Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
Mustafa Kemal Öz

Student Number:
b2230356179

1 Problem Definition

Optimizing the efficiency of various algorithms, including search and merge algorithms, relies heavily on effective sorting. Moreover, the widespread availability of vast information through modern computing and the internet underscores the importance of efficient information retrieval. Evaluating the efficiency of sorting algorithms involves applying them to datasets of diverse sizes and characteristics. In this assignment, I tried to demonstrate the efficiency of different sorting and searching algorithms.

2 Solution Implementation

To demonstrate the varying efficiencies among algorithms, Java implementations of Insertion Sort, Merge Sort, and Counting Sort are developed. Additionally, Binary Search and Linear Search algorithms are implemented to showcase differences in searching methodologies. Subsequently, the algorithms undergo testing with diverse input sizes and data types, including sorted and random data sets. This process aids in pinpointing optimal and non-optimal scenarios and comparing them with theoretical expectations.

2.1 Insertion Sort

Example how to add Java code:

```java
public int[] sort(int[] data) {
        if (data == null) {
            throw new IllegalArgumentException("data must not be null.");
        }

        int n = data.length;
        int[] sorteddataay = data.clone();

        for (int j = 1; j < n; j++) {
            int key = sorteddataay[j];
            int i = j - 1;

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

2.2 Counting Sort

```java
public int[] sort(int[] data) {
        int max = data[0];

        for (int i = 1; i < data.length; i++) {
            if (data[i] > max) {
                max = data[i];
            }
        }

        return countingSort(data.clone(), max);
    }

    private int[] countingSort(int[] arr, int k) {
        int[] count = new int[k + 1];

        for (int num : arr) {
            count[num]++;
        }

        for (int i = 1; i <= k; i++) {
            count[i] += count[i - 1];
        }

        int[] output = new int[arr.length];

        for (int i = arr.length - 1; i >= 0; i--) {
            output[--count[arr[i]]] = arr[i];
        }

        System.arraycopy(output, 0, arr, 0, arr.length);

        return arr;
    }
```

3 Results, Analysis, Discussion

The test results are mostly compatible with the theoretical results, as seen in Tables 1-4. However, the Counting Sort algorithm exhibited a slightly suspicious spike when sorting sorted data with an input size of 250K. This anomaly was anticipated due to the significant change in the range of inputs. Additionally, the performance of linear search on random data appears somewhat erratic, which was also expected given the random nature of the test, resulting from the random generation of targets.

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	0.2	0.3	0.3	0.9	4.2	14.1	51.2	198.3	813.9	3438.1
Merge sort	0.1	0.1	0.2	0.4	0.7	1.5	2.7	6.1	11.8	23.7
Counting sort	79.1	65.0	65.4	66.7	65.6	65.4	65.5	65.8	66.8	79.5
	Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.6
Merge sort	0.1	0.1	0.1	0.4	0.5	1.0	1.6	3.4	5.2	11.2
Counting sort	0.0	0.0	0.0	0.1	0.0	0.1	0.1	0.2	0.6	74.7
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.2	0.7	2.8	7.2	26.1	102.6	413.0	1597.7	6118.1
Merge sort	0.0	0.0	0.0	0.1	0.2	0.5	1.2	3.1	5.3	10.8
Counting sort	80.7	66.9	69.1	67.2	66.8	66.7	67.0	68.9	68.3	70.7

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)	1917.4	1538.6	6046.6	444.6	971.5	1500.3	2582.2	4156.7	9371.2	12347.3
Linear search (sorted data)	62.7	107.2	126.5	335.1	498.2	1083.9	2235.3	4780.7	9315.4	19201.7
Binary search (sorted data)	207.7	84.5	107.5	120.6	121.4	126.2	163.4	118.3	191.0	244.0

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	$O(1)$

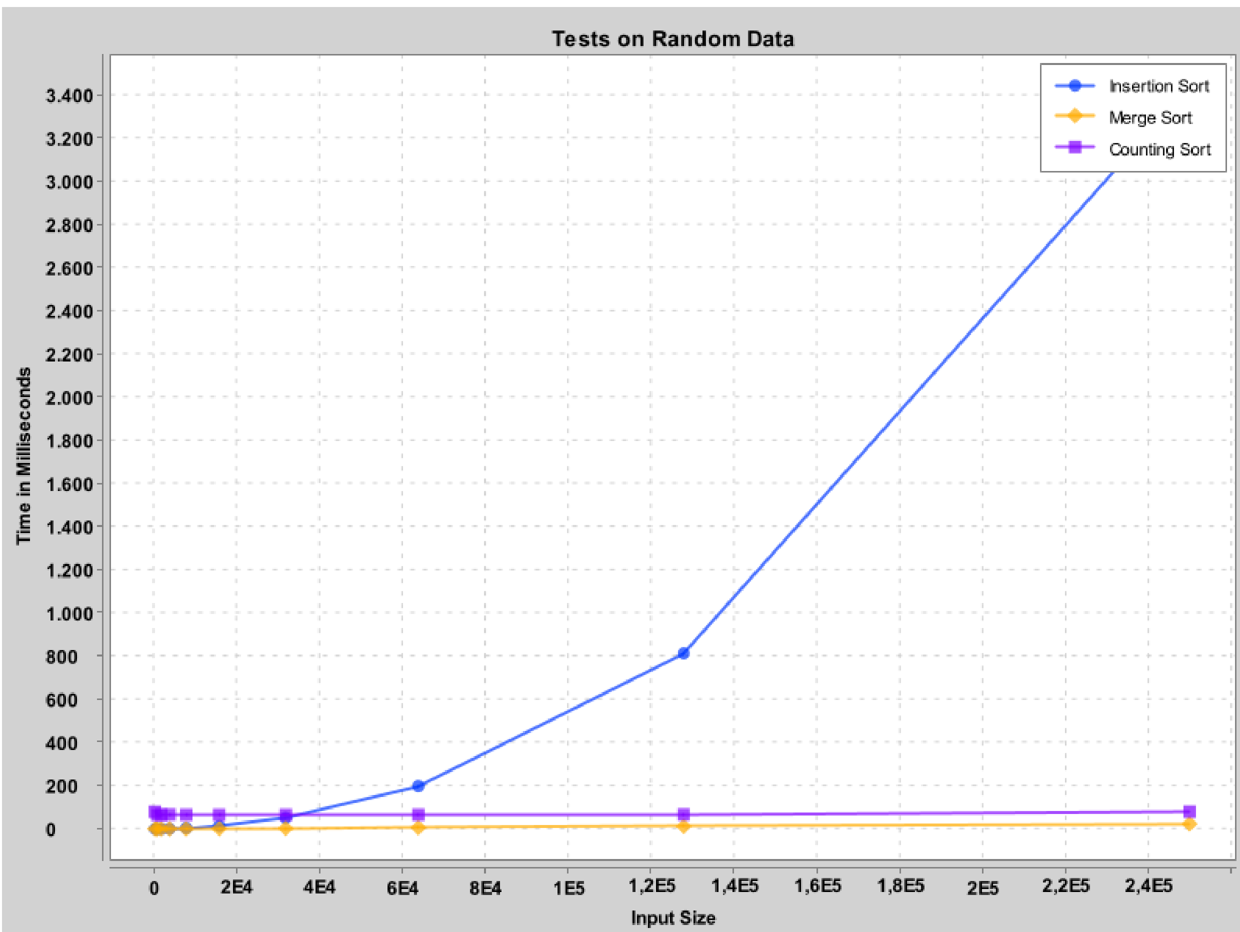

Figure 1: Random Data Sorting Graph

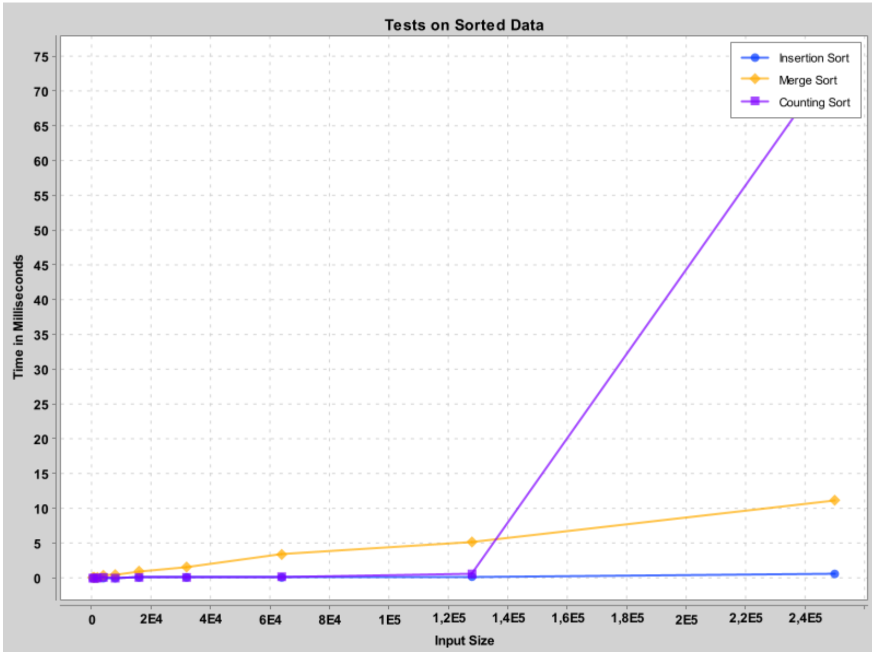

Figure 2: Sorted Data Sorting Graph

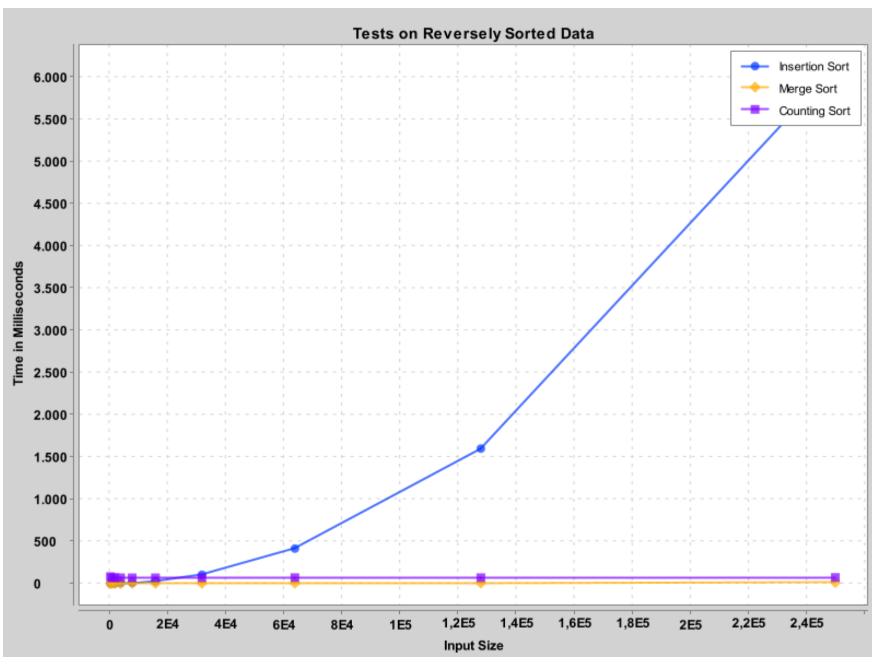

Figure 3: Reversely Sorted Data Sorting Graph

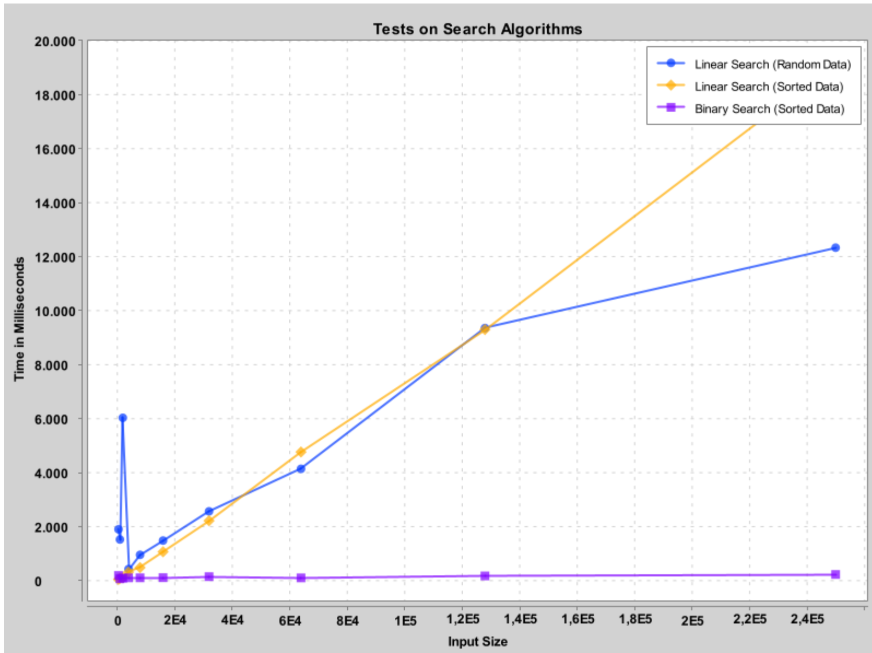

Figure 4: Searching for Randomly Generated Target Graph

4 Notes

The results of this experiment, along with the graphs in Figures 1-4, demonstrate the practical data's close resemblance to the theoretical data. Small deviations from the theoretical results are primarily attributed to the operating system's functioning and the Java Virtual Machine (JVM).

References

- https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/