Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2022 Spring

Programming Assignment 1

March 9, 2022

Student name:
Oğuzhan DENİZ

Student Number:
b21946022

1 Problem Definition

We are trying to figure out that if these sort algorithms' running time is changing with input size or not, and if yes, what is the time complexity of that algorithm? We have 4 sort algorithms, 2 of them is comparison based (insertion, merge) and other 2 is non-comparison based (pigeonhole, counting).

2 Solution Implementation

First, we need to implement our sorting algorithms, then we have to sort our data 10 times for each input size and take average sorting time of 10 sorts.

2.1 Insertion Sort

```
1  public class InsertionSort {
2
3      public static void insertionSort(int arr[])
4      {
5          for(int i = 1; i < arr.length; ++i)
6          {
7              int key = arr[i];
8              int j = i - 1;
9
10             while(j >= 0 && arr[j] > key)
11             {
12                 arr[j+1] = arr[j];
13                 j -= 1;
14             }
15             arr[j+1] = key;
16         }
17     }
18 }
```

The best case for this algorithm is Ω(n), which is the input is sorted. Because for loop always active without restrictions but if arr[j] is less than key in every input (sorted array), inner while loop will never start.
The average and worst cases for this algorithm is $\Theta(n^2)$ and $O(n^2)$, because there is loop inside loop, which means we are traversing our array completely for each item in array.
This algorithm is not using extra arrays, so it's auxiliary space complexity is O(1).

2.2 Merge Sort

```
19 public class MergeSort {
20
21     public static void merge(int arr[], int left, int middle, int right)
```

```
22      {
23          int n1 = middle - left + 1;
24          int n2 = right - middle;
25
26          int leftArr[] = new int[n1];
27          int rightArr[] = new int[n2];
28
29          for(int i = 0; i < n1; ++i)
30          {
31              leftArr[i] = arr[left + i];
32          }
33          for(int i = 0; i < n2; ++i)
34          {
35              rightArr[i] = arr[middle + 1 + i];
36          }
37
38          int i = 0, j = 0;
39          int k = left;
40
41          while(i < n1 && j < n2)
42          {
43              if(leftArr[i] <= rightArr[j])
44              {
45                  arr[k] = leftArr[i];
46                  i++;
47              }
48              else
49              {
50                  arr[k] = rightArr[j];
51                  j++;
52              }
53              k++;
54          }
55          while(i < n1)
56          {
57              arr[k] = leftArr[i];
58              i++;
59              k++;
60          }
61          while(j < n2)
62          {
63              arr[k] = rightArr[j];
64              j++;
65              k++;
66          }
67      }
68
69      public static void sort(int arr[], int left, int right)
```

```
70      {
71          if (left < right)
72          {
73              int middle = left + (right - left) / 2;
74
75              sort(arr, left, middle);
76              sort(arr, middle + 1, right);
77
78              merge(arr, left, middle, right);
79          }
80      }
81  }
```

All cases for this algorithm is O(nlogn), which means there is no effect on this algorithm whether array is sorted or not.
However, this algorithm uses extra arrays for each item in array, so it's leads to O(n) auxiliary space complexity.

2.3 Pigeonhole Sort

```
82  import java.util.Arrays;
83
84  public class PigeonholeSort {
85
86      public static void pigeonholeSort(int[] arr)
87      {
88          int min = Arrays.stream(arr).min().getAsInt();
89          int max = Arrays.stream(arr).max().getAsInt();
90          int range = max - min + 1;
91
92          int[] hole = new int[range];
93          Arrays.fill(hole, 0);
94
95          for(int i = 0; i < arr.length; i++)
96          {
97              hole[arr[i] - min]++;
98          }
99          int index = 0;
100
101         for(int i = 0; i < range; i++)
102         {
103             while(hole[i]-- > 0)
104             {
105                 arr[index++] = i + min;
106             }
107         }
108     }
```

```
}
```

This algorithm is different than upper 2 algorithms, because it requires no comparison between items. It is independent whether array is sorted or not but dependent on array size and range of items (max - min).
So, it's best-average-worst time complexities are all O(n+N) (n is size of array, N is range) and it's space complexity is also O(n+N).

2.4 Counting Sort

```java
import java.util.Arrays;

public class CountingSort {

    public static void countingSort(int[] arr)
    {
        int max = Arrays.stream(arr).max().getAsInt();
        int min = Arrays.stream(arr).min().getAsInt();

        int range = max - min + 1;

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

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

This algorithm is similar to pigeonhole, but differs at some points. It requires no comparison, but needs extra space. It's average case is O(n+N) (n is size of array, N is range), best case is O(n) if all elements are same (so best case is linear) and worst case is approximately O(N) if range of

array is so big.
Auxiliary space complexity for this algorithm is O(N) which is range of array.

3 Results, Analysis, Discussion

I used miliseconds for random data tests and used nanoseconds for sorted and reversely sorted tests, results were more understandable in that way.

3.1 Random Data

Sort operation on randomly distributed data

Table 1: Results of the running time tests performed on the random data of varying sizes (in ms).

Algorithm	Input Size									
	512	1024	2048	4096	8192	16384	32768	65536	131072	251281
Insertion sort	0.3	0.5	0.8	1.4	5.5	23.2	88.1	335.1	1396.9	5821.9
Merge sort	0.2	0.1	0.3	0.6	0.4	1.2	2.7	5.7	11.9	24.4
Pigeonhole sort	132.8	113.0	111.7	111.8	112.0	111.5	111.7	112.4	113.6	115.2
Counting sort	77.0	75.2	75.6	75.5	76.1	76.4	77.0	78.9	82.3	85.8

Complexity analysis tables to complete:

Table 2: 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	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
Pigeonhole Sort	$\Omega(n + N)$	$\Theta(n + N)$	$O(n + N)$
Counting Sort	$\Omega(n)$	$\Theta(n + N)$	$O(N)$

Table 3: Auxiliary space complexity of the given algorithms.

Algorithm	**Auxiliary Space Complexity**
Insertion Sort	$O(1)$
Merge Sort	$O(n)$
Pigeonhole Sort	$O(n + N)$
Counting Sort	$O(N)$

n is size of array, N is range of array

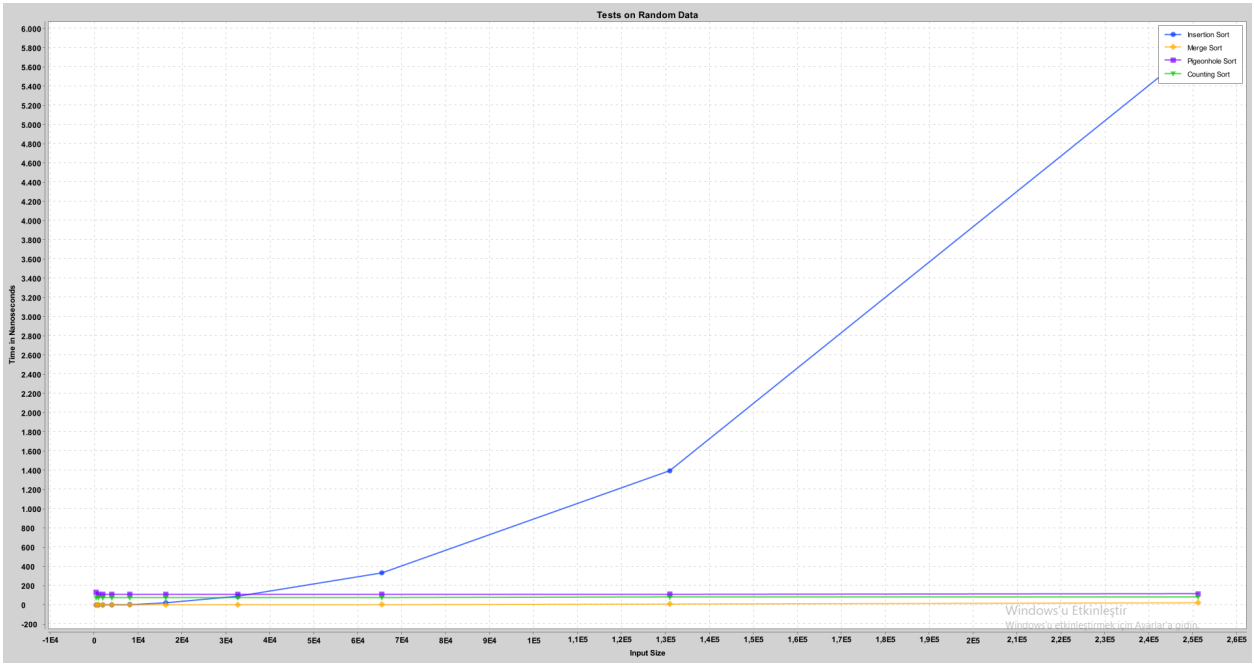

Figure 1: Tests on Random Data.

3.2 Sorted Data

Sort operation on sorted data

Table 4: Results of the running time tests performed on the sorted data of varying sizes (in ns).

Algorithm	Input Size									
	512	1024	2048	4096	8192	16384	32768	65536	131072	251281
Insertion sort	932610.0	24620.0	49590.0	85430.0	41430.0	80480.0	146900.0	303900.0	592110.0	781720.0
Merge sort	15060.0	30710.0	62850.0	134040.0	268410.0	595520.0	1210500.0	3291800.0	5207180.0	1.248374E7
Pigeonhole sort	751330.0	79240.0	140900.0	114160.0	124710.0	256440.0	446830.0	1506390.0	1958060.0	1.777121E8
Counting sort	84010.0	52330.0	100990.0	79250.0	88920.0	136260.0	290490.0	467950.0	1652860.0	9.208074E7

3.3 Reversely Sorted Data

Sort operation on reversely sorted data

Table 5: Results of the running time tests performed on the reversely sorted data of varying sizes (in ns).

Algorithm	Input Size									
	512	1024	2048	4096	8192	16384	32768	65536	131072	251281
Insertion sort	562340.0	554910.0	582090.0	2381820.0	9393600.0	4.137859E7	1.7160687E8	6.9087258E8	2.84696239E9	1.056338954E10
Merge sort	17430.0	34170.0	67400.0	147370.0	284310.0	613590.0	1254380.0	2660780.0	6541610.0	1.209975E7
Pigeonhole sort	1037280.0	388660.0	5825550.0	1.832607E7	2.228804E7	6.739376E7	1.0711644E8	1.138306E8	1.1657798E8	1.1745236E8
Counting sort	270130.0	210260.0	2428410.0	1.056735E7	1.443801E7	3.617166E7	7.417094E7	8.15517E7	8.337775E7	8.436759E7

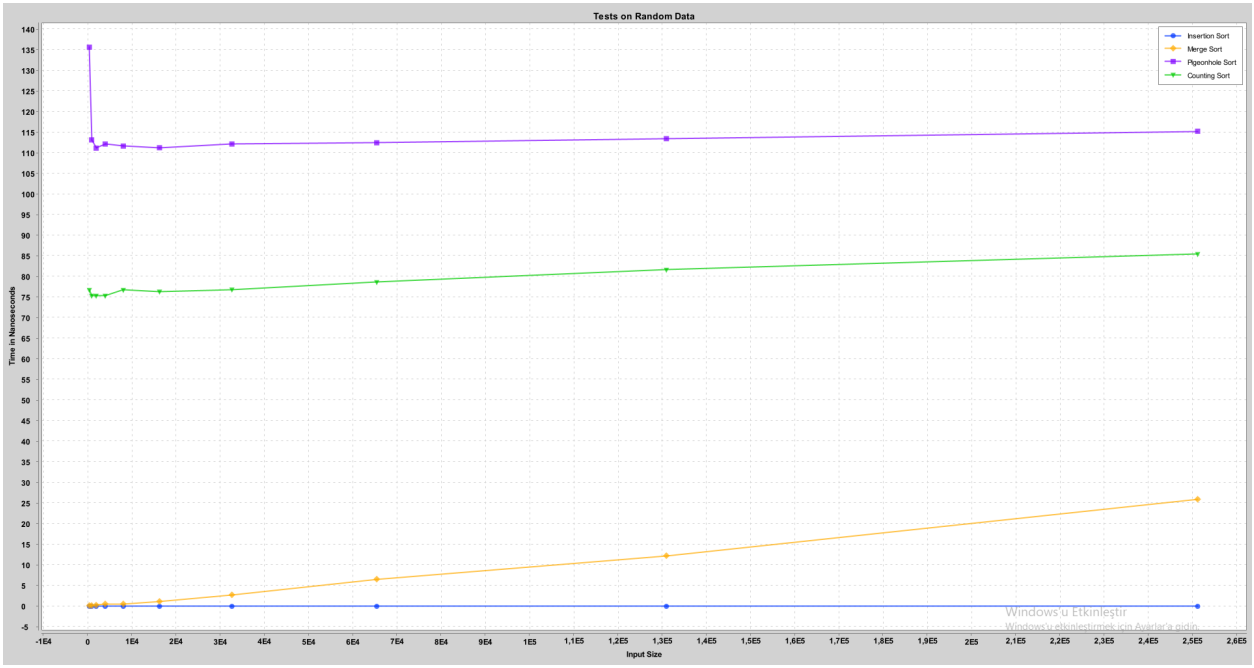

Figure 2: Tests on Random Data without Insertion Sort.

4 Notes

I got some problems during coding part, testing part and preparing the report part, i will try to explain these in this section for not to confuse you.

- This is my first time using Latex and Overleaf, some figures and tables are not in the place that i wanted, i couldn't figure it out. But I guess it's not big deal, they're still understandable.
- My results on sorted data are a bit unexpected, actually i wasn't sure what is the correct result on this part. I wish you could share the expected results, because I'm pretty sure my algorithm and testing process' are pretty accurate.
- In Figure 2, I got the cache problem which you mention about at Piazza but i couldn't able to fix the problem, I hope it's not a big problem because everything you wanted is working.
- To run my code properly, you need to do following things:
  If you want to do it on random array, you need to comment out red and green area in Figure 5, also change comment on orange area in Figure 6 (use stringToInt method instead of getArray method).
  If you want to do it on sorted array, just comment out green area in Figure 5, also use getArray method in orange area in Figure 6.
  If you want to do it on reversely sorted array, leave it like Figure 5 and 6.

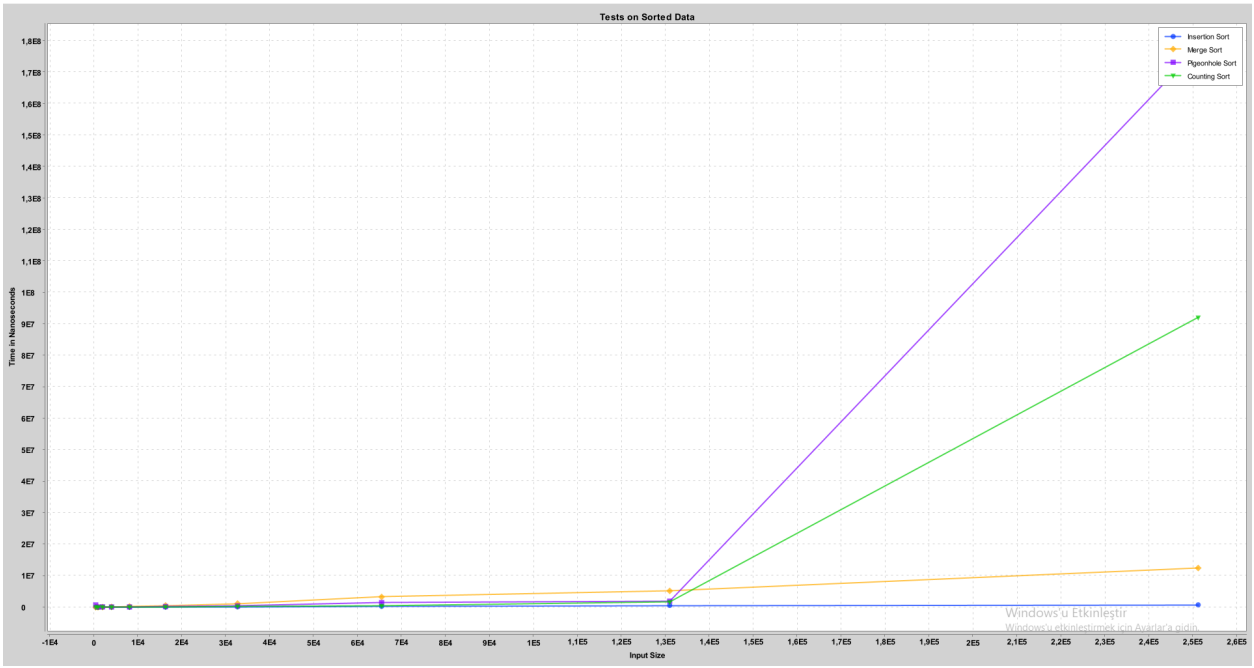

Figure 3: Tests on Sorted Data.

Briefly, red area creates an array from our data, turning it into integers. Green area reverses our integer array. In orange area, stringToInt method for random part, getArray method for other two parts.

References

- https://iq.opengenus.org/
- https://www.geeksforgeeks.org/

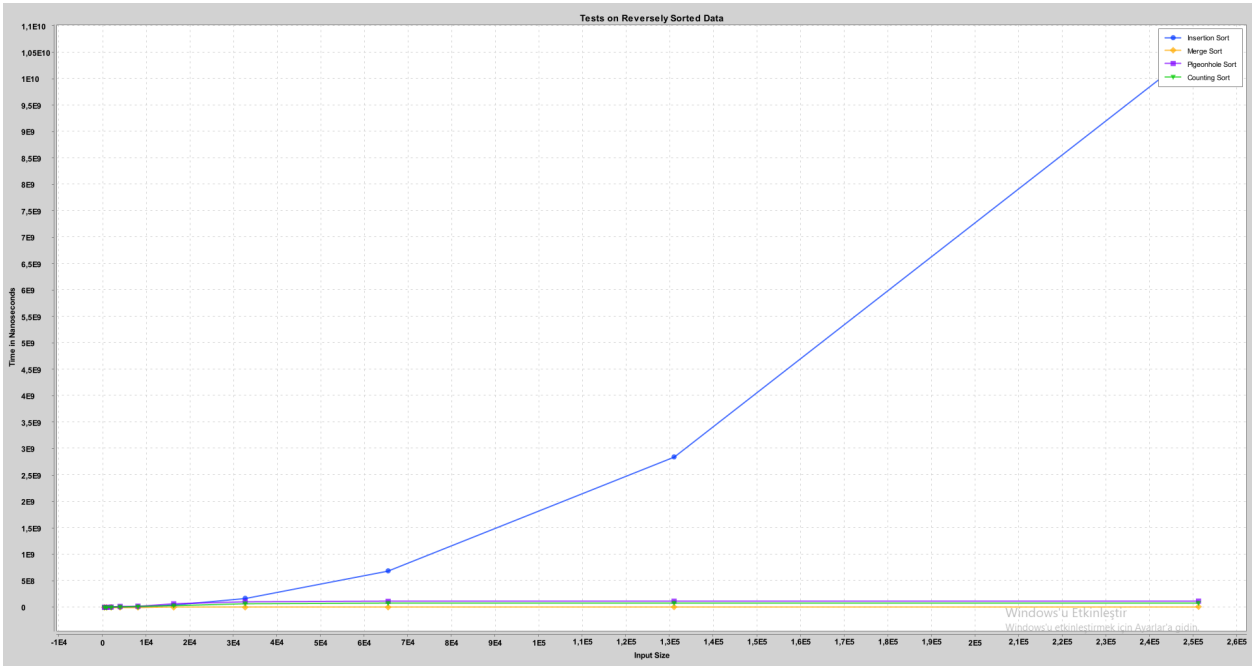

Figure 4: Tests on Reversely Sorted Data.

```java
        // X axis data
        int[] inputAxis = {512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 251281};

        // Create sample data for linear runtime
        double[][] yAxis = new double[4][10];

        CSVReader reader = new CSVReader();

        ArrayList<String> data = reader.reader("TrafficFlowDataset.csv");
        int[] intData = new int[251281];

        for(int x = 0; x < 251281; x++)
        {
            intData[x] = Integer.parseInt(data.get(x + 1));
        }

        MergeSort.sort(intData, 0, intData.length - 1);
        intData = reverseArray(intData);

        System.out.println("Insertion Sort:");
        for(int t = 0; t < inputAxis.length; t++)
        {
            double time = avgTime(data, intData, "insertion", inputAxis[t]);
            yAxis[0][t] = time;
        }

        System.out.println("MergeSort:");
        for(int t = 0; t < inputAxis.length; t++)
        {
            double time = avgTime(data, intData, "merge", inputAxis[t]);
            yAxis[1][t] = time;
        }

        System.out.println("Pigeonhole Sort:");
        for(int t = 0; t < inputAxis.length; t++)
        {
            double time = avgTime(data, intData, "pigeonhole", inputAxis[t]);
            yAxis[2][t] = time;
        }

        System.out.println("Counting Sort:");
        for(int t = 0; t < inputAxis.length; t++)
        {
            double time = avgTime(data, intData, "counting", inputAxis[t]);
            yAxis[3][t] = time;
        }
```

Figure 5: Inside of main method.

```
101
102             System.out.print("Sort Size is: " + sortSize);
103
104             for(int i = 0; i < 10; i++)
105             {
106 //              int[] array = stringToInt(data, sortSize);
107                 int[] array = getArray(intArray, sortSize);
108
109                 double start = System.nanoTime();
110                 InsertionSort.insertionSort(array);
111                 double end = System.nanoTime();
112
113                 avgTime += (end - start);
114             }
115             System.out.println(" Average time is: " + avgTime / 10 + " ns");
116             return avgTime / 10.0;
117         }
118
119         else if(sortType.equals("merge"))
120         {
121             double avgTime = 0;
122
123             System.out.print("Sort Size is: " + sortSize);
124
125             for(int i = 0; i < 10; i++)
126             {
127 //              int[] array = stringToInt(data, sortSize);
128                 int[] array = getArray(intArray, sortSize);
129
130                 double start = System.nanoTime();
131                 MergeSort.sort(array, 0, array.length - 1);
132                 double end = System.nanoTime();
133
134                 avgTime += (end - start);
135
136             }
137             System.out.println(" Average time is: " + avgTime / 10 + " ns");
138             return avgTime / 10.0;
139         }
140
141         else if(sortType.equals("pigeonhole"))
142         {
143             double avgTime = 0;
144
145             System.out.print("Sort Size is: " + sortSize);
146
147             for(int i = 0; i < 10; i++)
148             {
149 //              int[] array = stringToInt(data, sortSize);
150                 int[] array = getArray(intArray, sortSize);
151
152
```

Figure 6: The method where I run the tests (avgTime method).