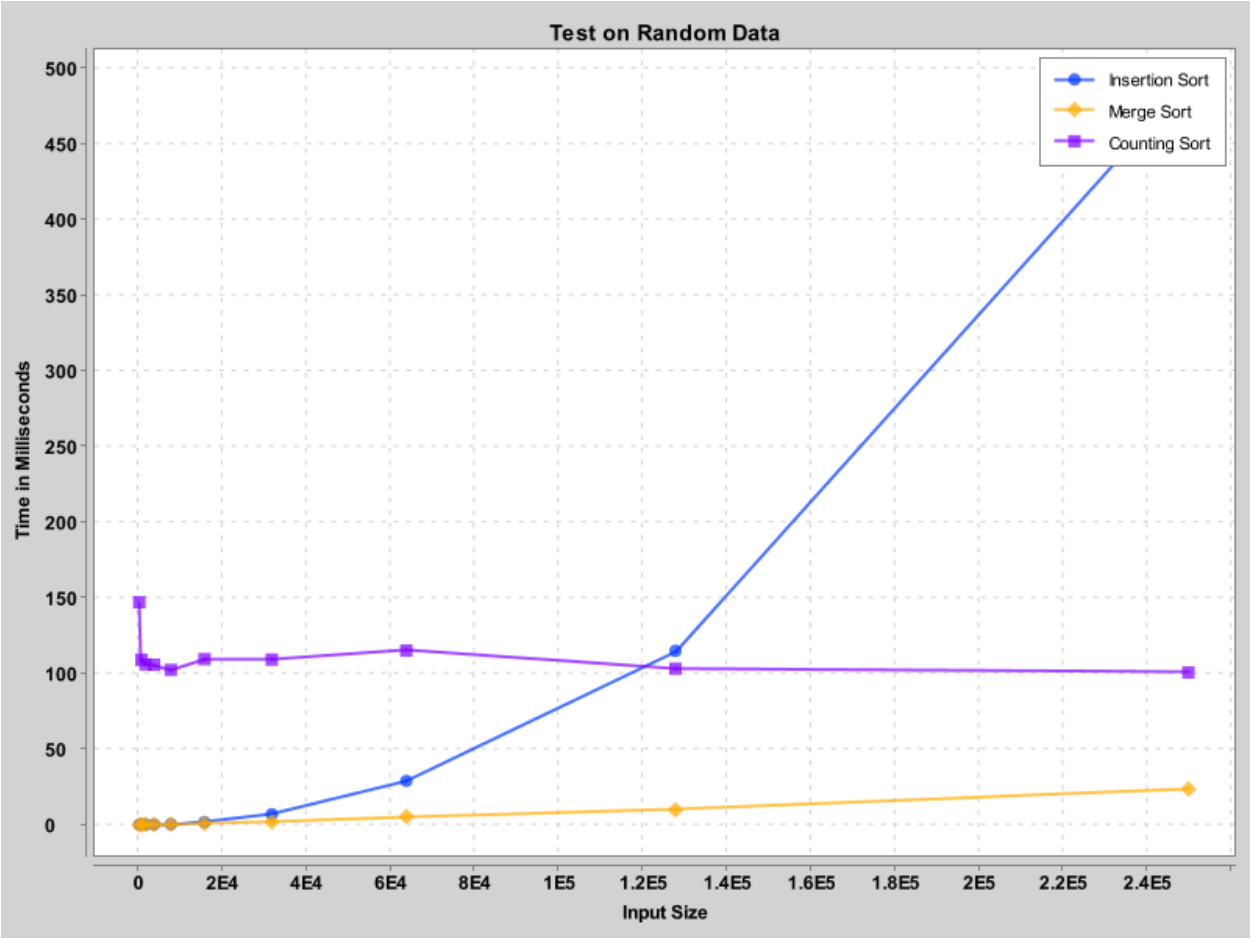

Figure 1: Tests on Random Data.

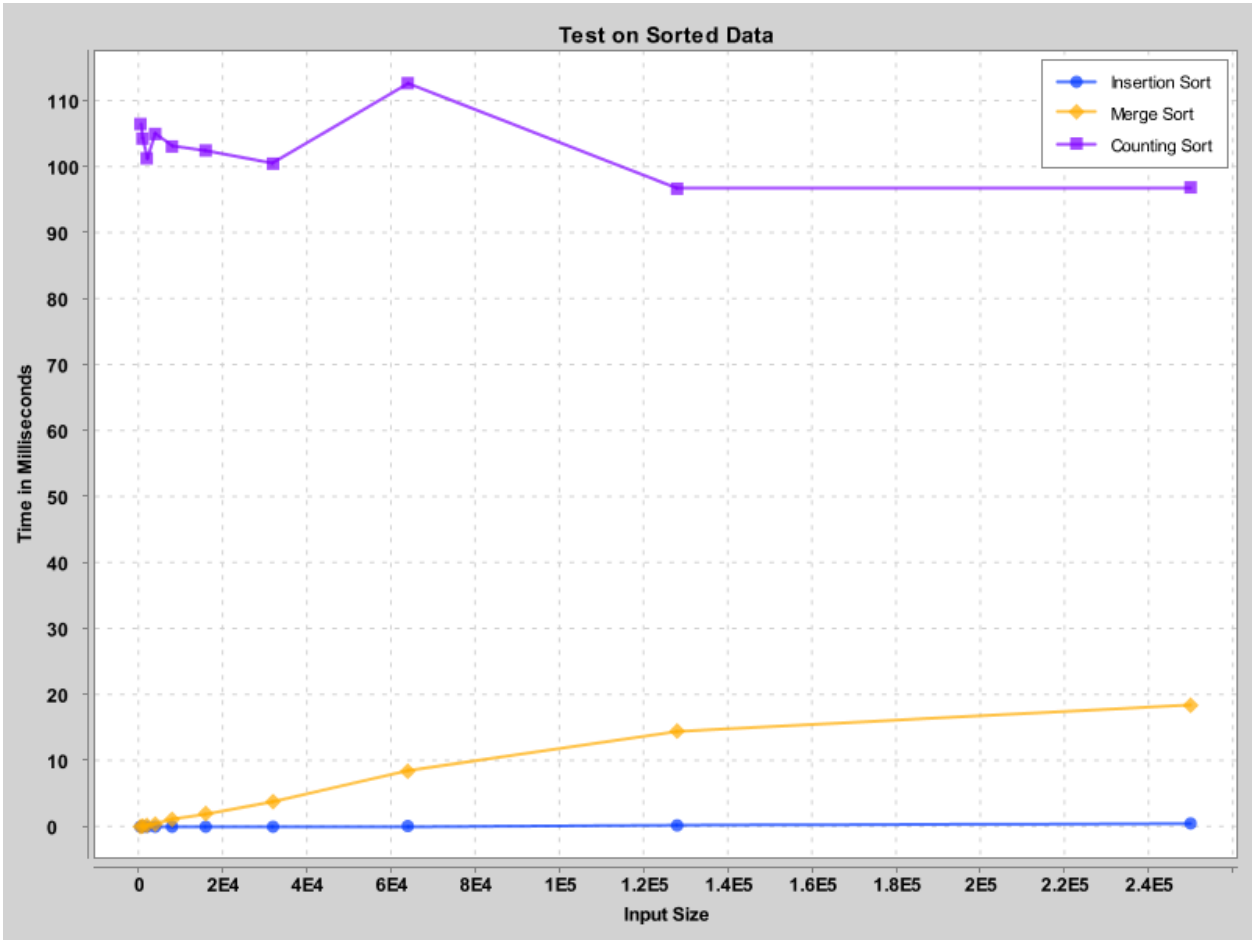

Figure 2: Tests on Sorted Data.

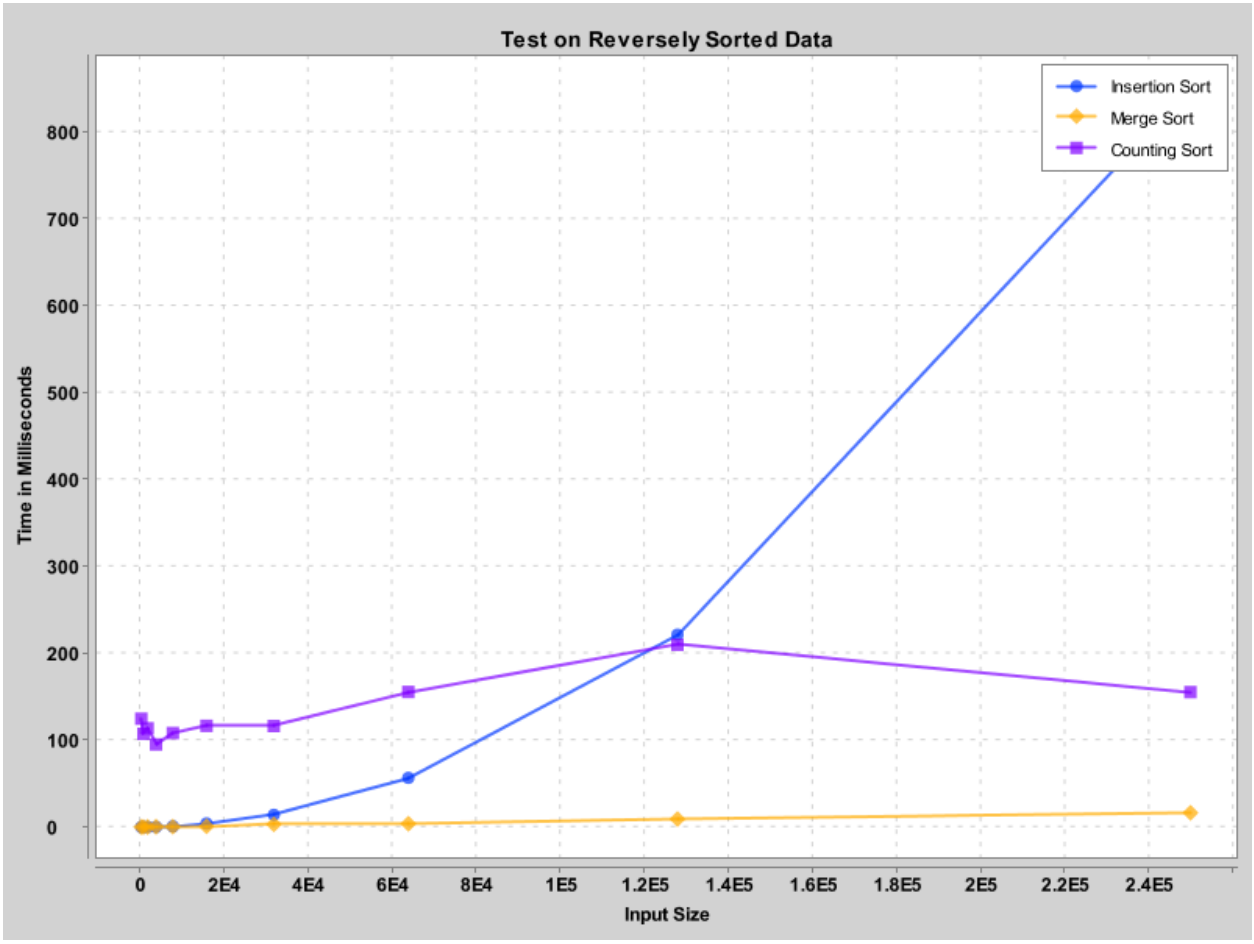

Figure 3: Tests on Reversely Sorted Data.