Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2022 Spring

Programming Assignment 1

March 11, 2022

Student name:
Alper Özöner

Student Number:
b2200765037

1 Problem Statement

Efficient sorting is important for optimizing the efficiency of other algorithms that require input data to be sorted. The efficiency of a sorting algorithm can be observed by applying it to sort datasets of varying sizes and other characteristics of the dataset instances that are to be sorted. In this report, we will be classifying the given sorting algorithms (insertion, merge, pigeonhole, and counting sort) based on their run-time data obtained from 3 experiments with random data (1), sorted data (2), and reversed data (3). Additionally, we will also look into the space complexities of the algorithms we will be using in this report.

2 Java Codes of the Sorting Algorithms

Java codes of the four sorting algorithms created for this report are included below:

2.1 Insertion Sort

```java
public class InsertionSort {
    public static void sort (Integer[] a) {
        for (int j = 1; j < a.length; j++) {
            int key = a[j];
            int i = j - 1;
            while (i >= 0 && a[i] > key) {
                a[i + 1] = a[i];
                i--;
            }
            a[i + 1] = key;
        }
    }
}
```

2.2 Merge Sort

```java
import java.util.*;

public class MergeSort {

    private static void merge(Comparable[] a, Comparable[] aux, int lo, int
        mid, int hi) {
        // populate aux (clone) array
        for (int k = lo; k <= hi; k++) {
            aux[k] = a[k];
        }

        // merge here
        int i = lo, j = mid+1;
        for (int k = lo; k <= hi; k++) {
            if (i > mid)
                a[k] = aux[j++];
            else if (j > hi)
                a[k] = aux[i++];
            else if (less(aux[j], aux[i]))
                a[k] = aux[j++];
            else
                a[k] = aux[i++];
        }
    }

    // recursively sort
    public static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
        {
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, aux, lo, mid);
        sort(a, aux, mid + 1, hi);
        merge(a, aux, lo, mid, hi);
    }

    private static boolean less(Comparable v, Comparable w) {
        return v.compareTo(w) < 0;
    }
}
```

2.3 Pigeonhole Sort

```java
import java.util.Arrays;

public class PigeonHole {
    public static void sort(Integer[] a, int n, int min, int max)
    {
        int range, i, j, index;
        range = max - min + 1;
        int[] phole = new int[range];
        Arrays.fill(phole, 0);

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

        index = 0;

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

2.4 Counting Sort

```
import java.util.Arrays;
import java.util.List;
public class CountingSort {
    static void sort(Integer[] arr, int n, int min, int max) {
        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];
        }
    }
}
```

3 Run-time Experiment Results

Three running time results tables corresponding to three experiment sets performed on the given random,(1) sorted,(2) and reversely sorted data,(3) for varying input sizes are given below. All four algorithms are tested and work as expected.

3.1 Experiment Results on the Given Random 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	0.2	0.1	0.4	0.8	2.1	9.5	70.9	174.2	1248.3	6552.6
Merge sort	0.1	0	0	0.3	4.5	12.6	12	11.9	21	42.9
Pigeonhole sort	152.5	149.4	140.9	146.2	140.8	147.2	145.2	144.4	150.2	150.8
Counting sort	100.6	95	93.7	97.2	93.9	97.7	95.4	99.9	103.4	105.9

3.2 Experiment Results on the Sorted Data

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	0.2	0.2
Merge sort	0.1	0	0	1.7	5.1	5.6	2.8	6.8	14.7	31.6
Pigeonhole sort	0	0	0.1	0	0	0	0	0	1.9	267.5
Counting sort	0	0	0	0	0	0	0	0	0.9	129.4

3.3 Experiment Results on the Reversely Sorted Data

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	0.3	0.2	0.3	1.5	4.1	23.6	69.5	277.9	1307.8	5901.6
Merge sort	0	0.1	0.1	0.4	4.3	11.6	8.9	12.7	20.3	36.7
Pigeonhole sort	0.4	0.1	5.8	23	25.1	63.3	137.6	149.8	150.8	151.7
Counting sort	0.3	0	1.9	11.7	16.2	41	92.3	98.5	104.5	105.4

4 Time and Space Complexity Tables

Looking at the run-time in milliseconds in part 3, the pigeonhole and counting sort algorithms had very low run-times, similar to the performance of merge sort, while vastly outperforming insertion sort. However, one stark difference was that while the run-time values of the merge sort linearly increased, the values of pigeonhole and counting sort **hardly increased between two consecutive input sizes**, for example, the difference in time was negligible between an input size of **131072 to 251281 in the reversely sorted table**. Which was the last two biggest input sizes. The time complexity expressions for pigeonhole and counting sort algorithms have been calculated by looking at the experimental run-time data obtained from the tables in part 3 and can be seen below:

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	$\Omega(n)$	$\Theta(n)$	$O(n)$
Counting Sort	$\Omega(n)$	$\Theta(n)$	$O(n)$

Pigeonhole and counting sort algorithms are better performing when it comes to their run-times, as seen in previous parts; however, they use up **much more space than the insertion and merge sort**. This is because insertion sort and merge sort are comparison-based, while pigeonhole and counting sort are non-comparison based algorithms and perform the sorting task by creating auxiliary arrays **for tallying the elements which require at least an N-sized array** where N is given input size. The space complexity of pigeonhole and counting sort algorithms are included below:

Table 5: Auxiliary space complexity of the given algorithms.

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

5 Graphs of the 3 Experiments

5.1 Graph of the Random Data Experiment

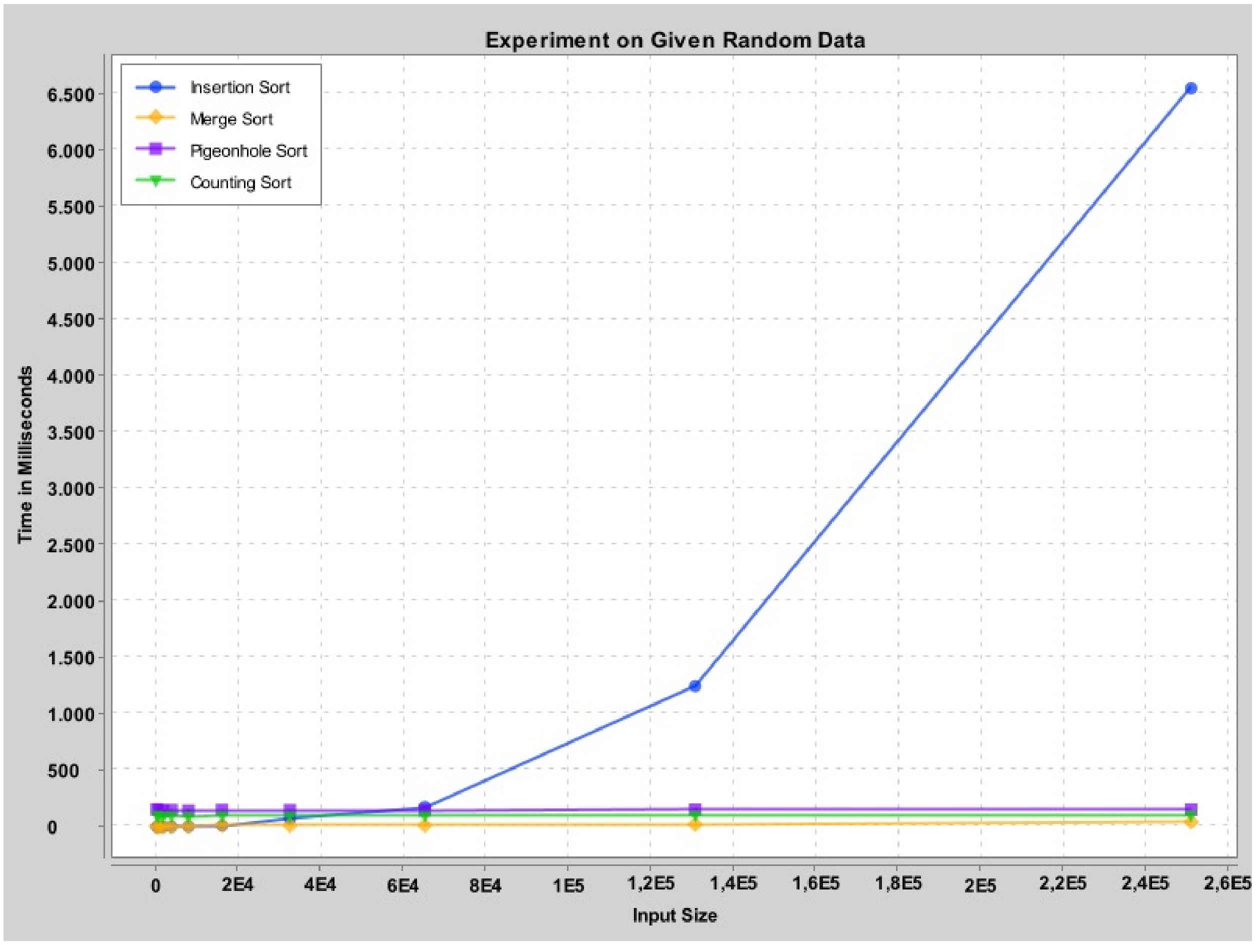

Figure 1: Plot of the functions.

5.2 Graph of the Sorted Data Experiment

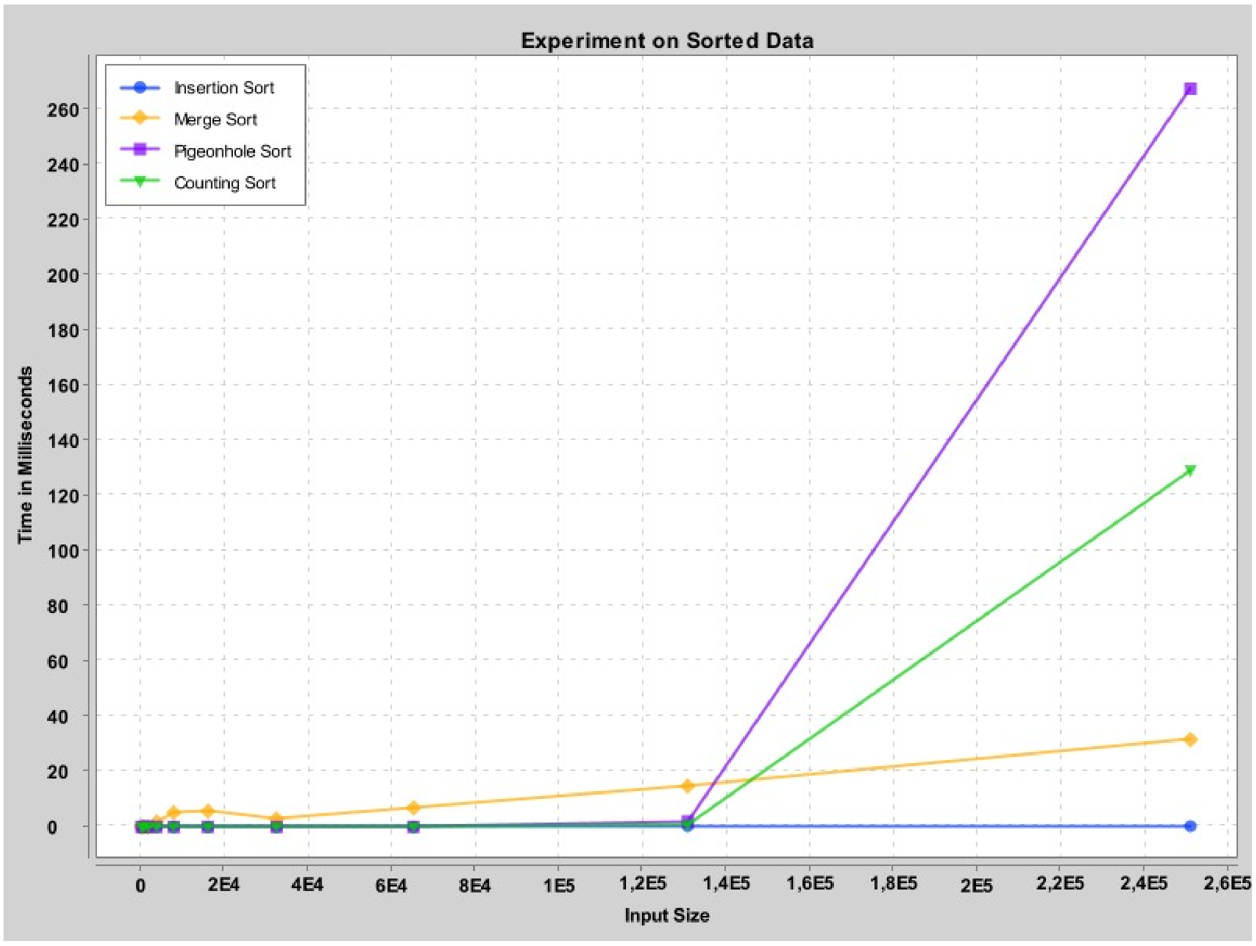

Figure 2: Plot of the functions.

5.3 Graph of the Reversely Sorted Data Experiment

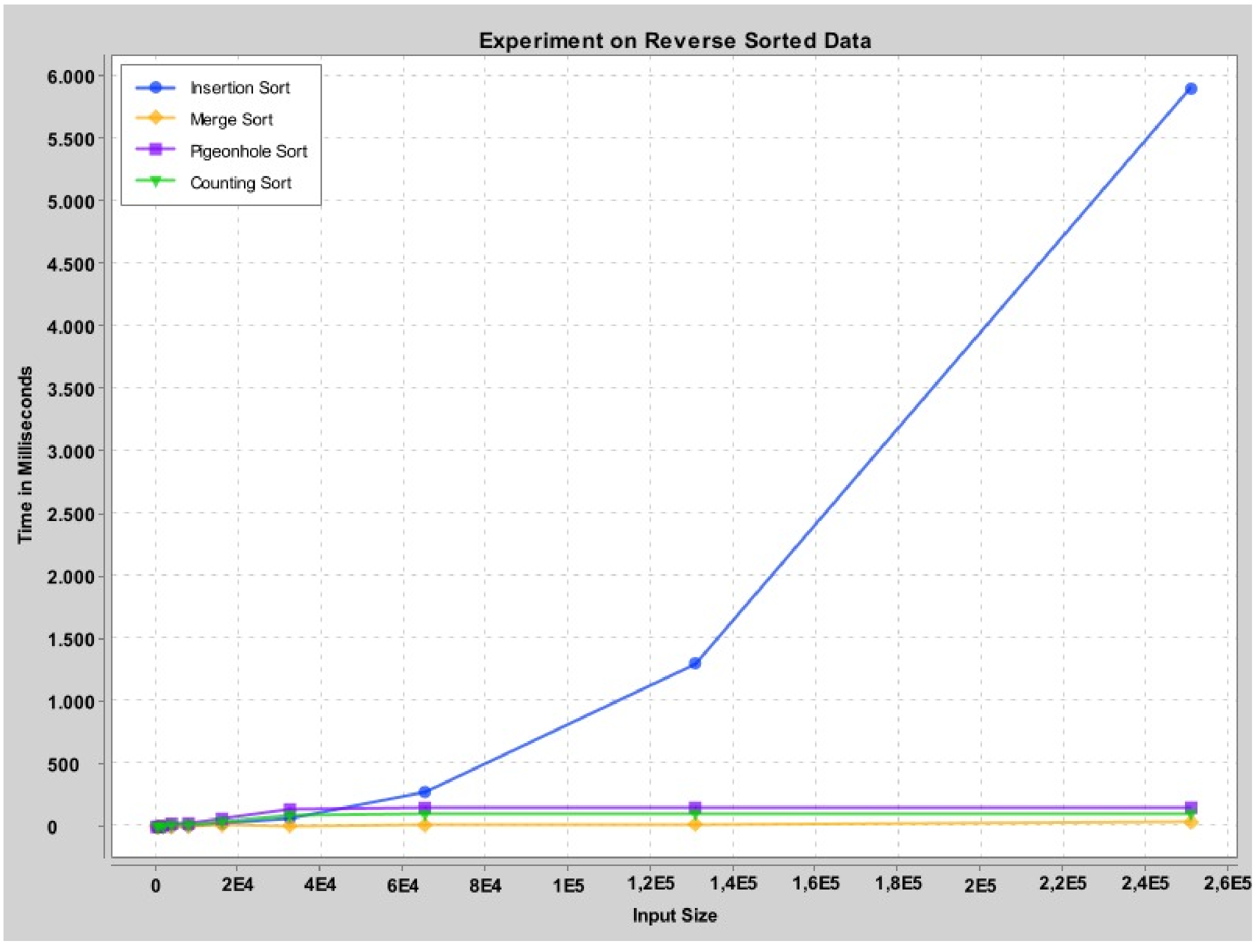

Figure 3: Plot of the functions.

6 Discussion & Conclusion

Question 1:
What are the best, average, and worst cases for the given algorithms in terms of the given input data to be sorted?
Answer:
For insertion sort, the best case seems to be the sorted data as it only needed to make n-1 comparisons. Interestingly enough, in our experiments, insertion sort performed slightly more slower with the random data when compared to the reversed data, which may be due to the fact that we have repeated the experiments only 10 times.On the other hand, the three other algorithms performed

very similarly with random (1) and reversed data (3) whereas they stopped sorting almost instantly with the sorted (2) set.
Question 2:
Do the obtained results (running times of your algorithm implementations) match their theoretical asymptotic complexities?
Answer:
The insertion sort definitely performed as intended, with a clear exponential curve which can clearly be seen in random (1) and reversed (3) data experiments. On the other hand, it is very hard to make a certain judgement about the other three algorithms (merge, pigeonhole, and counting sort) as their run-times appeared almost constant (but hardly increasing) with the random data (1), and linearly increasing with the reversed data (3). There was also an unexpected jump run-time in the biggest input size on the sorted data (2) experiment. This may be due to the operational time cost during the creation of auxiliary arrays which manifested itself most strongly in the biggest input size of 251281.

References

- https://iq.opengenus.org/time-and-space-complexity-of-counting-sort/
- https://en.wikibooks.org/wiki/LaTeX/Floats,_Figures_and_Captions
- https://www.geeksforgeeks.org/time-complexity-and-space-complexity/