Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 17, 2024

Student name:
Ethem Ateş

Student Number:
b2210356076

1 Problem Definition

The objective of this assingment is to analyze the running times of some special sorting and searching algorithms on datasets of different sizes. Through this analysis, we aim to understand their computational complexities and determine their efficiencies and performance characteristics.

In this assignment, we goal to categorise given sorting and looking algorithms primarily based on 2 criteria:

Computational Complexity(Time): Determining the best, worst, and average-case behavior concerning the size of the dataset. Table 1 illustrates a comparison of computational complexity for some well-known sorting and searching algorithms.

Space Complexity: Some algorithms may require additional auxiliary memory for sorting operations but some algorithms don't. Searching algorithms do not require additional memory.

2 Solution Implementation

The purpose of this assignment is to calculate the running times of the specified sorting and searching algorithms and compare them using graphs that we create using XChart library. First of all I opened the csv file with using BufferedReader and Reader Classes and skipped unnecessary columns and took the wanted column. Since we will perform the operations on this column, I stored the data in this column into arrays using a while loop. Then implemented the sorting and searching algorithms in Java classes named as "searchclass" and "sortclass" thanks to given pseudocodes in assignment PDF. In the experiment part I have implemented classical insertion sort, merge sort, and counting sort algorithms. While creating them, I paid attention to their pseudocodes and ensured they have the correct time complexities. And then, I add the given starter code to Main class. After I get the results of the running time of the algorithms, added all the results to 2D arrays which is going to be needed in creating graphs in XChart.

2.1 Insertion Sort

My insertion sort code:

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

2.2 Merge Sort

My merge sort code:

```
12 public static void mergesort(int[] array){
13         if(array.length < 2){ // base case durumu
14             return;
15         }
16         int array_length = array.length;
17         int middleofarray = (int) array.length/2;
18         int[] leftarray = new int[middleofarray];
19         int[] rightarray = new int[array_length - middleofarray];
20         for(int i = 0; i < array.length; i++){
21             if(i<middleofarray){
22                 leftarray[i] = array[i];
23             }
24             else{
25                 rightarray[i-middleofarray] = array[i];
26             }
27         }
28         mergesort(leftarray);
29         mergesort(rightarray);
30         merge(leftarray, rightarray,array);
31     }
32     public static void merge(int[] left, int[] right, int[] array){
33         int minimum_number = Math.min(right.length, left.length);
34         int arrayindex = 0;
35         int i = 0;
36         for(; i< minimum_number; i++){
37             if(left[i]>right[i]){
38                 array[arrayindex] = right[i];
39                 arrayindex++;
40             }
41             if(left[i]<right[i]){
42                 array[arrayindex] = left[i];
43                 arrayindex++;
44             }
45         }
46         while(i < right.length) {
47             array[arrayindex] = right[i];
48             i++;
49             arrayindex++;
50         }
51         while(i < left.length){
52             array[arrayindex] = left[i];
53             i++;
54             arrayindex++;
55         }
56
```

```
    }
```

2.3 Counting Sort

My counting sort code:

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

        return output;
    }
```

3 Results, Analysis, Discussion

Below are the time values for the three sorting algorithms being worked on, with various 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	0.50	0.02	0.04	0.13	0.60	2.24	8.90	33.78	140.60	590.46
Merge sort	0.27	0.09	0.15	0.25	0.49	1.12	1.99	4.84	10.57	20.68
Counting sort	390.10	110.79	109.15	111.41	109.23	109.69	109.05	114.29	120.13	111.78
	Sorted Input Data Timing Results in ms									
Insertion sort	0.01	0.001	0.003	0.006	0.01	0.02	0.04	0.10	0.21	0.34
Merge sort	0.02	0.05	0.09	0.18	0.41	0.87	1.87	3.72	7.52	14.89
Counting sort	234.55	110.54	110.35	107.52	110.26	115.42	108.68	108.70	113.37	121.84
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.03	0.017	0.07	0.26	1.05	4.22	19.17	68.29	279.94	1090.21
Merge sort	0.03	0.04	0.07	0.24	0.36	0.84	1.60	3.51	7.81	15.09
Counting sort	116.18	114.85	108.51	114.30	108.58	110.61	107.82	109.44	110.57	234.33

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)	2009.6	1678.3	214.8	402.2	652.4	1281.0	2661.6	5032.4	9616.8	19166.6
Linear search (sorted data)	533.2	192.5	204.4	371.0	649.0	1288.4	2352.4	4877.7	9605.9	19182.5
Binary search (sorted data)	328.6	157.2	136.5	86.7	84.7	104.6	109.4	107.5	111.4	157.9

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	$O(1)$

A visual representation of XChart with random values from different inputs: Fig. 1.

A visual representation of XChart with sorted values from different inputs: Fig. 2.

A visual representation of XChart with reversed values from different inputs: Fig. 3.

A visual representation of XChart with search values from different inputs: (I wrote the name of the chart wrong) Fig. 4.

4 Notes and Conclusion

By visualizing the performance of algorithms through graphs, we were able to observe the values they obtained for different input sizes. While code implementation allows us to automatically estimate the time complexities of algorithms, seeing the values obtained for different inputs on graphs provides a clearer understanding. The shape of the graphs gives us a visual representation of the time complexities, making it easier to discern patterns and trends.

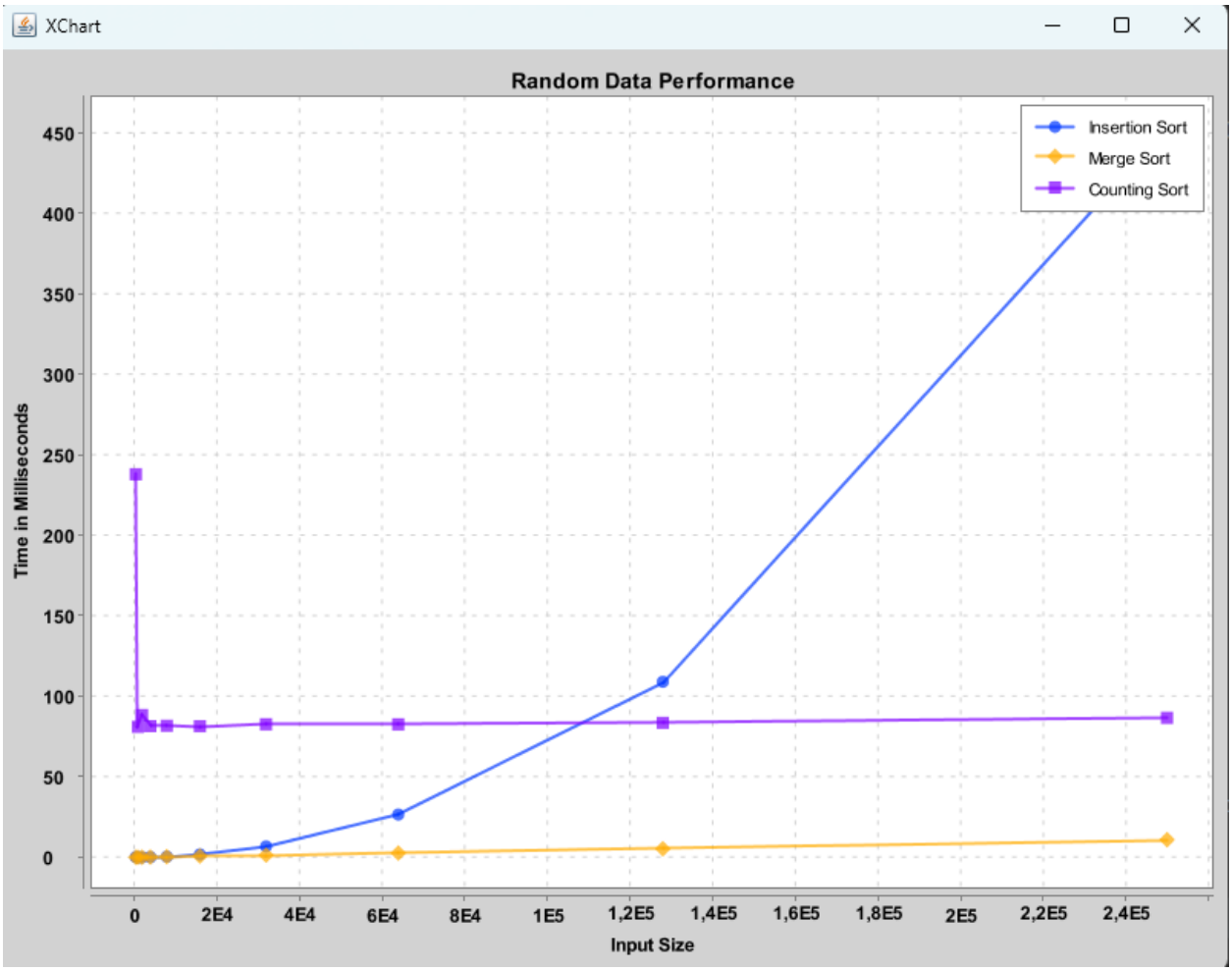

Figure 1: Plot of the functions.

References

- https://www.geeksforgeeks.org/insertion-sort/
- https://www.geeksforgeeks.org/counting-sort/?ref=shm
- https://www.geeksforgeeks.org/merge-sort/?ref=shm
- https://www.geeksforgeeks.org/linear-search/?ref=shm
- https://www.geeksforgeeks.org/binary-search/?ref=shm

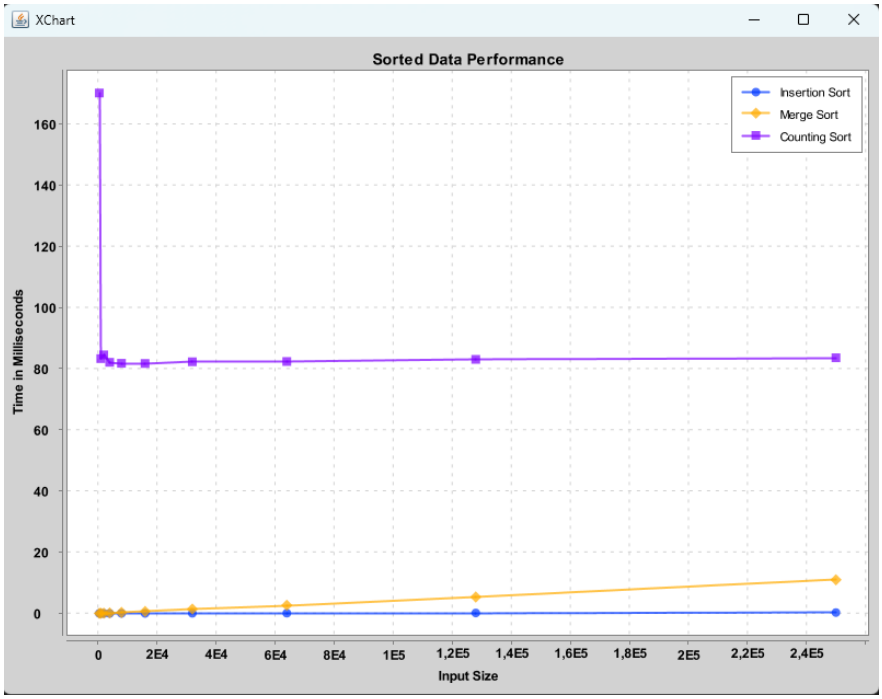

Figure 2: A smaller plot of the functions.

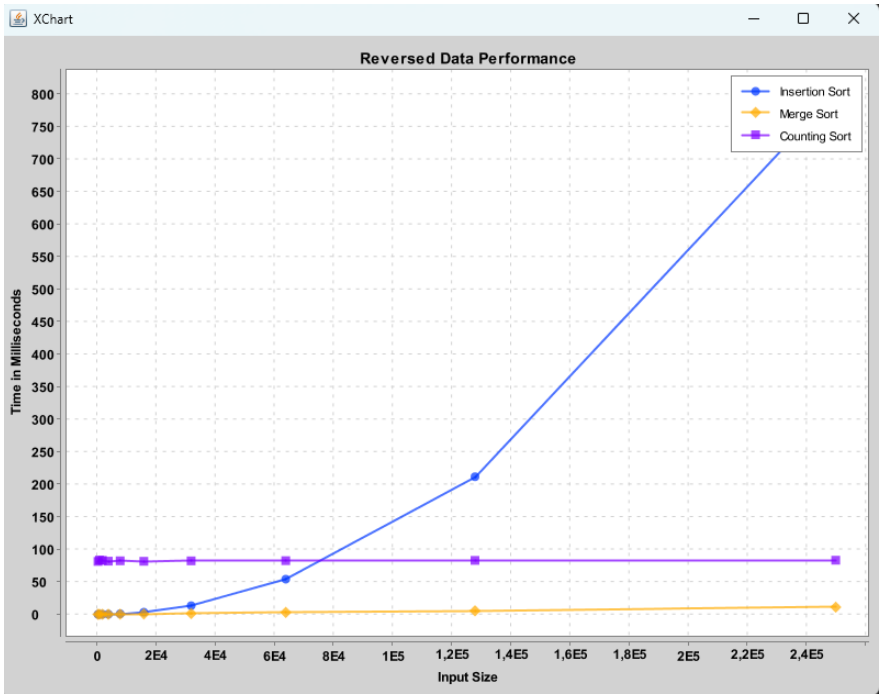

Figure 3: A smaller plot of the functions.

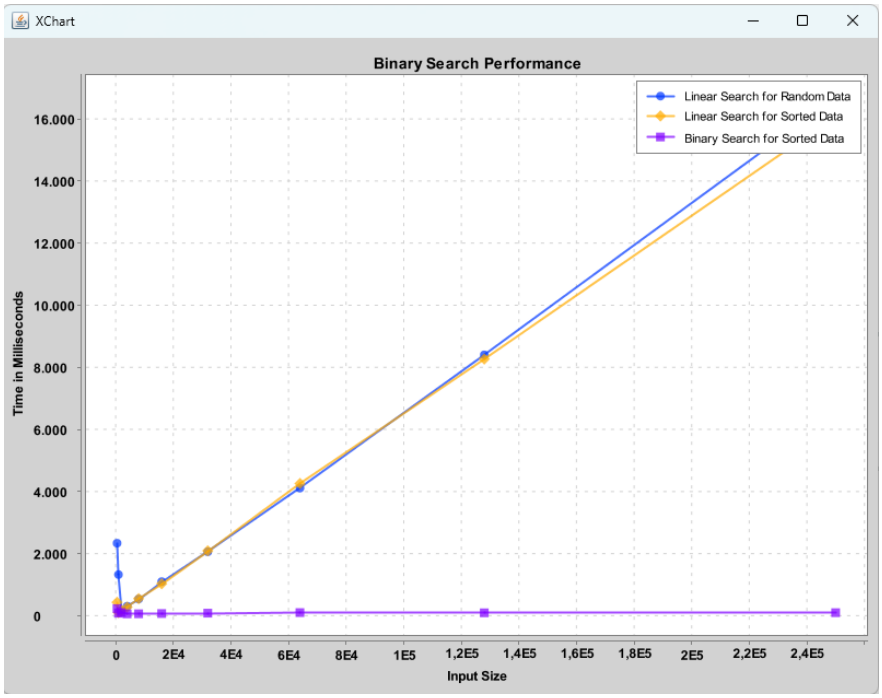

Figure 4: A smaller plot of the functions.