Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
BORAN ÇAKIL

Student Number:
b2210356071

1 Problem Definition

In this project we are trying to compare several sorting and searching algorithms. We have a set of value and we will use our sorting and searching algorithms on it to test how the algorithms react with different cases.

2 Solution Implementation

We will use 3 sorting and 2 searching algorithms.

2.1 Sorting Algorithm 1: Counting Sort

```
1  public class CountingSort {
2
3      public static double[] sort(double[] array, int k) {
4          double count[] = new double[k + 1];
5          double output[] = new double[array.length];
6          int size = array.length;
7          for (int i = 0; i < size; i++) {
8              int j = (int) array[i];
9              count[j]++;
10         }
11         for (int i = 1; i < k + 1; i++) {
12             count[i] += count[i - 1];
13         }
14         for (int i = size - 1; i >= 0; i--) {
15             int j = (int) array[i];
16             count[j] = count[j] - 1;
17             output[(int) count[j]] = array[i];
18         }
19         return output;
20     }
21
22 }
```

2.2 Sorting Algorithm 2: Insertion Sort

```
24 public class InsertionSort {
25     public static void sort(double[] values) {
26         for (int j = 1; j < values.length; j++) {
27             double key = values[j];
28             int i = j - 1;
29             while (i >= 0 && values[i] > key) {
30                 values[i + 1] = values[i];
31                 i--;
```

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

2.3 Sorting Algorithm 3: Merge Sort

```
public class MergeSort {

    public static double[] sort(double[] array) {
        int n = array.length;
        if (n <= 1) {
            return array;
        }
        double[] left = new double[n / 2];
        double[] right = new double[n - n / 2];
        left = sort(left);
        right = sort(right);
        return merge(left, right);
    }
    private static double[] merge(double[] a, double[]b) {
        double[] c = new double[a.length + b.length];

         int i = 0, j = 0, k = 0;
         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;
    }
}
```

2.4 Search Algorithm 1: Linear Search

```java
public class LinearSearch {

    public static double search(double[] array, double x) {
        int size = array.length;
        for (int i = 0; i < size; i++) {
            if (array[i] == x) {
                return i;
            }
        }
        return -1;
    }
}
```

2.5 Search Algorithm 2: Binary Search

```java
public class BinarySearch {

    public static int binarySearch(double[] array, double x) {
        int low = 0;
        int high = array.length - 1;

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

        if (array[low] == x) {
            return low;
        } else if (array[high] == x) {
            return high;
        }

        return -1;
    }


}
```

3 Results, Analysis, Discussion

By the results I understood that every algorithm can be better for special cases. Like linear sort is does not waste any time if we have an array that is nearly sorted. And while testing I understood that each time we ran the program the values differ because there is so much going on at the background. Because of that when we want to measure the time precisely we have to run the algorithms multiple time and find the average time.

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	1.6	2.0	5.6	20.2	81.5	348.3	1563.4	6765.2
Merge sort	0.0	0.0	0.0	0.2	1.6	0.0	1.7	1.5	6.4	11.1
Counting sort	234.0	207.1	205.0	204.8	203.2	205.8	213.0	201.8	213.2	260.4
	Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.4	0.0	0.1
Merge sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.9	2.1	13.4
Counting sort	211.6	210.0	212.4	214.5	208.2	205.6	206.8	209.0	221.0	208.7
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.0	1.7	1.4	8.3	38.6	159.1	742.0	3190.2	12357.1
Merge sort	0.0	0.0	0.0	0.0	0.1	0.1	1.6	1.9	4.8	13.5
Counting sort	224.7	199.5	205.3	211.0	209.2	210.8	202.6	207.6	211.7	225.7

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)	747500.0	585200.0	925300.0	1963300.0	4385300.0	6227900.0	1.06722E7	1.92541E7	3.30402E7	5.20989E7
Linear search (sorted data)	261100.0	443400.0	929400.0	3240400.0	5012700.0	8823700.0	1.26335E7	2.53788E7	4.5848E7	7.44258E7
Binary search (sorted data)	571700.0	99800.0	113000.0	140500.0	166700.0	172500.0	229600.0	216100.0	272900.0	290300.0

Complexity analysis tables to complete (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	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Linear Search	$\Omega(1)$	$\Theta(n)$	$O(n)$
Binary Search	$\Omega(n)$	$\Theta(logn)$	$O(\log n)$

Table 4: Auxiliary space complexity of the given algorithms.

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

3 sorting algorithms used on randomly generated data.: Fig. 1.

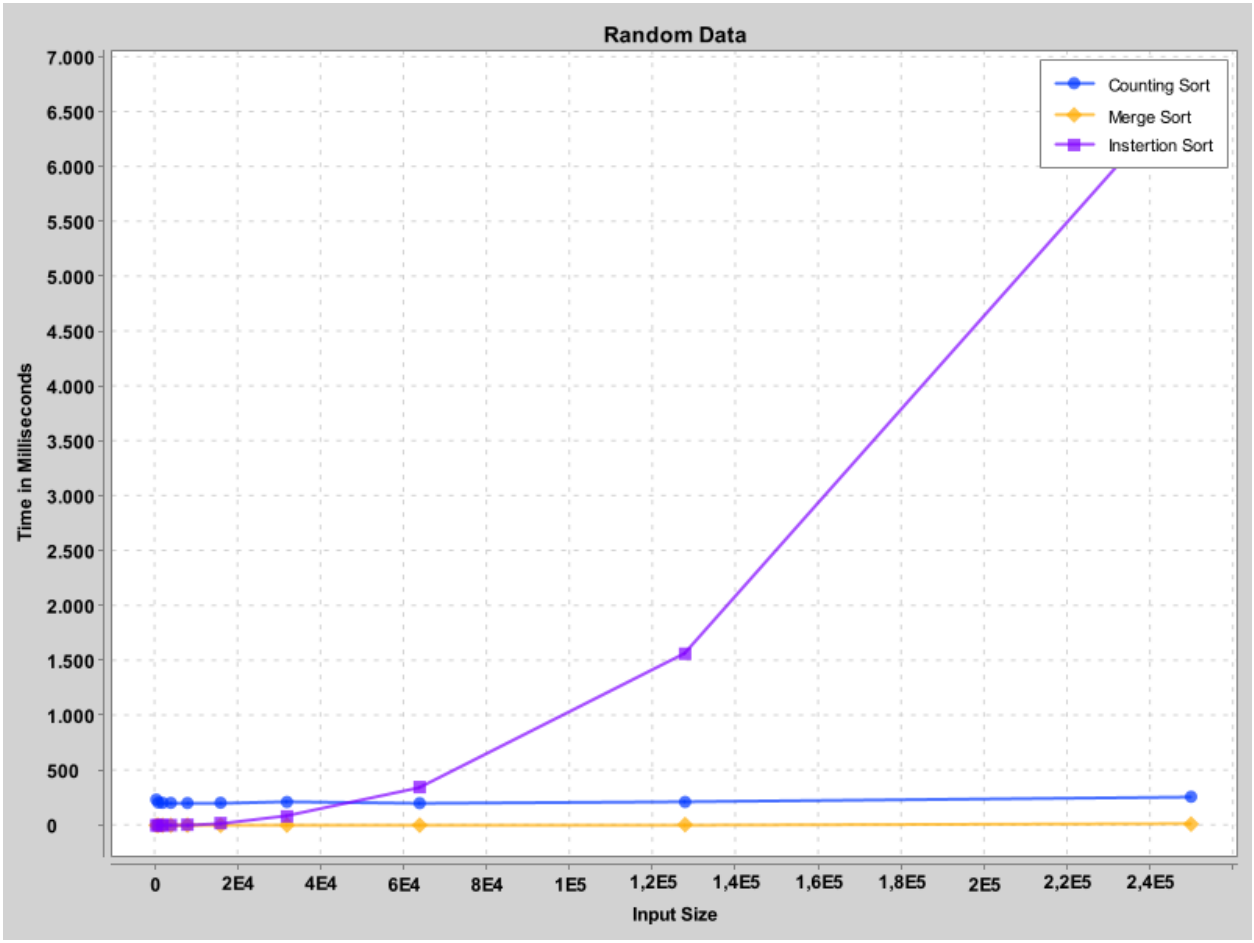

Figure 1: Sorting algorithms on randomly generated data

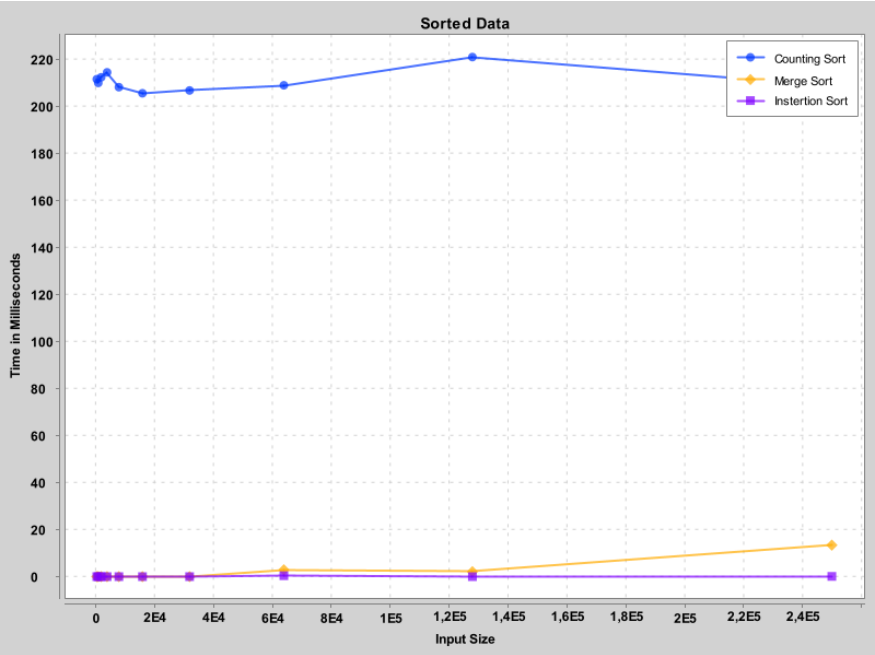

Figure 2: Sorting algorithms on sorted data

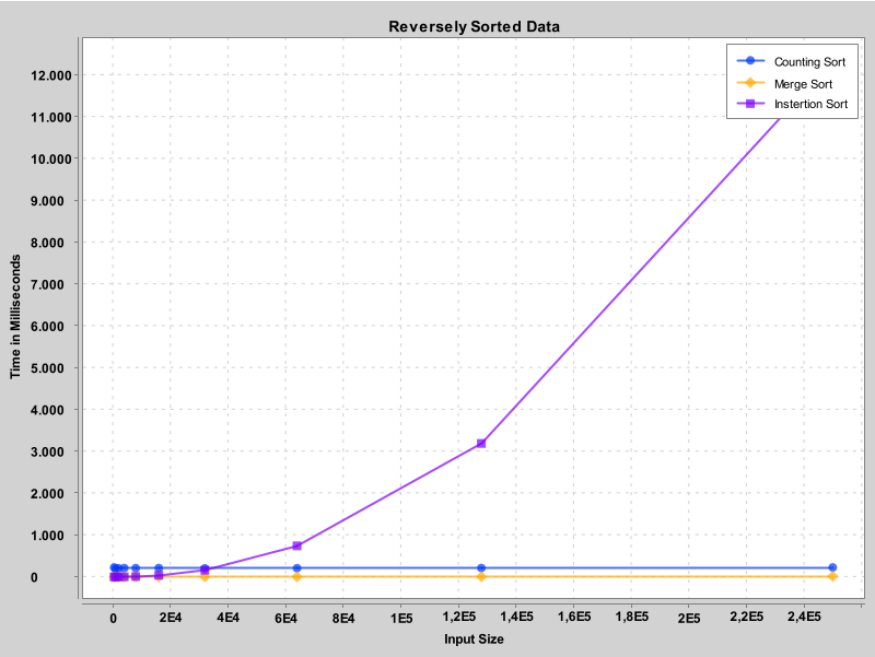

Figure 3: Sorting algorithms on reversely sorted data

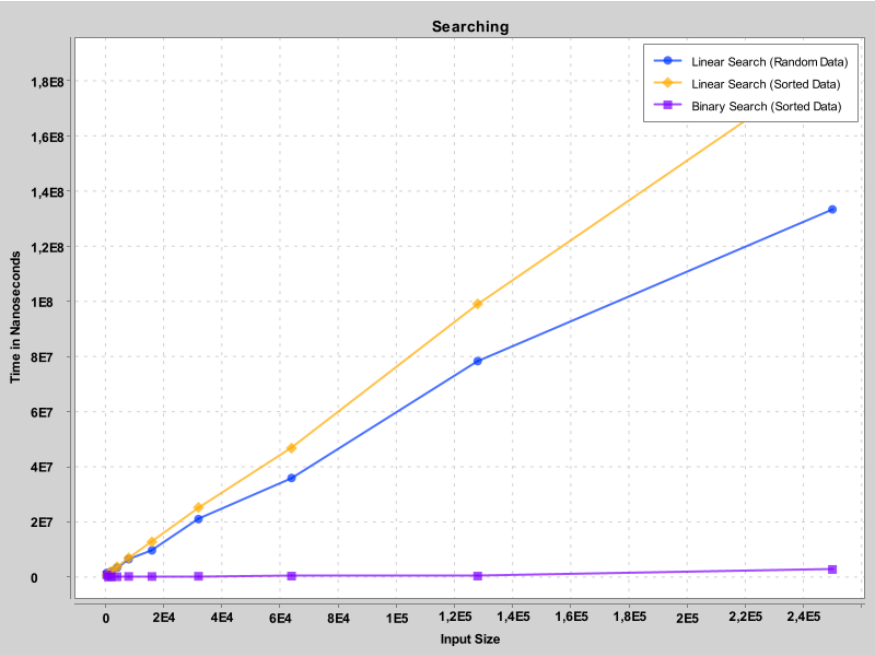

Figure 4: Searching algorithms on data

4 Notes

Algorithms react differently on different data sizes and types. It is important to use the right one for the right cause.

References

- https://www.geeksforgeeks.org