Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
Gökdeniz Şimşek

Student Number:
2210356067

1 Problem Definition

The main problem in this assignment is to measure the efficiency of different sorting and search algorithms. The goal is to compare the efficiency of these algorithms under different conditions.

The way Assignment works is to run these algorithms on lists with different input sizes and different features. For this, we apply sorting algorithms (such as Insertion Sort, Merge Sort, and Counting Sort) to randomly sorted lists, sorted lists, and reverse sorted lists. We also compare the results and show them on the graph. Similarly, in this assignment, the running times of the algorithms for searching (such as Linear Search, and Binary Search) are calculated for different input sizes. We compare the time of linear search on randomly sorted lists, sorted lists, and binary search algorithms on sorted lists and show it on the graph.

2 Solution Implementation

A Java application containing different sorting and searching algorithms is given to solve the situation described in the problem definition. Below are the codes for this application:

2.1 Insertion Sort

To sort an array of size N in ascending order iterates over the array and it compares the current element to its predecessor, if the key element is smaller than its predecessor, compare it to the elements before.

```java
public class Insertion {
    public static int[] insertionSort(int[] A) {
        for (int j = 1; j < A.length; j++) {
            int key = A[j];
            int i = j - 1;
            while (i >= 0 && A[i] > key) {
                A[i + 1] = A[i];
                i--;
            }
            A[i + 1] = key;
        }
        return A;
    }
}
```

2.2 Merge Sort

It is a sorting algorithm that works by splitting an array into smaller subarrays, sorting each subarray, and recombining the subarrays.

```java
public class Merge {
    public static int[] mergeSort(int[] A) {
        int n = A.length;
        if (n <= 1) {
            return A;
        }
        int mid = n / 2;
        int[] left = new int[mid];
        int[] right = new int[n - mid];
        // Split the array into two
        for (int i = 0; i < mid; i++) {
            left[i] = A[i];
        }
        for (int i = mid; i < n; i++) {
            right[i - mid] = A[i];
        }
        // Sort sorted subarrays
        left = mergeSort(left);
        right = mergeSort(right);
        // Combining
        return merge(left, right);
    }
    public static int[] merge(int[] A, int[] B) {
        int[] C = new int[A.length + B.length];
        int i = 0, j = 0, k = 0;
        // As long as both subarrays have elements
        while (i < A.length && j < B.length) {
            if (A[i] > B[j]) {
                C[k++] = B[j++];
            } else {
                C[k++] = A[i++];
            }
        }
        while (i < A.length) {
            C[k++] = A[i++];
        }
        while (j < B.length) {
            C[k++] = B[j++];
        }
        return C;
    }
}
```

2.3 Counting Sort

This sorting algorithm counts the frequency of each distinct element in the input array and places the elements in their correct sorted positions.

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

        // Count the occurrences of each element in A
        for (int i = 0; i < A.length; i++) {
            count[A[i]]++;
        }
        // Update the count array to store the actual position of each element
        for (int i = 1; i < count.length; i++) {
            count[i] += count[i - 1];
        }
        // Build the output array
        for (int i = A.length - 1; i >= 0; i--) {
            output[count[A[i]] - 1] = A[i];
            count[A[i]]--;
        }
        return output;
    }
}
```

2.4 Linear Search

The linear search algorithm starts at one end and examines each element of the list until the desired element is found.

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

2.5 Binary Search

Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half.

```java
public class BinarySearch {
    public static int binarySearch(int[] A, int x) {
        int low = 0;
        int high = A.length - 1;

        while (high - low > 1) {
            int mid = (high + low) / 2;
            if (A[mid] < x) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }

        if (A[low] == x) {
            return low;
        } else if (A[high] == x) {
            return high;
        }
        return -1;
    }
}
```

3 Results, Analysis, Discussion

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	0	0	0	0	1	6	27	115	437
Merge sort	0	0	0	0	0	1	3	7	14	28
Counting sort	400	124	124	125	125	125	127	127	133	128
	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	2	4	4	8	15
Counting sort	0	0	0	0	0	0	0	0	0	133
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	0	0	3	13	50	211	769
Merge sort	0	0	0	0	0	1	2	4	8	16
Counting sort	130	123	123	122	123	127	125	125	130	125

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)	1388	979	180	299	542	1115	2329	3778	9665	16835
Linear search (sorted data)	142	618	100	251	402	1078	2875	5053	12504	24194
Binary search (sorted data)	494	77	75	69	56	81	150	217	387	464

Complexity analysis tables (Table 3 and Table 4):

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

Table 4: Auxiliary space complexity of the given algorithms.

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

The time measurements are given in the tables above. Sorting algorithms are given in Table 1, and search algorithms are given in Table 2. Time and space complexities calculated according to measurements are given in Table 3 and Table 4.

According to the data obtained, the Insertion Sort algorithm works with the best efficiency on the sorted list. It works on randomly sorted and reversely sorted lists in a linear relationship with the number of inputs. When we look at Merge Sort, it works with similar efficiency for all list types. It works in a linear relationship with the number of inputs. Likewise, Counting Sort works with similar efficiency for all list types and input numbers.

When we examined the search algorithms, it was observed that the Linear Search algorithm worked more efficiently on a randomly sorted list. There is a linear time measurement with the number of inputs for both sorted and randomly sorted lists. When we look at the Binary Search algorithm, it works with a similar efficiency for all input numbers.

Graphs regarding the measurements are below:

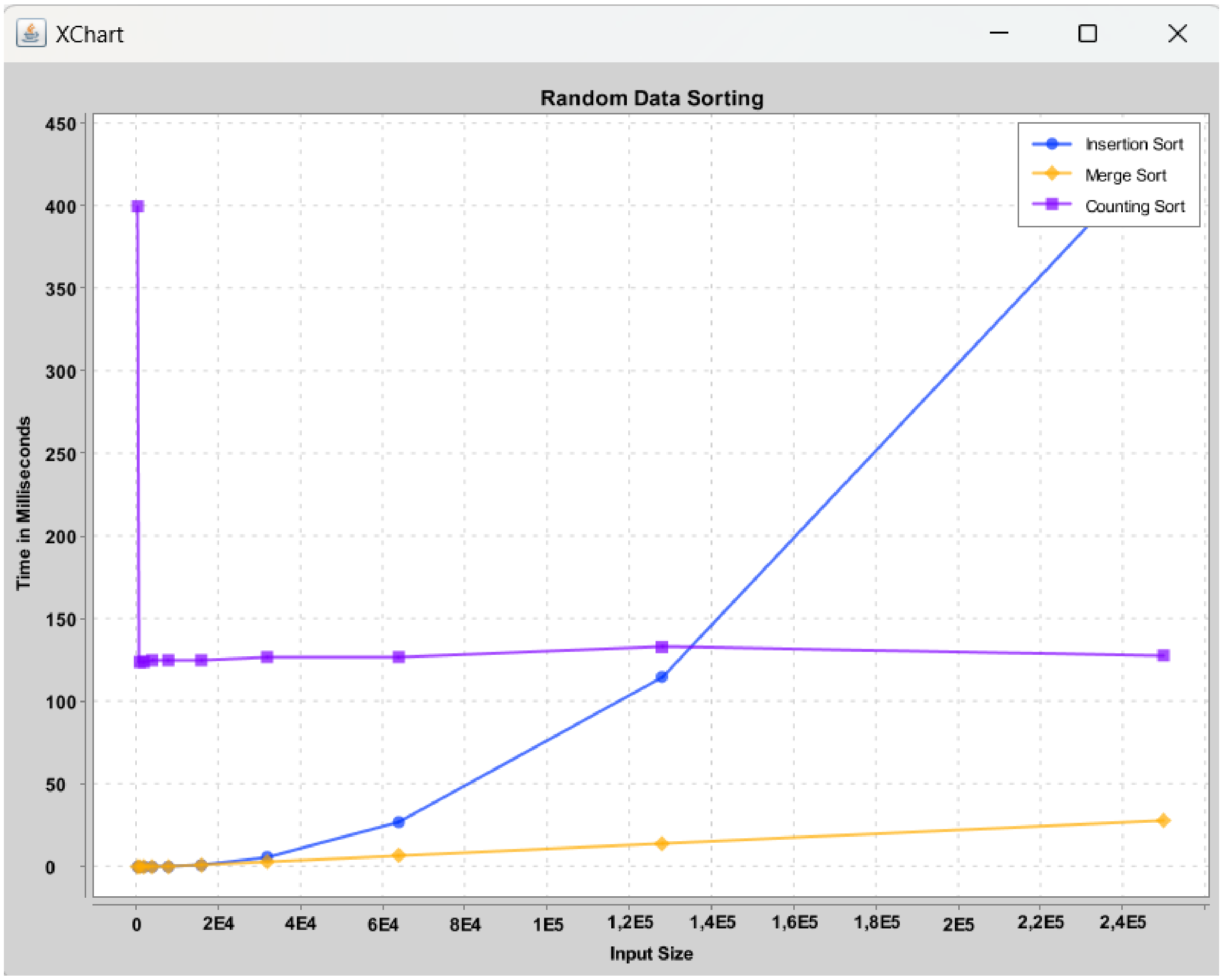

Figure 1: Sorting Algorithm Tests on Random Data

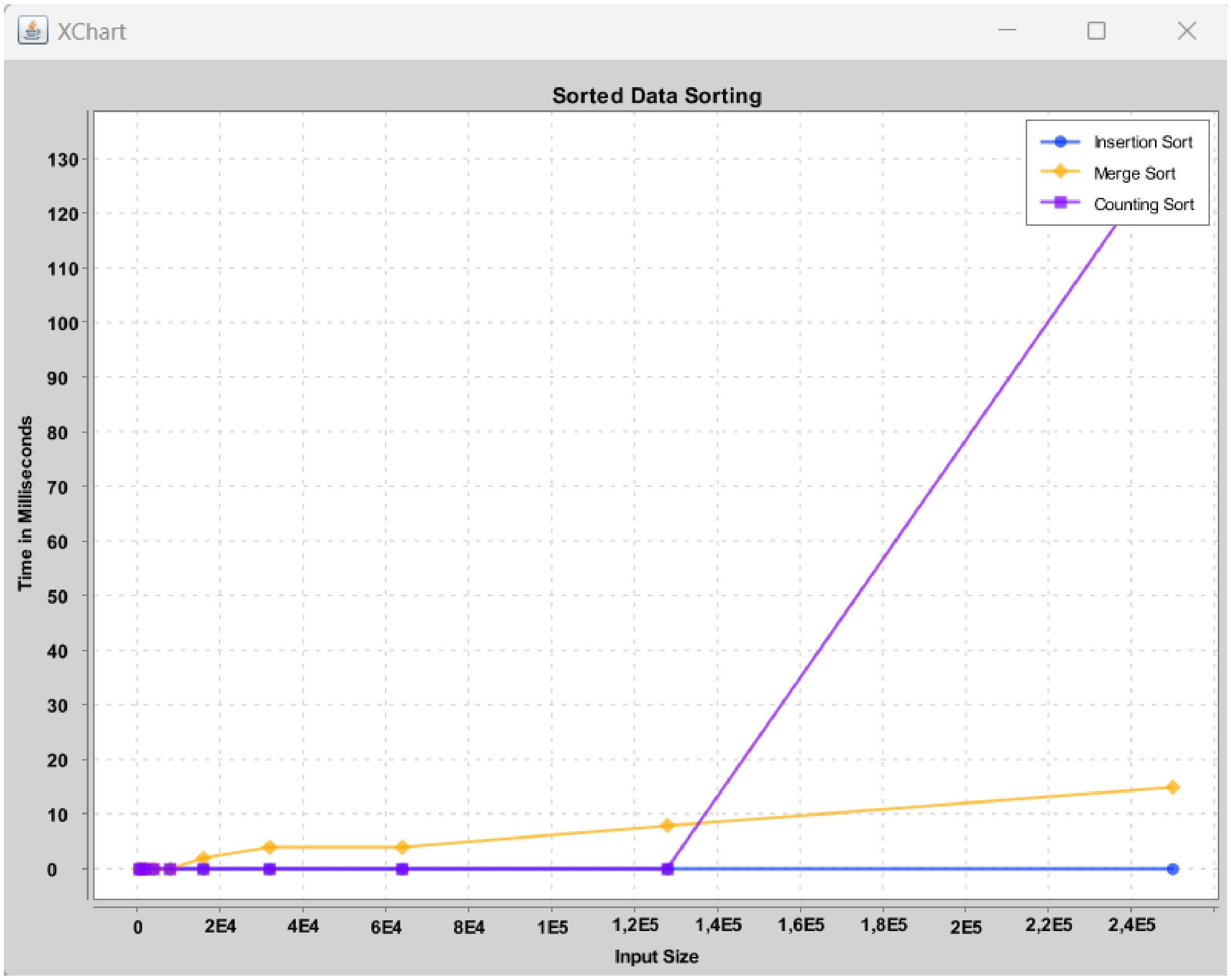

Figure 2: Sorting Algorithm Tests on Sorted Data

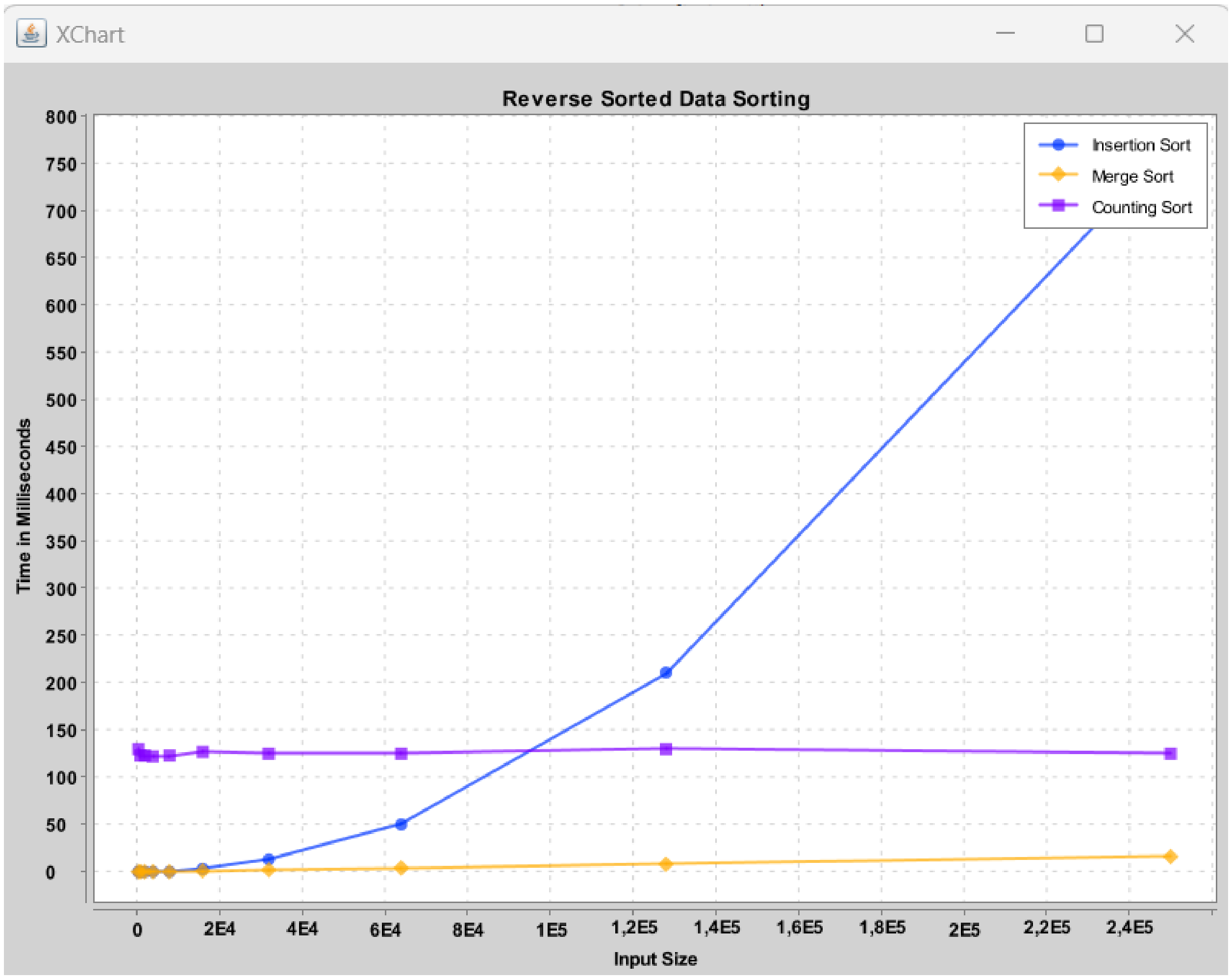

Figure 3: Sorting Algorithm Tests on Reverse Sorted Data

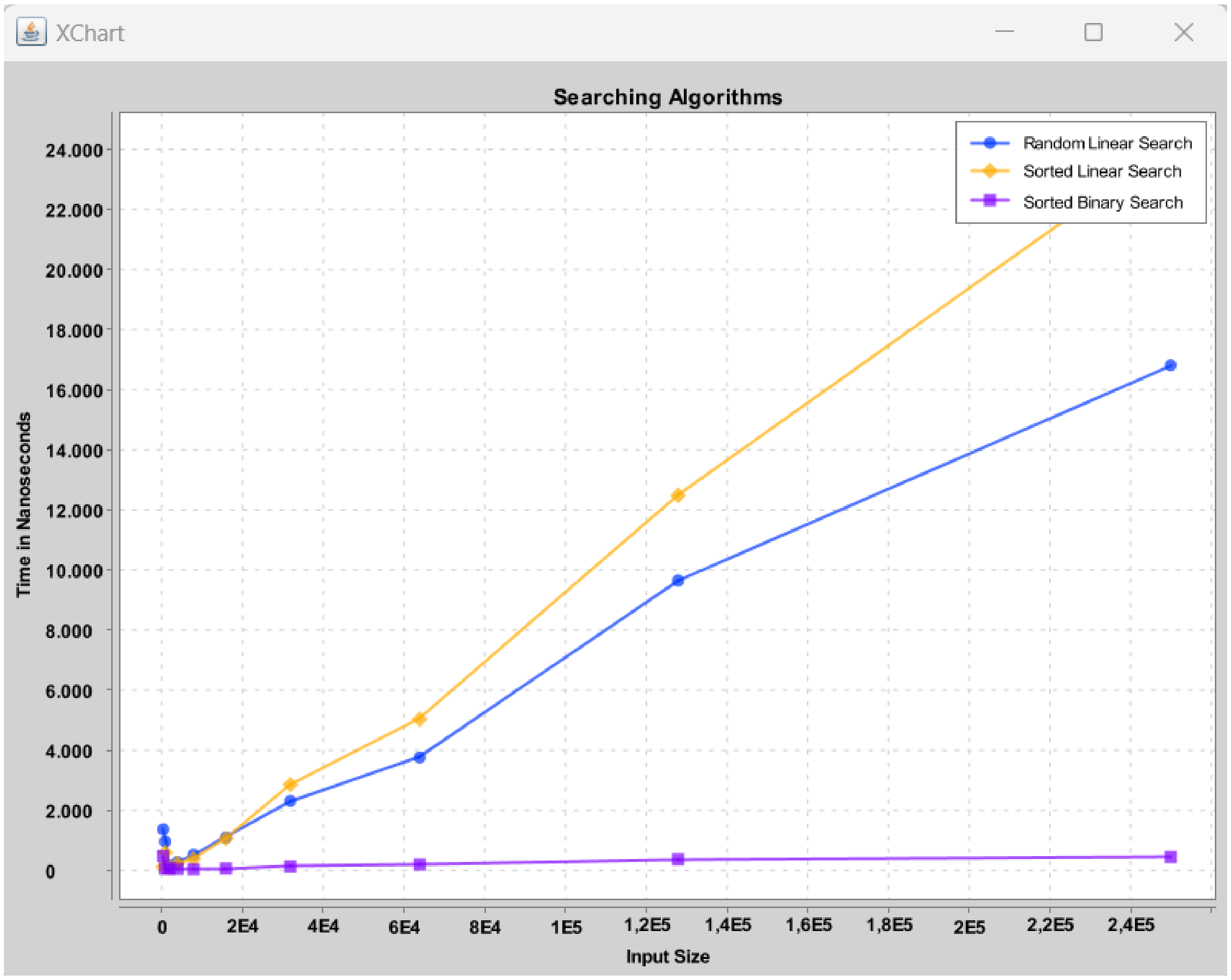

Figure 4: Searching Algorithm Tests