Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
Ahmet İkbal Kayapinar

Student Number:
b2210356043

1 Problem Definition

In this experiment, we compared three sorting algorithms with different complexities and two search algorithms with different complexity on data sets of different types and sizes such as random, sorted and reversely sorted, and observed how theoretical time complexities would yield results in practice.

2 Solution Implementation

2.1 Insertion Sort

```
public static void 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 = i - 1;
        }
        a[i + 1] = key;
    }
}
```

2.2 Merge Sort

```
private static void merge(int[] a, int[] aux, int lo, int mid, int hi)
{
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }
    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        if (i > mid) a[k] = aux[j++];
        else if (j > hi) a[k] = aux[i++];
        else if (aux[j] < aux[i]) a[k] = aux[j++];
        else a[k] = aux[i++];
    }
}

private static void sort(int[] a, int[] aux, int lo, int hi)
{
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
```

```java
    merge(a, aux, lo, mid, hi);
}

public static void mergeSort(int[] a)
{
    int[] aux = new int[a.length];
    sort(a, aux, 0, a.length - 1);
}
```

2.3 Counting Sort

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

    for (int i = 0; i < size; i++) {
        count[a[i]]++;
    }
    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;
}
```

2.4 Linear Search

```java
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

```java
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	1.6	0.0	1.2	4.7	18.8	70.8	277.5	1098.6	4029.1
Merge sort	0.0	0.0	0.0	0.0	1.6	0.0	1.5	4.7	10.9	18.7
Counting sort	264.0	178.2	107.2	88.6	87.7	90.1	89.6	90.2	91.7	94.7
	Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.6	0.0
Merge sort	0.0	0.0	0.0	0.0	0.0	0.0	1.5	1.6	3.1	6.3
Counting sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.5	0.0	96.5
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.0	0.0	0.0	1.6	3.1	15.7	57.6	228.7	831.8
Merge sort	0.0	0.0	0.0	0.0	0.0	0.0	1.6	1.5	3.2	6.2
Counting sort	90.1	89.8	88.6	90.2	88.6	88.0	90.0	91.6	91.6	90.1

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)	606.1	111.4	167.4	269.5	475.9	805.8	1613.0	3142.0	5857.9	10543.7
Linear search (sorted data)	48.8	80.9	97.2	250.7	354.2	777.2	1988.2	4247.5	8478.0	16306.3
Binary search (sorted data)	185.8	181.5	78.7	85.3	83.4	96.1	121.7	120.6	145.1	164.8

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	$O(k)$
Linear Search	$O(1)$
Binary Search	$O(1)$

3.1 Analysis and Discussion

In theory, the best case for insertion sort is that the array is sorted, the worst case is that the array is reverse sorted. Merge sort will have similar complexity in all cases. Counting sort will have similar complexity in all cases. The best case for linear search is that the searched element is at the beginning of the array, and the worst case is at the end. The best case for binary search is that the searched element is in the middle of the array, the worst case is that the element is at the end or at the beginning.

Although the plots are largely consistent with the theory, I detected some differences in the plots and my own additional observations. I tried all the outliers over and over again without any other action on the computer and got similar results with the plots.

In theory, reversely sorted array, which is the worst case for insertion sort, works faster than a randomly distributed array. In my opinion, the reason for this is hardware, not software. As we will see in later examples, the computer runs the algorithms that are run for the first time relatively slowly, even if the data set is different, and then speeds up. This may be due to the way processor caches work or another hardware reason. An example of this can be seen in the counting sort that works on a random data set. Although I ran the test more than ten times, the algorithm worked slowly on small data sets when it was first run, but as the data set grew, it got faster instead of slowing down.

Although counting sort is a less complex algorithm in theory, it is defeated by merge sort in almost every case in these tests. The reason for this is that the k value is very high in this data set, leaving the n value partially unimportant. One of the proofs of this is that counting sort is faster in sequential data until the largest input size is reached, but when the numbers grow, it lags behind merge sort.

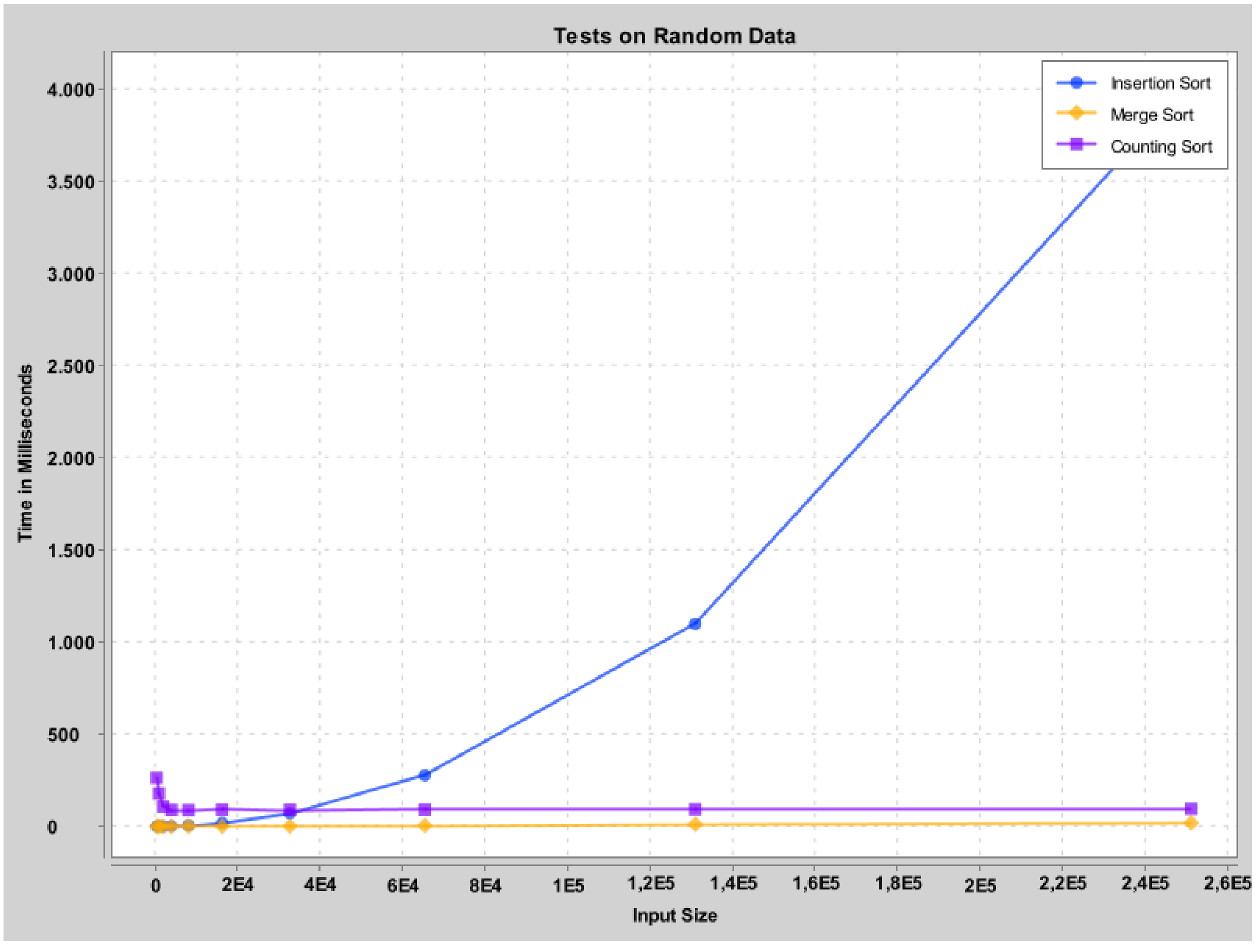

Figure 1: Tests on Random Data

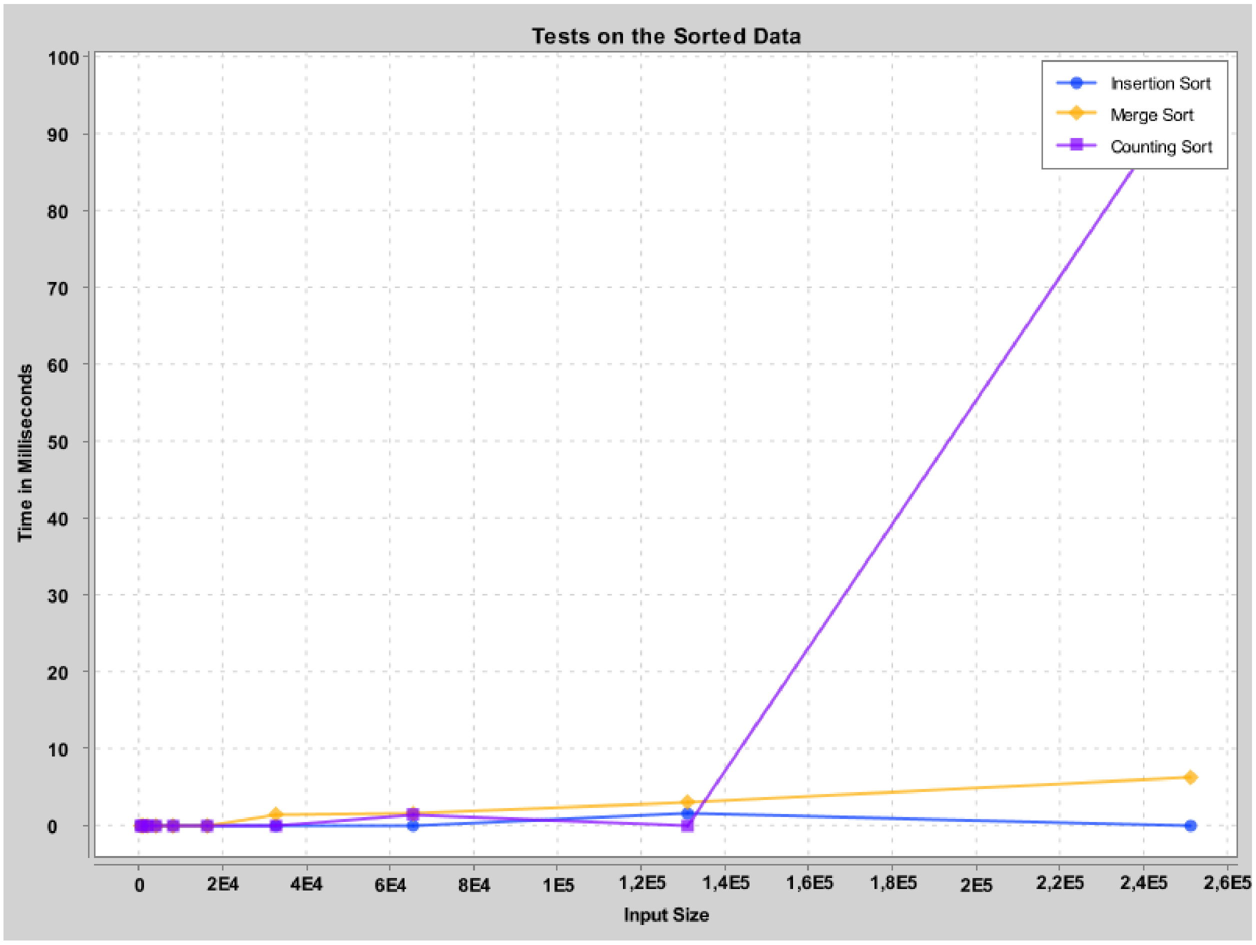

Figure 2: Tests on the Sorted Data

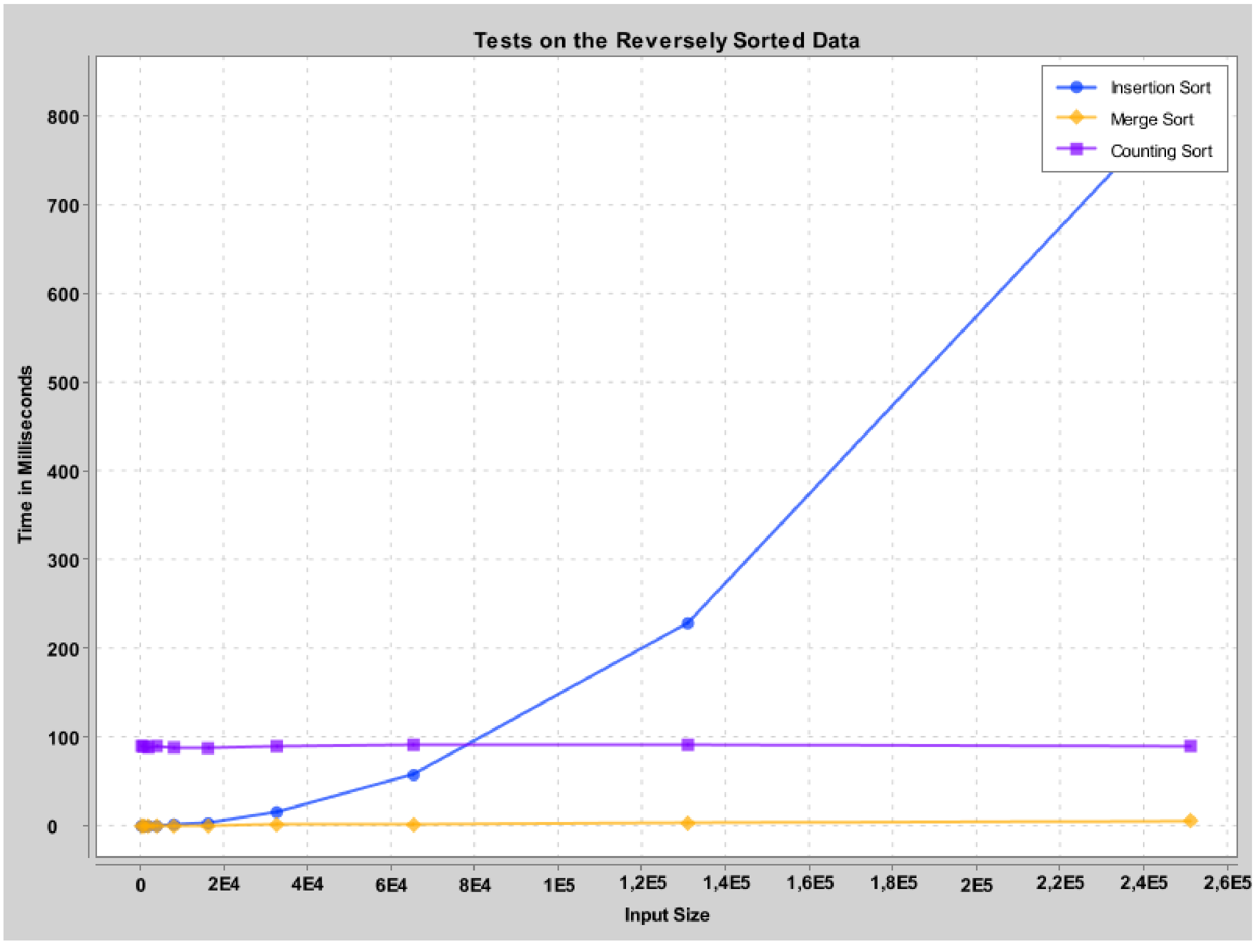

Figure 3: Tests on the Reversely Sorted Data

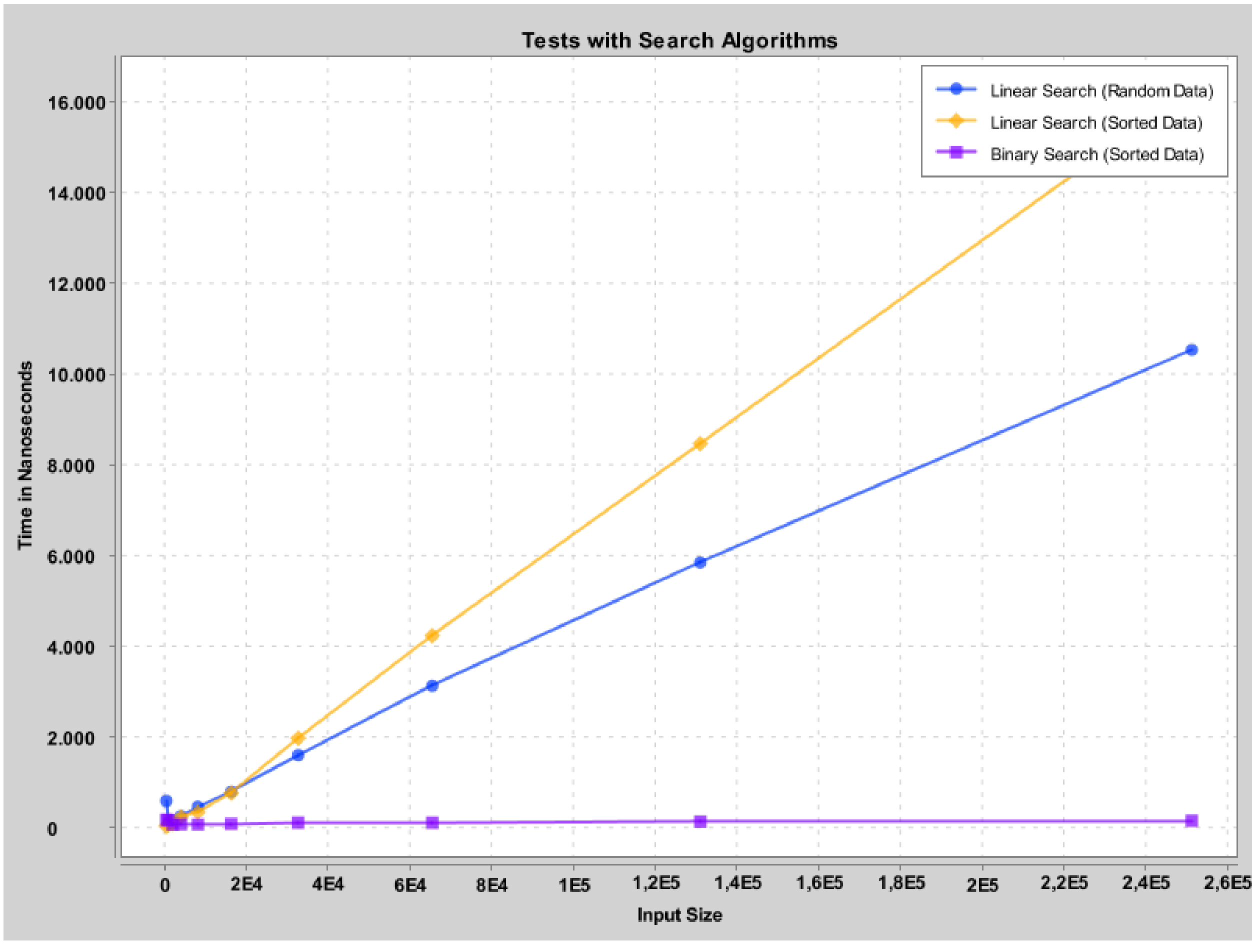

Figure 4: Tests with Search Algorithms