Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 21, 2024

Student name:
Cansu ASLAN

Student Number:
2210356079

1 Problem Definition

The assignment aims to analyze the runtime behaviors of sorting and searching algorithms across datasets of varying sizes to understand their computational complexities, efficiencies, and performance attributes. Algorithms will be categorized based on two main criteria: computational complexity (time) and space complexity. Computational complexity will be assessed in terms of best-case, worst-case, and average-case behaviors relative to dataset size. Space complexity will consider whether algorithms require additional auxiliary memory for sorting operations, with searching algorithms typically not needing extra memory. This structured analysis will provide insights into the behaviors and characteristics of the algorithms under examination.

2 Solution Implementation

This solution aims to analyze the performance of specified sorting and searching algorithms by comparing their running times using graphs generated with the XChart library.

Initially, a CSV file is opened, and the desired column is extracted into arrays using BufferedReader and Reader Classes. Sorting and searching algorithms are then implemented in Java classes based on provided pseudocodes.

In the experimental phase, sorting and searching are conducted using nested loops, with attention given to avoiding redundant sorting of the original array. Running time results are stored in 2D arrays for graph generation with XChart.

The implemented sorting algorithms include insertion sort, merge sort, and counting sort, ensuring adherence to their respective time complexities. The results are then utilized for graphical analysis to demonstrate the relationship between algorithmic running times and their theoretical complexities.

2.1 Sorting Algorithm 1

Insertion Sort:

```
1  public static void insertionSort(int[] array){
2      for(int i = 1; i <  array.length; i++){
3          int key = array[i];
4          int j = i-1 ;
5          while(j >= 0 && array[j] > key){
6              array[j + 1] = array[j];
7              j--;
8           }
9          array[j + 1] = key;
10     }
11 }
```

2.2 Sorting Algorithm 2

Merge Sort:

```
12 public static void mergeSort(int[] array, Integer low, Integer high){
13         if(low < high){
14             Integer mid = (low + high) / 2;
15             mergeSort(array, low, mid);
16             mergeSort(array, mid + 1, high);
17 
18             merge(array,low,mid,high);
19         }
20     }
21 
22     public static void merge(int[] array, int low, int mid, int high){
23         Integer numberOne = mid - low + 1;
24         Integer numberTwo = high - mid;
25 
26         Integer[] left = new Integer[numberOne];
27         Integer[] right = new Integer[numberTwo];
28 
29         for (int i = 0; i < numberOne;i ++){
30             left[i] = array[low + i];
31         }
32 
33         for(int j = 0; j < numberTwo; j++){
34             right[j] = array[mid + 1 + j];
35         }
36 
37         int i = 0, j = 0, k = low;
38 
39         while (i < numberOne && j < numberTwo){
40             if(left[i] <= right[j]){
41                 array[k] = left[i];
42                 i ++;
43             }else{
44                 array[k] = right[j];
45                 j++;
46             }
47             k++;
48         }
49         while (i < numberOne){
50             array[k] = left[i];
51             i++;
52             k++;
53         }
54         while (j < numberTwo){
55             array[k] = right[j];
56             j++;
```

```
            k++;
        }
    }
```

2.3 Sorting Algorithm 3

Counting Sort:

```
public static int[] countingSort(int[] input, int k){
        int[] count = new int[k+1];
        int size = input.length;
        int[] output = new int[size];
        for (int i = 0; i < size; i ++){
            int j = input[i];
            count[j] = count[j] + 1;
        }
        for (int i = 1; i <= k; i++){
            count[i] += count[i - 1];
        }
        for (int i = size - 1; i >= 0 ; i--){
            int j = input[i];
            count[j]--;
            output[count[j]] = input[i];
        }
        return output;
    }
```

2.4 Searching Algorithm 1

Linear Search:

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

2.5 Searching Algorithm 2

Binary Search:

```
88  public static int BinarySearch(int[] input, int x){
89          int low = 0;
90          int high = input.length - 1;
91          while (high - low > 1 ){
92              int mid = (high + low) / 2;
93              if (input[mid] < x ){
94                  low = mid +1 ;
95              }else{
96                  high = mid;
97              }
98          }
99          if (input[low] == x){
100             return low;
101         } else if (input[high]== x) {
102             return high;
103         }
104         return -1;
105     }
```

3 Results, Analysis, Discussion

A list of time values is provided for three sorting algorithms under development, across different inputs.

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	0.1	0.1	0.5	1.3	4.5	25.6	74.6	301.7
Merge sort	0.2	0.1	0.2	0.2	0.2	0.4	2.0	3.9	9.6	18.4
Counting sort	109.5	93.3	93.4	92.6	92.9	93.5	94.3	94.5	94.6	95.3
	Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.2	0.4
Merge sort	0.1	0.2	0.4	0.1	1.6	1.0	1.6	3.5	8.1	18.8
Counting sort	108.5	92.7	92.9	93.1	92.9	94.5	94.8	94.1	94.4	96.0
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.3	0.3	0.5	1.6	5.6	22.2	90.7	368.0	1452.1	5658.9
Merge sort	0.0	0.1	0.2	0.4	0.4	1.2	1.6	3.8	8.9	19.1
Counting sort	92.9	92.6	93.0	93.4	93.2	93.4	94.8	94.3	94.1	95.5

Running time test results for search algorithms are given in Table 2.

Complexity analysis tables to complete (Table 3 and Table 4):

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)	1904.9	1437.2	1024.6	647.4	866.6	1459.3	2829.9	4743.6	8682.0	16544.0
Linear search (sorted data)	1376.2	1555.3	795.9	666.1	883.2	1466.8	2811.9	4841.4	8706.9	16362.5
Binary search (sorted data)	951.3	622.2	1781.8	468.3	396.8	362.1	457.1	347.0	349.4	369.1

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

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	$O(1)$

4 Notes and Conclusion

In conclusion, the visualization of algorithmic performance through graphical representations offers invaluable insights into the efficiency and behavior of sorting algorithms such as Merge Sort, Insertion Sort, and Counting Sort. Through graph analysis, we can observe the values obtained for varying input sizes, facilitating a deeper understanding of their time complexities. Merge Sort, known for its stability and consistent performance, typically demonstrates a time complexity of $O(nlogn)$, as confirmed by our graphical analysis. Insertion Sort, while simple and intuitive, tends to exhibit a time complexity of $O(n^2)$, as reflected in our visualizations, making it less suitable for large datasets. Counting Sort, characterized by its linear time complexity $O(n+k)$, where k is the range of the input, showcases remarkable efficiency, especially for datasets with a limited range of values. By examining the shape of the graphs generated from our experiments, we can discern clear patterns and trends, aiding in the selection of the most appropriate sorting algorithm for specific tasks and input sizes. Thus, the combination of code implementation and visual analysis provides a comprehensive understanding of the performance characteristics of Merge Sort, Insertion Sort, and Counting Sort, enabling informed decision-making in algorithm selection.

A visual representation of sorting tests on random input: Fig. 1.

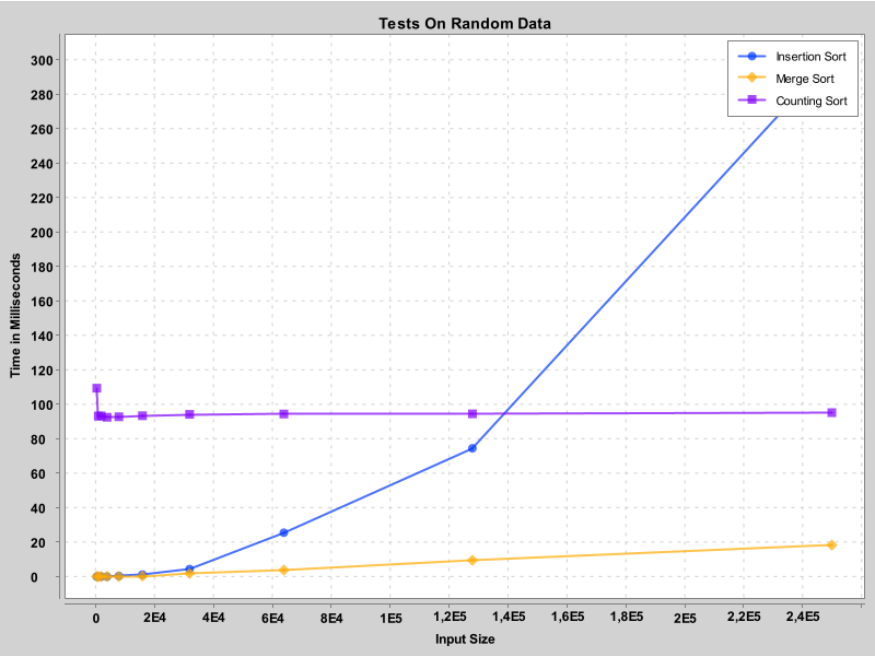

Figure 1: Plot of the functions.

A visual representation of sorting tests on sorted inputs: Fig. 2.
A visual representation of sorting tests on reversely sorted inputs: Fig. 3.
A visual representation of searching tests on random and sorted inputs: Fig. 4.

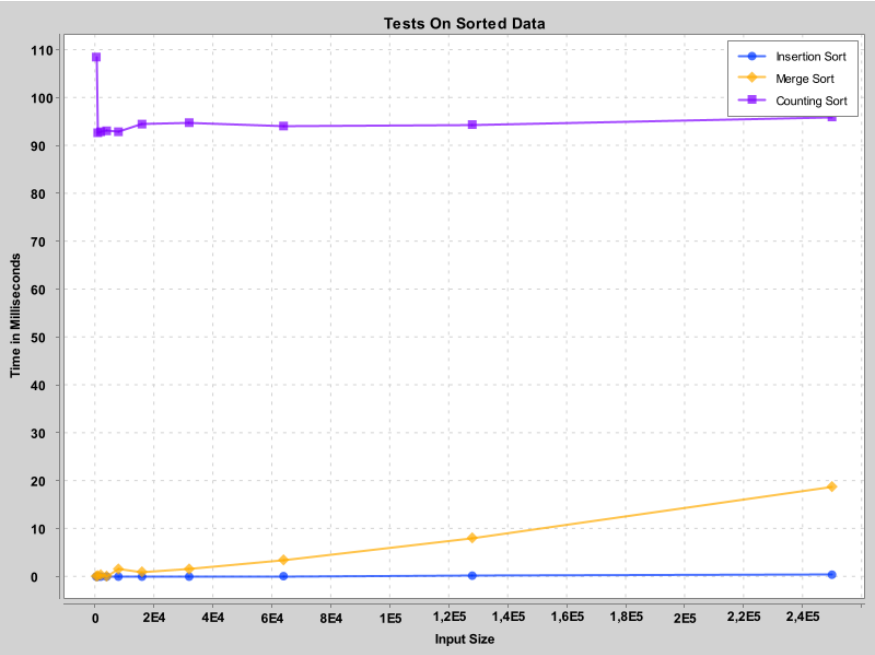

Figure 2: A smaller plot of the functions.

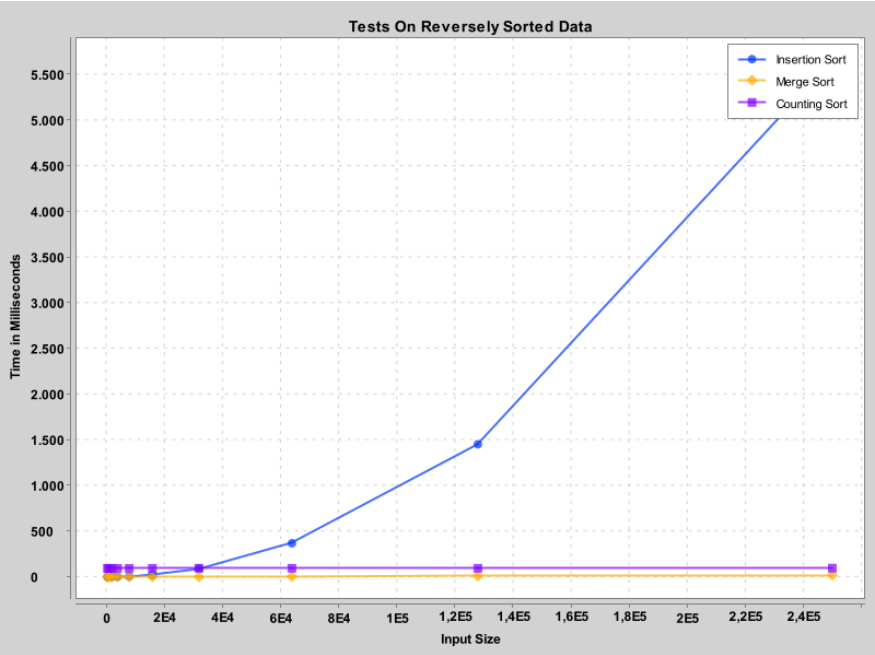

Figure 3: Plot of the functions.

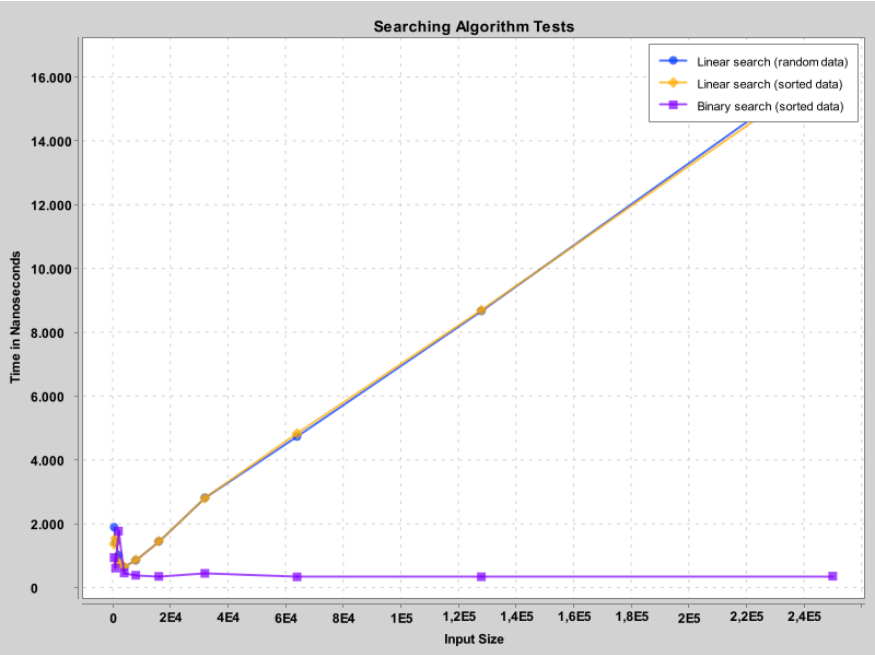

Figure 4: Plot of the functions.

References

- https://www.geeksforgeeks.org/merge-sort/
- https://www.geeksforgeeks.org/insertion-sort/
- https://www.geeksforgeeks.org/counting-sort/
- https://www.geeksforgeeks.org/merge-sort/
- https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/
- https://stackoverflow.com/questions/4300653/conversion-of-nanoseconds-to-milliseconds-and-nanoseconds-999999-in-java
- https://www1.cmc.edu/pages/faculty/aaksoy/latex/latexfive.html