Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2022 Spring

Programming Assignment 1

February 26, 2022

Student name:
Mert Ali Yalçin

Student Number:
21946682

1 Problem Definition

Analysis of algorithms is the area of computer science that provides tools to analyze the efficiency of different methods of solutions. Efficiency of an algorithm depends on these parameters; i) how much time, ii) memory space, iii) disk space it requires. Analysis of algorithms is mainly used to predict performance and compare algorithms that are developed for the same task. Also it provides guarantees for performance and helps to understand theoretical basis. A complete analysis of the running time of an algorithm involves the following steps:

- Implement the algorithm completely.
- Determine the time required for each basic operation
- Identify unknown quantities that can be used to describe the frequency of execution of the basic operations.
- Develop a realistic model for the input to the program
- Analyze the unknown quantities, assuming the modeled input.
- Calculate the total running time by multiplying the time by the frequency for each operation, then adding all the products.

In this experiment, you will analyze different sorting algorithms and compare their running times on a number of inputs with changing sizes.

Efficient sorting is important for optimizing the efficiency of other algorithms (such as search and merge 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 assignment, you will be classifying the given sorting algorithms based on two criteria:

- Computational (Time) Complexity: Determining the best, worst and average case behavior in terms of the size of the dataset.
- Auxiliary Memory (Space) Complexity: Some sorting algorithms are performed "in-place" using swapping. An in-place sort needs only O(1) auxiliary memory apart from the memory used for the items being sorted. On the other hand, some algorithms may need O(log n) or O(n) auxiliary memory for sorting operations.

The main objective of this assignment is to show the relationship between the running time of the algorithm implementations with their theoretical asymptotic complexities. You are expected to implement the algorithms given as pseudocodes, and perform a set of experiments on the given datasets to show that the empirical data follows the corresponding asymptotic growth functions. To do so, you will have to consider how to reduce the noise in your running time measurements, and plot the results to demonstrate and analyze the asymptotic complexities.

2 Solution Implementation

2.1 Insertion Sort

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

2.2 Merge Sort

```java
    public static void merge(int[] a, int[] aux, int lo, int mid, int hi)
    {
        for(int k = lo; k<=hi; k++) {aux[k] = a[k];}

        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(aux[j] < aux[i]) {a[k] = aux[j++];}
            else {a[k] = aux[i++];}
        }
    }

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

    public static void mergeSort(int[] a)
    {
        int[] aux = new int[a.length];
        recMergeSort(a, aux, 0, a.length - 1);
    }
```

2.3 Pigeonhole Sort

```
    public static void pigeonholeSort(int[] a)
    {
        int n = a.length;
        int min = a[0];
        int max = a[0];
        for(int i = 0; i < n; i++)
        {
            if(a[i] > max) {max = a[i];}
            if(a[i] < min) {min = a[i];}
        }
        int range = max - min + 1;
        int[] holes = new int[range];
        Arrays.fill(holes, 0);

        for (int i = 0; i < n; i++)
        {
            holes[a[i] - min]++;
        }

        int index = 0;

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

2.4 Counting Sort

```
    public static void countingSort(int[] array, int size)
    {
        int[] output = new int[size + 1];
        int max = array[0];
        for (int i = 1; i < size; i++) {
          if (array[i] > max)
```

```java
            max = array[i];
        }
        int[] count = new int[max + 1];
        for (int i = 0; i < max; ++i) {
          count[i] = 0;
        }
        for (int i = 0; i < size; i++) {
          count[array[i]]++;
        }
        for (int i = 1; i <= max; i++) {
          count[i] += count[i - 1];
        }
        for (int i = size - 1; i >= 0; i--) {
          output[count[array[i]] - 1] = array[i];
          count[array[i]]--;
        }
        for (int i = 0; i < size; i++) {
          array[i] = output[i];
        }
    }
```

3 Results, Analysis, Discussion

Your explanations, results, plots go in this section...

Running time test results are given in Table 1. Copy-paste the table to add two more for the results on sorted and reversely sorted data test. Don't forget to change the captions.

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	0,5	0,0	0,0	0.3	3,7	20,1	74,4	284,5	1167,9	4882,0
Merge sort	0,0	0,0	0,0	0,0	0,0	0,2	1,0	3,6	8,3	18,2
Pigeonhole sort	129,4	112,4	108,8	107,2	110,2	110,7	113,9	110,1	116,3	115,1
Counting sort	119,3	123,4	123,1	122,3	124,1	123,8	124,0	123,7	132,4	137,1

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	0,0	0,0	0,7	0,1	0,3
Merge sort	0,0	0,0	0,0	0,0	0,0	0,0	0,1	1,0	2,1	9,3
Pigeonhole sort	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,6	191,6
Counting sort	0,0	0,4	0,0	0,0	0,0	0,3	0,2	0,0	0,0	141,0

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

Algorithm	Input Size									
	512	1024	2048	4096	8192	16384	32768	65536	131072	251281
Insertion sort	0,2	0,0	0,0	1,2	8,5	35,9	146,0	579,0	2367,0	8958,5
Merge sort	0,0	0,0	0,0	0,0	0,0	0,0	0,0	1,0	3,0	7,9
Pigeonhole sort	0,0	0,0	3,3	23,4	25,5	61,5	111,5	107,7	114,0	120,4
Counting sort	116,3	110,0	108,3	115,3	116,3	114,8	119,4	122,6	121,1	124,7

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

Table 5: Auxiliary space complexity of the given algorithms.

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

where n is the number of elements and N is the range of the input data
Auxiliary spaces needed for pigeonhole sort and counting sort. For pigeonhole sort, in line 55, the array named "holes" causes extra space. For counting sort, in line 81, the array named "count" causes extra space.

1.

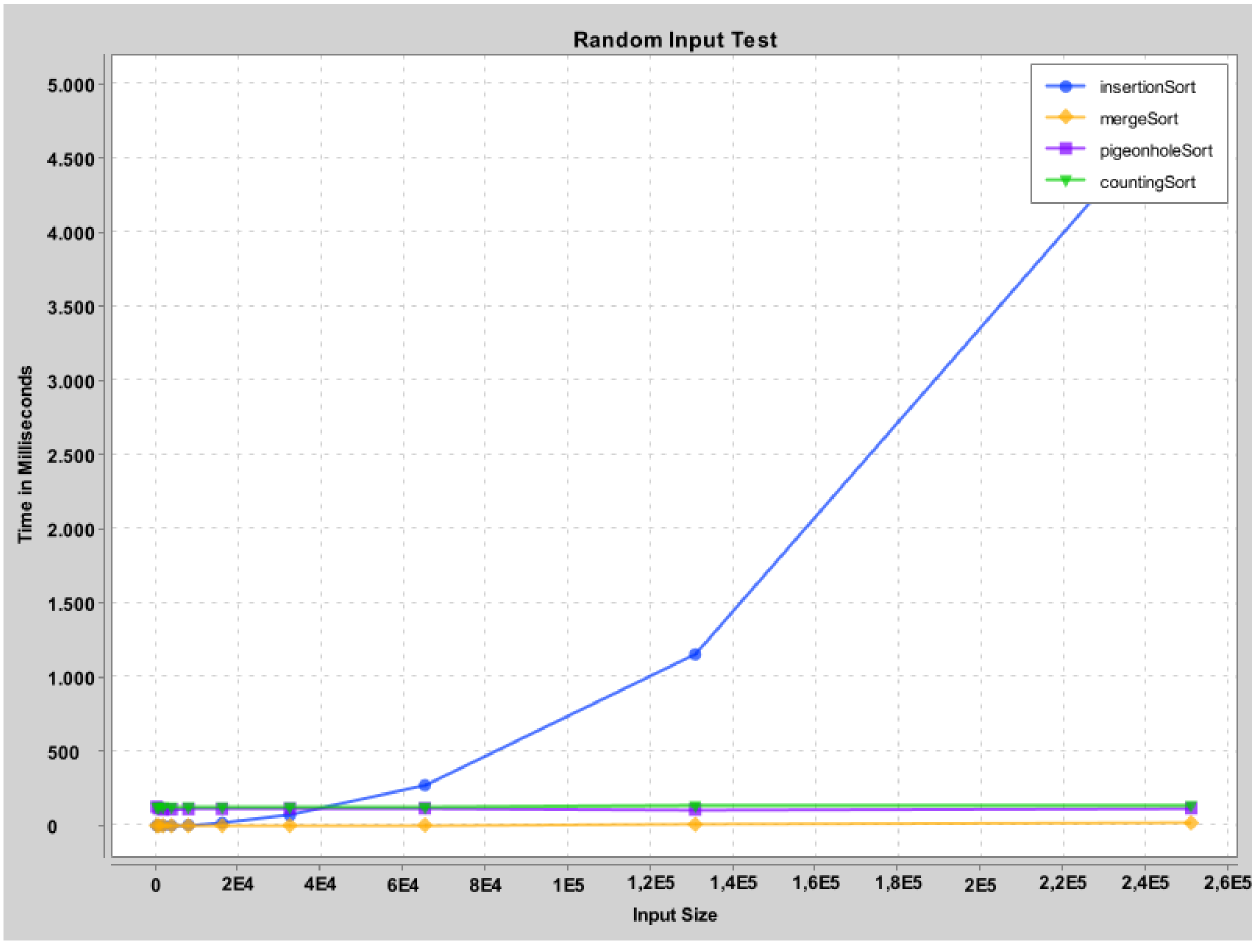

Figure 1: Plot of the functions used on random data.

2.

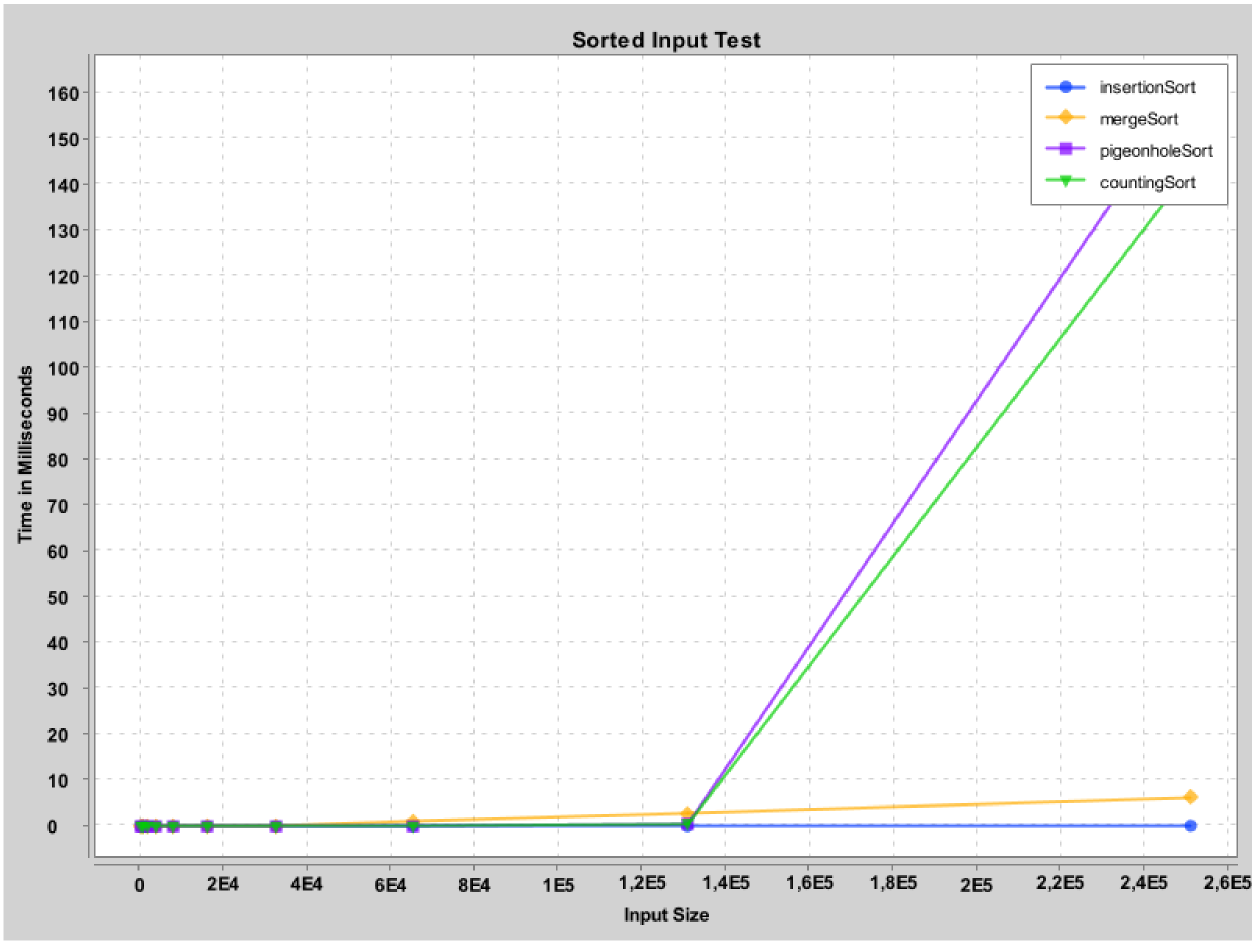

Figure 2: Plot of the functions used on sorted data.

3.

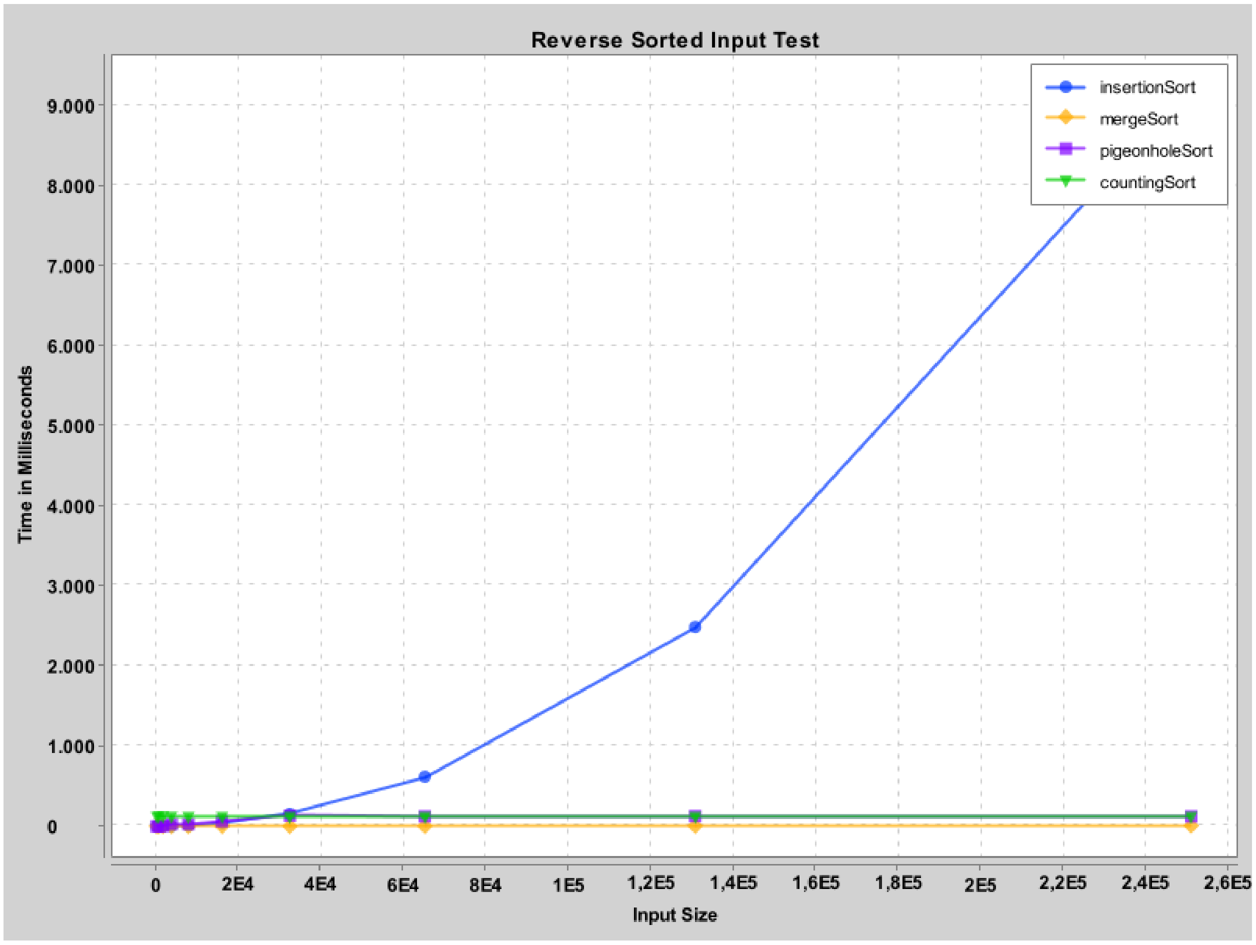

Figure 3: Plot of the functions used on reverse sorted data.

- What are the best, average, and worst cases for the given algorithms in terms of the given input data to be sorted?
  insertion sort == best: sorted / average: random / worst: reverse sorted For merge sort, pigeonhole sort and counting sort, the initial condition of the data does not matter. This means that they are input insensitive.

- Do the obtained results (running times of your algorithm implementations) match their theoretical asymptotic complexities?
  Yes, they do.
  –Insertion Sort:
  As you can see from the plots, the insertion works fastest at sorted input data(linear time). When you consider the largest input size (251282) in reverse sorted and random input data plots, you can clearly see that insertion sort works a bit faster on random data than reverse sorted data. The shape of the curve represents the $O(n^2)$ complexity as expected from its theoretical asymptotic complexity.
  –Merge Sort:
  As expected from its theoretical asymptotic complexity which is O(nlogn), the merge sort gives a fast and consistent performance in all three types of data.
  –Pigeonhole sort and Counting sort:
  Both of them works in O(n+N) time as expected. The sudden spike in sorted input data in the last input size is caused by the range of data which is N. When you consider the input size 131072, the maximum element in the input is 282627. When the program goes the next input size (251281) which is the last one, the maximum element in the input is 119999780. When you consider these two maximum elements, the input range(N) goes up by nearly 424 times. This increase in the input range(N) causes the sudden spike in the plot of sorted input data.

References

- BBM204 Lecture Notes
- BBM202 Lecture Notes