Hacettepe University

Computer Engineering Department

BBM204 Software Laboratory II - 2022 Spring

# Programming Assignment1
# Algorithm Complexity Analysis

March 9, 2022

Student name:
Melike Nur DULKADİR

Student Number:
b21992919

# 1. Problem Definition

In this assignment, we measure the running time of 4 different sort algorithms on inputs of different sizes and we make algorithm complexity analysis

# 2.Solution Implementation

For the solution, I first wrote the sort algorithms. I created new arrays according to different input sizes of each sort algorithm and filled these arrays with the data in the dataset. Then I combined these arrays according to the sort algorithms and sent them to the experiment method. I called the sort algorithms according to different input numbers and measured the running times. I filled the y axis on the graph with these measured running times. Finally, I printed the graphics of the running times according to the input size using xchart.

## 2.1 Insertion Sort

```java
public class InsertionSort {
    public static void insertionSort(int array[]) {
        int n = array.length;
        for (int j = 1; j < n; j++) {
            int elem = array[j];
            int i = j - 1;
            while ((i > -1) && (array[i] > elem)) {
                array[i + 1] = array[i];
                i--;
            }
            array[i + 1] = elem;
        }
    }
}
```

## 2.2 Merge Sort

```java
public class MergeSort {
    public static int[] mergeSort(int[] array) {
        int n = array.length;
        if (n < 2) {
            return array;
        }
        int mid = n / 2;
        int[] left = new int[mid];
        int[] right = new int[n - mid];

        for (int i = 0; i < mid; i++) {
            left[i] = array[i];
        }
        for (int i = mid; i < n; i++) {
            right[i - mid] = array[i];
        }
        left = mergeSort(left);
        right= mergeSort(right);
        return merge(left, right);
    }
```

```java
    // Function for merged array after dividing
    public static int[] merge(int[] left, int[] right) {
        int left_size = left.length;
        int right_size = right.length;
        int[] sorted_array = new int[left_size + right_size];
        int i = 0, j = 0, k = 0;

        // Comparing the elements in the right and left arrays and adding them to the new array
        while (i < left_size && j < right_size) {
            if (left[i] < right[j])
                sorted_array[k++] = left[i++];
            else
                sorted_array[k++] = right[j++];
        }

        // Add remaining elements of left array
        while (i < left_size)
            sorted_array[k++] = left[i++];

        // Add remaining elements of right array
        while (j < right_size)
            sorted_array[k++] = right[j++];

        return sorted_array;
    }
}
```

## 2.3 Pigeonhole Sort

```java
import java.util.*;
public class PigeonholeSort {
    public static void pigeonholeSort(int array[]) {
        int size = array.length;
        int min = array[0];
        int max = array[0];
        int range, i, j, index;

        // Finding max and min values
        for (int k : array) {
            if (k > max)
                max = k;
            if (k < min)
                min = k;
        }
        range = max - min + 1;
        // Creating new int array named pigeon_hole the same size as of the range
        int[] pigeon_hole = new int[range];
        Arrays.fill(pigeon_hole,  val: 0);

        for (i = 0; i < size; i++)
            pigeon_hole[array[i] - min]++;

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

## 2.4 Counting Sort

```java
public class CountingSort {
    public static void countSort(int array[]) {
        int size = array.length;
        int[] output = new int[size + 1];
        int max = array[0];

        // Find the max element of the array
        for (int i = 1; i < size; i++) {
            if (array[i] > max)
                max = array[i];
        }
        // Creating and initializeing count array
        int[] count = new int[max + 1];
        for (int i = 0; i < max; ++i) {
            count[i] = 0;
        }
        // Adding count of each element in the array to the count array
        for (int i = 0; i < size; i++) {
            count[array[i]]++;
        }
        // Adding the cummulative count of each element
        for (int i = 1; i <= max; i++) {
            count[i] += count[i - 1];
        }
        // Finding the index of each element and placing it in an output array
        for (int i = size - 1; i >= 0; i--) {
            output[count[array[i]] - 1] = array[i];
            count[array[i]]--;
        }
        System.arraycopy(output,  srcPos: 0, array,  destPos: 0, size);
    }
}
```

# 3. Results, Analysis, Discussion

I measured the average running time of each algorithm by running the inputs of different sizes 10 times in each experiment and taking the average running times. Then, I added each average time I obtained to the input size sections they belonged to in the tables of the relevant experiments.

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 | 1.5 | 2.0 | 0.9 | 5.1 | 12.7 | 40.7 | 164.1 | 658.3 | 2637.3 | 9809.5 |
| Merge Sort | 0.2 | 0.2 | 0.6 | 0.7 | 1.6 | 3.4 | 7.2 | 16.7 | 40.6 | 54.9 |
| Pigeonhole Sort | 316.9 | 172.9 | 179.7 | 179.3 | 182.5 | 172.8 | 160.5 | 170.9 | 170.9 | 164.4 |
| Counting Sort | 207.0 | 194.6 | 198.2 | 198.5 | 190.8 | 193.0 | 193.8 | 197.9 | 202.2 | 213.1 |

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 | 0.1 | 0.1 | 0.7 | 0.9 | 6.7 |
| Merge Sort | 0.1 | 0.3 | 1.5 | 0.9 | 1.9 | 3.2 | 5.3 | 11.0 | 26.2 | 36.2 |
| Pigeonhole Sort | 177.8 | 177.4 | 197.2 | 176.9 | 172.1 | 169.5 | 174.3 | 172.4 | 158.4 | 176.4 |
| Counting Sort | 210.0 | 207.4 | 211.8 | 204.0 | 202.8 | 197.9 | 196.2 | 200.0 | 202.7 | 196.6 |

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

| Algorithm | Input Size | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 512 | 1024 | 2048 | 4096 | 8192 | 16384 | 32768 | 65536 | 131072 | 251281 |
| Insertion Sort | 0.1 | 1.2 | 1.9 | 5.4 | 19.4 | 82.0 | 348.7 | 1326.9 | 5238.0 | 19652.0 |
| Merge Sort | 0.0 | 0.2 | 0.2 | 0.4 | 0.8 | 3.2 | 3.5 | 7.3 | 14.8 | 25.3 |
| Pigeonhole Sort | 155.9 | 168.2 | 176.7 | 161.7 | 164.4 | 171.3 | 161.9 | 172.3 | 175.5 | 171.3 |
| Counting Sort | 215.9 | 195.8 | 207.2 | 198.6 | 192.9 | 188.9 | 196.4 | 194.2 | 202.6 | 206.6 |

Then, I filled the Computational complexity comparison of the given algorithms and Auxiliary space complexity of the given algorithms tables using the results I obtained.

Table 4 : Computational complexity comparison of the given algorithms.

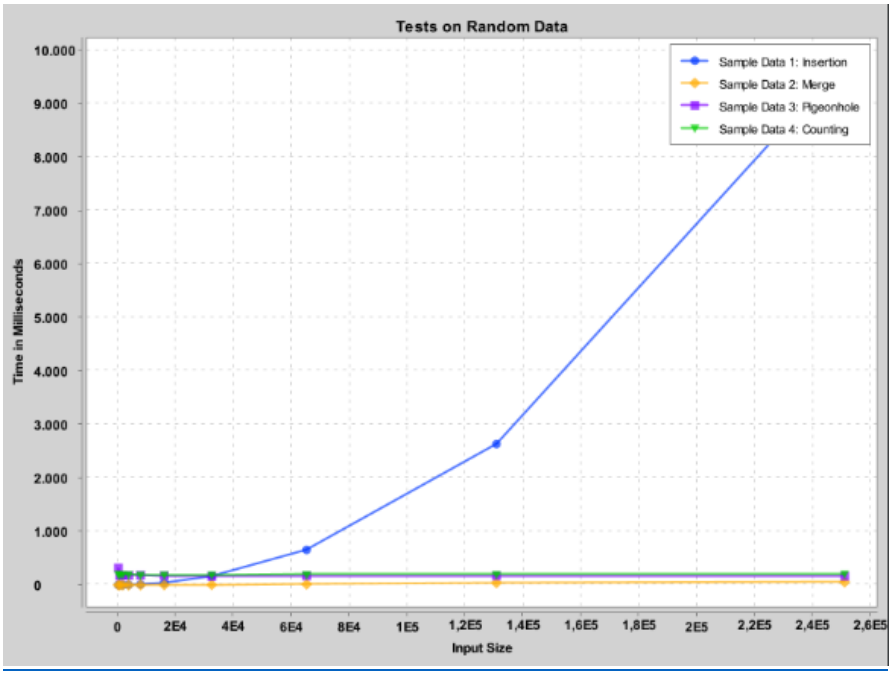
| **Algorithm** | **Best case** | **Average Case** | **Worst Case** |
|---|---|---|---|
| Insertion Sort | $\Omega(n)$ | $\Omega(n^2)$ | $\Omega(n^2)$ |
| Merge Sort | $\Omega(nlogn)$ | $\Omega(nlogn)$ | $\Omega(nlogn)$ |
| Pigeonhole Sort | $\Omega(n+N)$ | $\Omega(n+N)$ | $\Omega(n+N)$ |
| Counting Sort | $\Omega(n+k)$ | $\Omega(n+k)$ | $\Omega(n+k)$ |

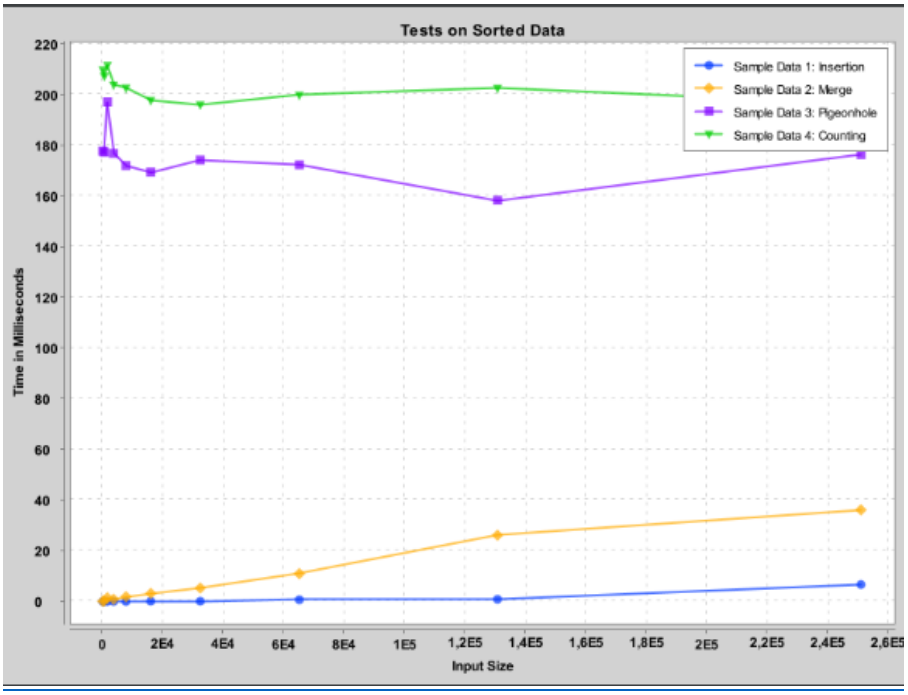
Table 5 : Auxiliary space complexity of the given algorithms.

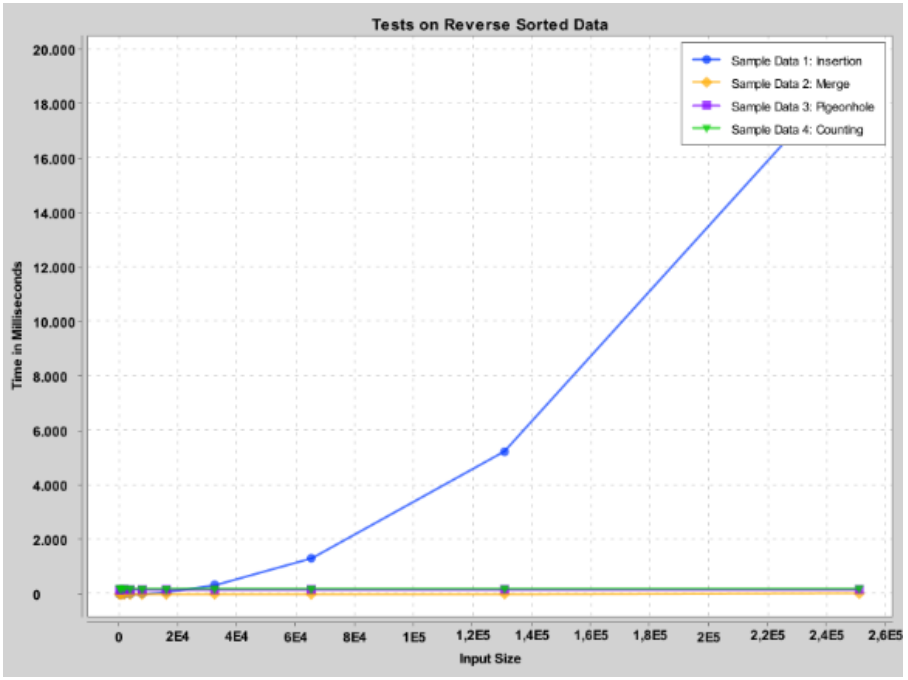
| **Algorithm** | Auxiliary Space Complexity |
|---|---|
| Insertion Sort | $O(1)$ |
| Merge Sort | $O(n)$ |
| Pigeonhole Sort | $O(n+N)$ |
| Counting Sort | $O(n+k)$ |

k : ***Maximum Value of the Integer Set***
N : ***Range of the Input Data***

Three graph results of obtained from running times:




Tests on Sorted Data
Sample Data 1: Insertion
Sample Data 2: Merge
Sample Data 3: Pigeonhole
Sample Data 4: Counting
Time in Milliseconds
220
200
180
160
140
120
100
80
60
40
20
0
0
2E4
4E4
6E4
8E4
1E5
1,2E5
1,4E5
1,6E5
1,8E5
2E5
2,2E5
2,4E5
2,6E5
Input Size


Tests on Reverse Sorted Data
Sample Data 1: Insertion
Sample Data 2: Merge
Sample Data 3: Pigeonhole
Sample Data 4: Counting
Time in Milliseconds
20.000
18.000
16.000
14.000
12.000
10.000
8.000
6.000
4.000
2.000
0
0
2E4
4E4
6E4
8E4
1E5
1,2E5
1,4E5
1,6E5
1,8E5
2E5
2,2E5
2,4E5
2,6E5
Input Size

# 4. **Conclusion**

Insertion sort is one of the basic sorting algorithms. It is less efficient than other basic sorting algorithms, but has some advantages. Stable, in-place, and adaptive are a few of them. Worst and average case time complexity of Insertion sort is $O(n^2)$. Best case occurs when the array is reverse sorted. This is an in-place algorithm so that space complexity is O(1). Also Insertion sort is a stable sorting algorithm.

Merge sort is another of the basic sorting algorithms. It is an efficient and common algorithm in general use. This algorithm also has divide-and-conquer feature. The time complexity is O(nlogn). Best case is O(nlogn). Merge sort is not an in-place algorithm, so that space complexity is O(N). But it can be modified to be in-place. This is a stable algorithm.

Pigeonhole sort can be used where the number of elements and the length of the range of possible key value are approximately the same. Counting sort has some differences even though it is similar. All cases of this algorithm, even the space complexity is O(n+N). It is an in-place sorting algorithm

Counting sort is an integer sorting algorithm. Worst and average case time complexity of counting sort is O(n + k). Best case occurs when the array is already sorted, O(n + k). The space complexity of counting sort is O(max). The larger the range of elements, the larger the space complexity.. Also counting sort is a stable sorting algorithm.

**If we have enough memory we can use pigeonhole and counting sort. Because they are fast running algorithms, but space complexity is inefficient. The most effective algorithm is merge sort because it is stable, fast, and space complexity is open to be improved. The worst algorithm is insertion sort can be said, because of the time complexity.**

# REFERENCES


1. https://www.geeksforgeeks.org/insertion-sort/
2. https://www.geeksforgeeks.org/merge-sort/
3. https://www.geeksforgeeks.org/java-program-for-pigeonhole-sort/
4. https://www.programiz.com/dsa/counting-sort
5. BBM204PA1__2022__(1).pdf
6. https://www.baeldung.com/java-invert-array