Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2022 Spring

Programming Assignment 1

March 9, 2022

Student name:
Fatih Pehlivan

Student Number:
b21946529

1 Problem Definition

In this assignment, I was given a data-set to sort. I examined different algorithm efficiency and showed that the relationship between the running time of the algorithm implementations with their theoretical asymptotic complexities

2 Solution Implementation

Code implementation

2.1 Insertion Sort

```
import java.util.concurrent.TimeUnit;

public class InsertionSort {
    public static long insertionSort(int[] arr){
        long startTime = System.nanoTime();
        int n = arr.length;
        for (int i = 1; i < n; ++i) {
            int key = arr[i];
            int j = i - 1;

            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = key;

        }
        long endTime = System.nanoTime();
        return TimeUnit.MILLISECONDS.convert(endTime - startTime, TimeUnit.
           NANOSECONDS);
    }
}
```

2.2 Merge Sort

```
import java.util.concurrent.TimeUnit;

public class MergeSort {

    public static long mergeSort(int[] arr){
        long startTime = System.nanoTime();
        sort(arr, 0, arr.length-1);
        long endTime = System.nanoTime();
        return TimeUnit.MILLISECONDS.convert(endTime - startTime, TimeUnit.
            NANOSECONDS);
    }

    private static void merge(int[] arr, int l, int m, int r)
    {
        int n1 = m - l + 1;
        int n2 = r - m;
        int[] L = new int[n1];
        int[] R = new int[n2];
        for (int i = 0; i < n1; ++i)
            L[i] = arr[l + i];
        for (int j = 0; j < n2; ++j)
            R[j] = arr[m + 1 + j];
        int i = 0, j = 0;
        int k = l;
        while (i < n1 && j < n2) {
            if (L[i] < R[j]) {
                arr[k] = L[i];
                i++;
            }
            else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }
```

```
    private static void sort(int[] arr, int l, int r)
    {

        if (l < r) {
            int m =l+ (r-l)/2;
            sort(arr, l, m);
            sort(arr, m + 1, r);
            merge(arr, l, m, r);
        }
    }
}
```

2.3 Counting Sort

```
import java.util.Arrays;
import java.util.concurrent.TimeUnit;

public class CountingSort {
    static long countingSort(int[] arr)
    {
        long startTime = System.nanoTime();
        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]--;
        }
        System.arraycopy(output, 0, arr, 0, arr.length);
        long endTime = System.nanoTime();
        return TimeUnit.MILLISECONDS.convert(endTime - startTime, TimeUnit.
           NANOSECONDS);
    }

}
```

2.4 Pigeonhole Sort

```java
import java.util.Arrays;
import java.util.concurrent.TimeUnit;

public class PigeonholeSort {
    public static long pigeonholeSort(int[] arr){
        long startTime = System.nanoTime();
        sort(arr, arr.length);
        long endTime = System.nanoTime();
        return TimeUnit.MILLISECONDS.convert(endTime - startTime, TimeUnit.
            NANOSECONDS);
    }

    private static void sort(int[] arr, int n)
    {
        int min = arr[0];
        int max = arr[0];
        int range, i, j, index;

        for (int a = 0; a < n; a++) {
            if (arr[a] > max)
                max = arr[a];
            if (arr[a] < min)
                min = arr[a];
        }

        range = max - min + 1;
        int[] phole = new int[range];
        Arrays.fill(phole, 0);

        for (i = 0; i < n; i++)
            phole[arr[i] - min]++;

        index = 0;

        for (j = 0; j < range; j++)
            while (phole[j]-- > 0){
                arr[index++] = j + min;
            }
    }
}
```

3 Results, Analysis, Discussion

.

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	0.1	0.0	0.0	0.0	1.0	4.0	19.4	74.0	302.1	4890.0
Merge sort	0.0	0.0	0.0	0.0	0.0	1.0	2.0	5.6	13.3	24.2
Pigeonhole sort	277.8	191.2	195.3	179.7	192.2	184.6	169.8	180.7	232.4	178.1
Counting sort	170.9	121.9	117.8	117.6	122.9	117.6	112.7	114.3	159.6	124.5

Table 2: Results of the running time tests performed on the sorted data of varying sizes (in ms).

Algorithm	Input Size									
	512	1024	2048	4096	8192	16384	32768	65536	131072	251281
Insertion sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Merge sort	0.0	0.1	0.0	0.1	0.3	1.2	2.8	7.1	7.6	13.1
Pigeonhole sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6	211.4
Counting sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.4	146.3

Table 3: Results of the running time tests performed on the reversely sorted data of varying sizes (in ms).

Algorithm	Input Size									
	512	1024	2048	4096	8192	16384	32768	65536	131072	251281
Insertion sort	0.0	0.0	0.0	2.0	8.0	35.7	155.5	636.3	2407.5	8673.4
Merge sort	0.0	0.1	0.0	0.1	0.3	1.2	1.0	2.7	6.2	13.4
Pigeonhole sort	0.0	0.0	9.5	32.0	31.0	78.9	185.4	214.5	216.0	236.2
Counting sort	0.0	0.0	6.4	19.0	20.8	51.9	114.4	151.6	121.3	150.7

Complexity analysis tables:

Table 4: 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	$\Omega(n)$	$\Theta(n + k)$	$O(k)$
Counting Sort	$\Omega(n)$	$\Theta(n + k)$	$O(k)$

Table 5: Auxiliary space complexity of the given algorithms.

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

I obtain auxiliary memory in the given pseudo-codes for pigeonhole sort 5 and 6th row, for counting sort 2 and 3th row.

3.1 Insertion Sort

Time complexity of Insertion for average and worst case are $O(n^2)$ because there are 2 loops in the code. 2.1. For best case $O(n)$ because in the code 2nd loop will never run. For worst case 2nd loop runs in first step n-1 second step n-2 so on. Mathematical proof is $\sum_{k=1}^{n}(n-k) = n^2/2 - n/2$

Auxiliary space complexity is O(1) because no array was created. We define some constant values (like n and times 2.1.).

3.2 Merge Sort

Best , average and worst case equal to each other. Because the algorithm divide the array until there are 2 elements remaining each array. Then sorts the arrays then merge them. $\sum_{k=0}^{n/2}(2^k * N/2^k) = n \log n$ 2.2

Auxiliary space complexity is O(n) because we create some arrays. Array sizes are n, n/2, n/4... Total space is $= \sum_{k=1}^{n/2}(n/k) < 2n$. See for more 4.4. So that space complexity should be linear

3.3 Pigeonhole Sort

Worst case: when data is skewed and range is large, K is too large according to n. O(k).

Best Case: When all elements are same, K is 1. (O(n)).

Average Case: O(N+K) (N and K equally dominant).

See for more 4.4

Auxiliary space complexity is O(k) because I create one more array it is named ”phole”. The size of k is max element of array - min element of array + 1. 2.4

3.4 Counting Sort

Time complexity is same as Pigeonhole Sort 3.3 Auxiliary space complexity is O(n + k) because I create two more array one is named "count". The size of k is max element of array - min element of array + 1. The other is output. It size's is n. 2.3

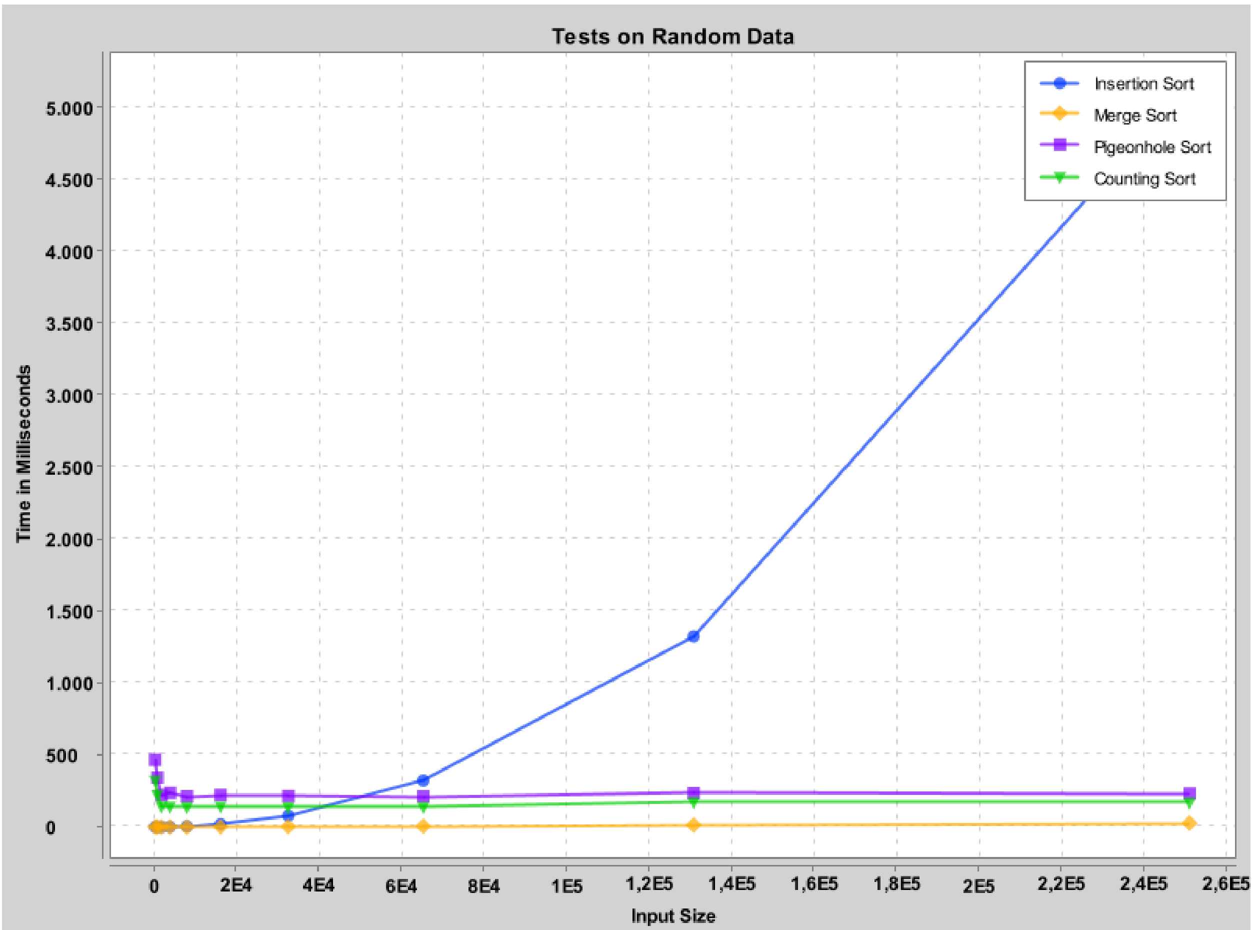

Figure 1: Random Data

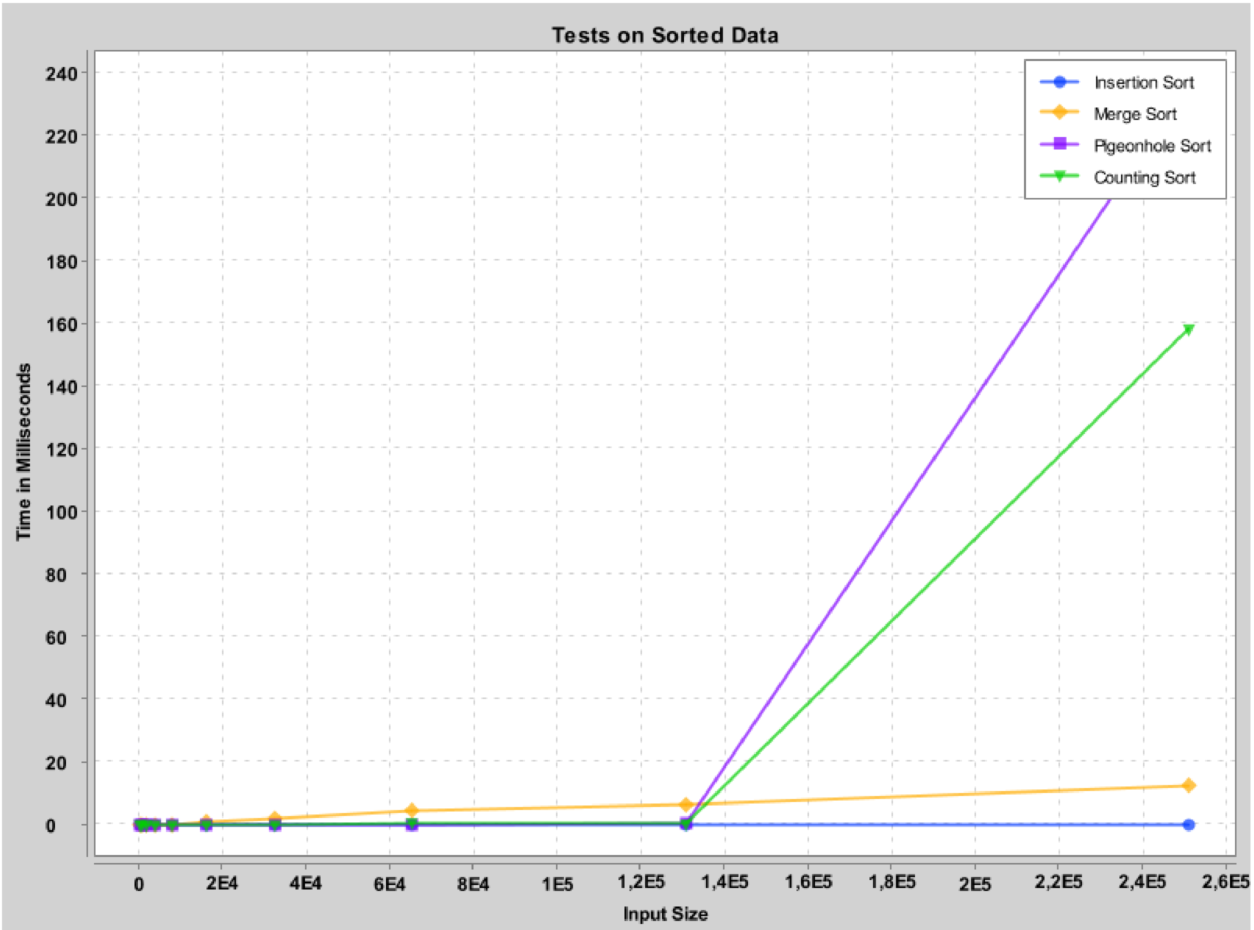

Figure 2: Sorted Data

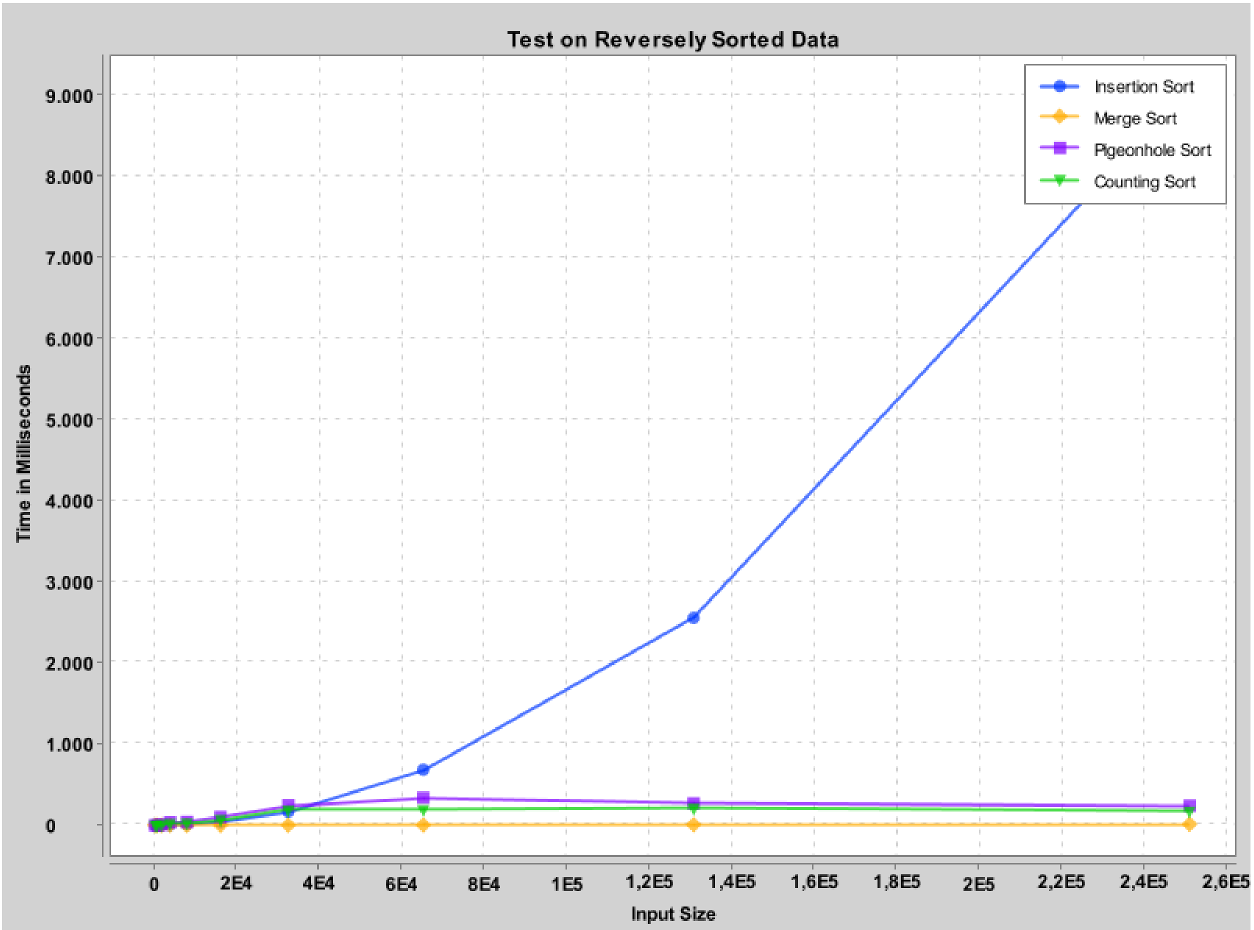

Figure 3: Reversely Sorted Data

4 Notes

4.1 Insertion Sort

Best Case: Sorted Data and it runs instant time
Worst Case: Reversely Sorted Data and it runs $O(n^2)$
Average Case: Random Data.
The obtained results match their theoretical asymptotic complexities. For best case the time is instant, for worst and average case when the input data is doubled, the running time is quadrupled.

4.2 Merge Sort

For all cases, the algorithm runs the same time results.
The obtained results match their theoretical asymptotic complexities. test results are close to each other.

4.3 Pigeonhole Sort

Best Case: Reversely sorted or sorted data. K value is very small (not entire values, we can see the result for 512, 1024 so on.).
Worst Case: When the data set length is small and k value is very big. We may not see this situation on these graphs directly.
Average Case: When k and length equally dominant. Like worst case we may not see this situation on these graphs directly.
The obtained results match their theoretical asymptotic complexities. Test results are close to each other when input size 251281. When the data sorted or reversely sorted and the input size is not very big we can see best case situation.

4.4 Counting Sort

Very similar to Pigeonhole Sort. I am going to explain why counting sort faster than Pigeonhole Sort. To do pigeonhole sort I used the for loop which contains while loop. This affects the time negatively.

The Computer Features

- Intel® Tiger Lake Core™ i7-11800H
- 8 gb RAM
- Windows 11

References

- https://stackoverflow.com/questions/63577325/why-is-merge-sort-space-complexity-on
- https://iq.opengenus.org/time-and-space-complexity-of-counting-sort/
- https://www.geeksforgeeks.org/insertion-sort/
- https://www.geeksforgeeks.org/merge-sort/
- https://www.geeksforgeeks.org/counting-sort/
- https://www.geeksforgeeks.org/java-program-for-pigeonhole-sort/