Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2022 Spring

Programming Assignment 1

March 6, 2022

Student name:
İrem Cengiz

Student Number:
b2200356006

1 Problem Definition

In computer science, algorithmic efficiency is a property of an algorithm which relates to the amount of computational resources used by the algorithm. An algorithm must be analyzed to determine its resource usage, and the efficiency of an algorithm can be measured based on the usage of different resources. 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.

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 assignment, I will be classifying the given sorting algorithms based on Computational (Time) Complexity and Auxiliary Memory (Space) Complexity.
Auxiliary Space is the extra space or temporary space used by an algorithm.
Time Complexity is defined as the number of times a particular instruction set is executed rather than the total time is taken.

2 Solution Implementation

2.1 Sorting Algorithm 1 : Insertion Sort

```
void InsertionSort(ArrayList<Integer> arr){
        int n = arr.size();
        for (int j=1; j<n; ++j)
        {
            int key = arr.get(j);
            int i = j-1;

            while (i>=0 && arr.get(i) > key)
            {
                arr.set(i+1, arr.get(i));
                i = i-1;
            }
            arr.set(i+1,  key);
        }
}
```

2.2 Sorting Algorithm 2: Merge Sort

```
16 public void mergeSort(int startIndex, int endIndex,ArrayList <Integer>
       inputArray) {
17
18         if (startIndex < endIndex && (endIndex - startIndex) >= 1) {
19             int mid = startIndex + (endIndex - startIndex) / 2;
20             mergeSort(startIndex, mid,inputArray);
21             mergeSort(mid+1, endIndex,inputArray);
22             merge(startIndex, endIndex,inputArray);
23         }
24     }
25
26     //merge
27     public void merge(int startIndex, int endIndex,ArrayList <Integer>
           inputArray) {
28         ArrayList<Integer> mergedSortedArray = new ArrayList<Integer>();
29         int midIndex = (endIndex + startIndex) / 2;
30         int leftIndex = startIndex;
31         int rightIndex = midIndex + 1;
32
33         while (leftIndex <= midIndex && rightIndex <= endIndex) {
34             if (inputArray.get(leftIndex) <= inputArray.get(rightIndex)) {
35                 mergedSortedArray.add(inputArray.get(leftIndex));
36                 leftIndex++;
37             } else {
38                 mergedSortedArray.add(inputArray.get(rightIndex));
39                 rightIndex++;
40             }
41         }
42         while (leftIndex <= midIndex) {
43             mergedSortedArray.add(inputArray.get(leftIndex));
44             leftIndex++;
45         }
46
47         while (rightIndex <= endIndex) {
48             mergedSortedArray.add(inputArray.get(rightIndex));
49             rightIndex++;
50         }
51
52         int i = 0;
53         int j = startIndex;
54         while (i < mergedSortedArray.size()) {
55             inputArray.set(j, mergedSortedArray.get(i++));
56             j++;
57         }
58     }
```

For Merge sort auxiliary space complexity is $O(n)$

(In line 28 we create an array)

2.3 Sorting Algorithm 3 : Pigeonhole Sort

```
60  public void pigeonhole_sort(ArrayList<Integer> arr, int n)
61    {
62        int min = arr.get(0);
63        int max = arr.get(0);
64        int range, i, j, index;
65        for(int a=0; a<n; a++)
66        {
67            if(arr.get(a) > max)
68                max = arr.get(a);
69            if(arr.get(a) < min)
70                min = arr.get(a);
71        }
72        range = max - min + 1;
73        int[] phole = new int[range];
74        Arrays.fill(phole, 0);
75
76        for(i = 0; i<n; i++)
77            phole[arr.get(i) - min]++;
78        index = 0;
79        for(j = 0; j<range; j++)
80            while(phole[j]-->0)
81                arr.set(index++,j+min);
82    }
```

For Pigeonhole sort auxiliary space cmplexity is $O(n + k)$. (In line 73 we create an array)

2.4 Sorting Algorithm 4 : Counting Sort

```
83
84      void counting_sort(ArrayList<Integer> nums) {
85          int k=nums.get(0);
86          for(int i=1;i<nums.size();i++){
87              if(nums.get(i)>k)
88                  k=nums.get(i);
89          }
90          int s = nums.size(); //s = size of sorted array
91          int sorted[] = new int[s]; //array to be sorted
92
93          int count[] = new int[k+1]; //size of array: max 1,000,000
94          for (int i = 0; i < k+1; ++i) {
95              count[i] = 0;
96          }
97          for (int i = 0; i < s; ++i) {
98              ++count[nums.get(i)];
99          }
100         for (int i = 1; i <= k; ++i) {
101             count[i] += count[i - 1];
102         }
103         //start sorting
104         for (int i = s - 1; i >= 0; i--) {
105             sorted[count[nums.get(i)] - 1] = nums.get(i);
106             --count[nums.get(i)];
107         }
108         for (int i = 0; i < s; ++i) {
109             nums.set(i, sorted[i]);
110         }
111     }
```

For counting sort auxiliary space complexity is $O(n+k)$ (In line 91 we create an array to be sorted)

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	1.6	0.0	3.2	7.7	34.3	138.3	565.0	2484.1	11875.0	52440.0
Merge sort	0.0	0.0	0.8	0.8	3.2	5.0	8.9	18.1	36.0	83.5
Pigeonhole sort	255.0	129.4	128.6	131.1	129.3	130.9	130.6	135.7	160.5	159.8
Counting sort	252.9	174.5	179.2	179.5	175.2	174.1	172.4	177.4	175.0	209.4

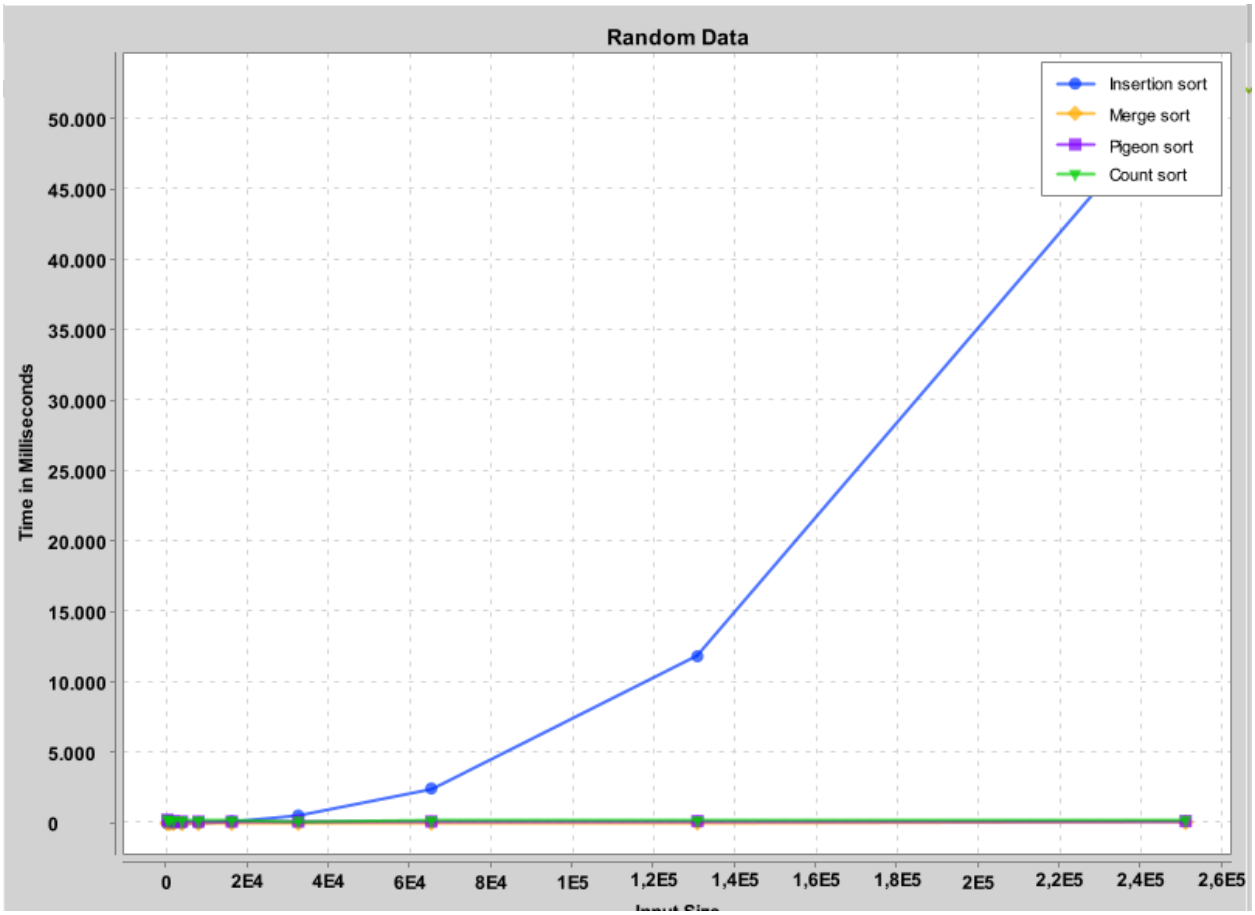

Figure 1: Plot of the random data

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	0.0	0.0	0.0	0.0	0.1	4.2
Merge sort	0.0	0.2	0.0	0.0	1.6	1.8	4.0	9.9	27.5	57.2
Pigeonhole sort	235.6	146.0	143.0	146.4	147.9	147.5	143.2	145.9	139.2	188.2
Counting sort	247.5	156.1	157.0	157.3	163.0	157.5	160.6	161.8	198.6	183.2

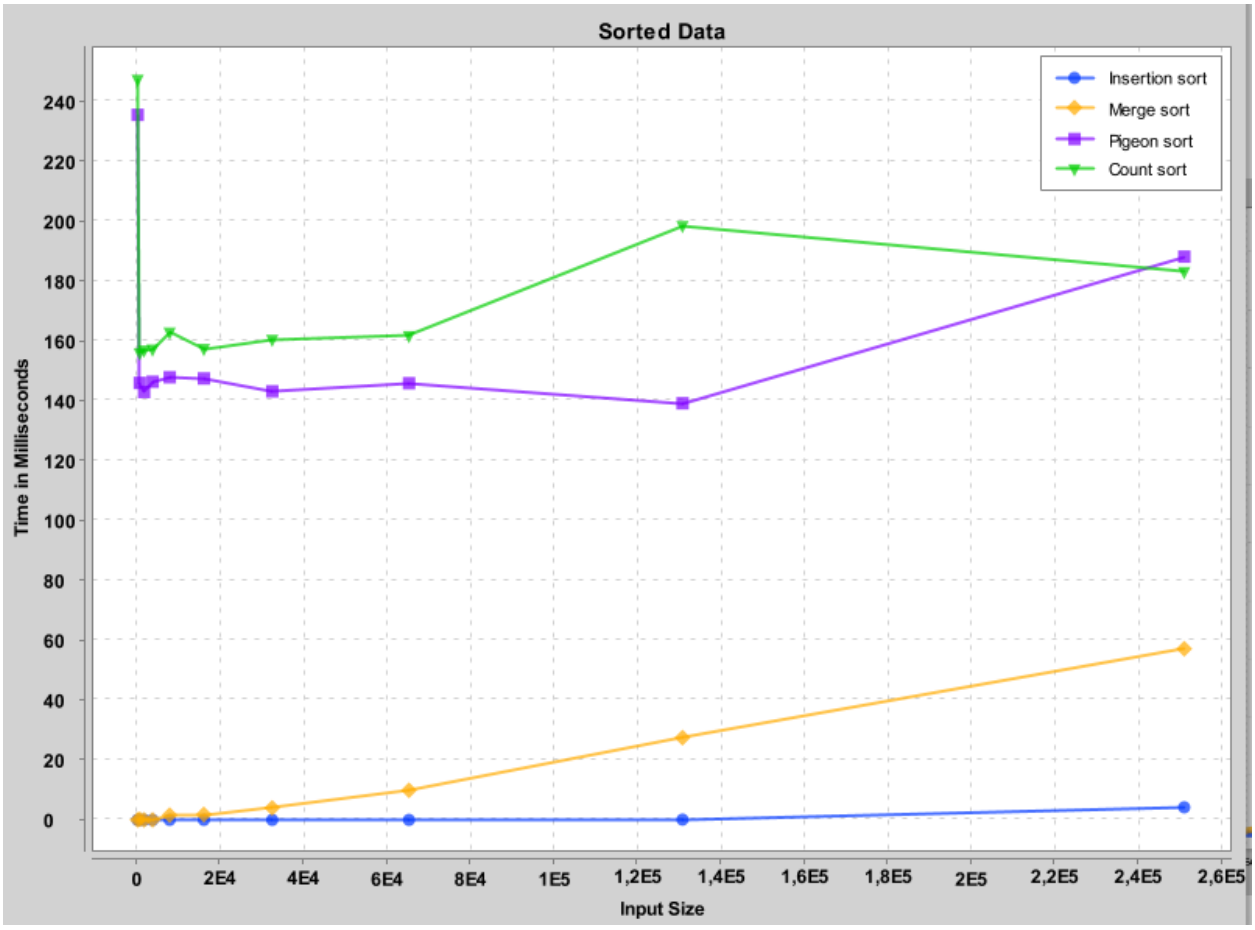

Figure 2: Plot of the sorted data

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

Algorithm	Input Size									
	512	1024	2048	4096	8192	16384	32768	65536	131072	251281
Insertion sort	0.8	1.6	5.1	17.4	58.8	249.3	972.3	3874.2	16189.2	61646.0
Merge sort	0.0	0.0	0.0	0.0	0.0	0.9	5.3	11.6	26.0	59.2
Pigeonhole sort	246.0	133.8	137.2	132.3	133.5	131.5	130.7	132.9	155.6	146.1
Counting sort	255.3	169.1	166.7	168.3	167.7	174.8	170.3	175.2	168.6	212.1

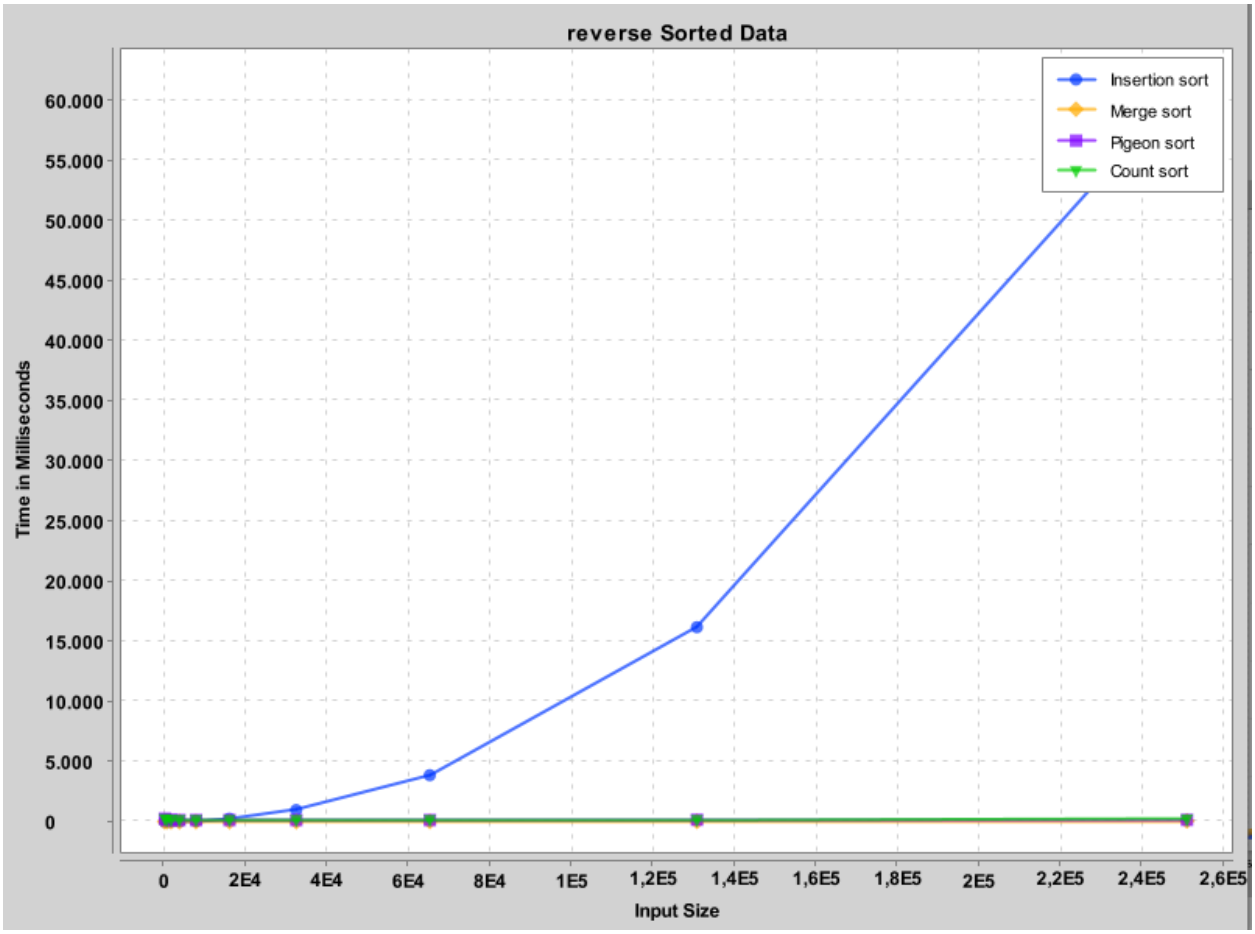

Figure 3: Plot of the reverse sorted data

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	$\Omega(n + k)$	$\Theta(n + k)$	$O(n + k)$

INSERTION SORT

Insertion Sort is an easy-to-implement, stable sorting algorithm with time complexity of $O(n^2)$ in the average and worst case, and O(n) in the best case. For very small n, Insertion Sort is faster than more efficient algorithms such as Quicksort or Merge Sort.

In average case : $\frac{1}{4}n^2 - \frac{1}{4}n$

The highest power of n in this term is n^2; the time complexity for shifting is, therefore, $O(n^2)$. This is also called "quadratic time".

In the worst case for insertion, the elements are sorted completely descending at the beginning. In each step, all elements of the sorted sub-array must, therefore, be shifted to the right so that the element to be sorted – which is smaller than all elements already sorted in each step – can be placed at the very beginning.

If the elements already appear in sorted order, there is precisely one comparison in the inner loop and no swap operation at all. With n elements, that is, n-1 steps (since we start with the second element), we thus come to n-1 comparison operations.Therefore: The best-case time complexity of Insertion Sort is: O(n)

MERGE SORT

Time complexity of Merge Sort is O(n*Log n) in all the 3 cases (worst, average and best) as merge sort always divides the array in two halves and takes linear time to merge two halves.

And that is regardless of whether the input elements are presorted or not. Merge Sort is therefore no faster for sorted input elements than for randomly arranged ones.

PIGEONHOLE SORT

Pigeonhole sorting is a sorting algorithm that is suitable for sorting lists of elements where the number of elements and the number of possible key values are approximately the same. It requires O(n + N) time where n is number of elements in input array and 'N' is number of possible values in array.

(Note: The ONLY difference between counting and pigeonhole sort is that in the secondary sort we move items twice i.e one on the input array and other in the bucket array. While in first sort we only move the items once.)

COUNTING SORT

The time complexity of counting sort algorithm is O(n+k) where n is the number of elements in the array and k is the range of the elements.The algorithm contains one or more loops that iterate to n and one loop that iterates to k.
Counting sort is most efficient if the range of input values is not greater than the number of values to be sorted. In that scenario, the complexity of counting sort is much closer to O(n), making it a linear sorting algorithm.
For sorted input sequences, the measurements correspond to the expected linear time complexity O(n + k).
For random input sequences, the measurements are slightly higher
When the array size doubles, the time required increases by a factor of about 2.1 to 2.2.
Input sequences sorted in descending order are sorted minimally slower than those sorted in ascending order.
Why Is Counting Sort Faster for sorted Elements Than for random Ones?
Let's explain with an example:
An auxiliary array with half a billion elements is 2 GB in size. If its elements are incremented in random order, a new cache line (typically 64 bytes) must be exchanged between RAM and CPU cache for almost every element. The larger the array, the lower the probability that the required cache line is in the CPU cache.
In contrast, if the array is incremented from front to back (or from back to front), 16 consecutive int values can be loaded from and written to the RAM in a single 64-byte block.
Why Is Counting Sort Faster for Items Sorted in Ascending Order Than for Items Sorted in Descending Order? If elements are sorted in ascending order, they are not changed and do not have to be written back to RAM. With elements sorted in descending order, every element of the array changes, so the whole array has to be written back into RAM once.

Table 5: Auxiliary space complexity of the given algorithms.

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

Space complexity of insertion sort:

The space complexity of Insertion Sort is constant since we do not need any additional memory except for the loop variables i and j and the auxiliary variable element to sort. This means that – no matter whether we sort ten elements or a million – we always need only these three additional variables. Constant complexity is noted as O(1).

Space complexity of merge sort:

In the merge phase, elements from two subarrays are copied into a newly created target array. In the very last merge step, the target array is exactly as large as the array to be sorted. Thus, we have a linear space requirement: If the input array is twice as large, the additional storage space required is doubled.
(In given pseudocode "line 13 = C: array" used to obtain the space complexity.)

Space complexity of pigeonhole sort:

This algorithm requires a lot of space even for a small but widely spread data. Pigeonhole sort algorithm requires two arrays. The first one is required to store the input elements so its size n.The second one is the pigeonhole array and has a size equla to range k. Space complexity becomes $O(n + k)$
(In given pseudocode "line 6 = holes: array ← [1 ,. . ., range]" used to obtain the space complexity)

Space complexity of counting sort:

The simplified algorithm requires an additional array of size k; therefore: The space complexity of the simplified counting sort algorithm is: O(k) In addition to the auxiliary array of size k, the general algorithm requires a temporary target array of size n; thus: The space complexity of the general counting sort algorithm is: O(n + k)
(In given pseudocode "2: count ← array of k + 1 zeros 3: output ← array of the same length as A" used to obtain the space complexity)

4 Discussion: Do the obtained results (running times of your algorithm implementations) match their theoretical asymptotic complexities?

For my insertion sort results:
*the runtime roughly quadruples when doubling the amount of input for random and reverse sorted elements,
*the runtime for sorted elements grows linearly and is significantly smaller.
So obtained results matched theoretical asymptotic complexities.

For my merge sort results:
*In all cases, the runtime increases approximately linearly with the number of elements, thus corresponding to the expected quasi-linear time – O(n log n).
Normally for sorted elements, merge Sort is about three times faster than for random elements.However, the number of comparison operations differs by only about one third. For elements reverse sorted, merge sort needs a little more time than for elements sorted in ascending order.But the difference between random-sorted-reverse sorted data in my tables and plots is not obvious. This may be due to some memory and cache issues.

For my pigeonhole sort results:
complexity of pigeonhole sort doesn't solely depend on the input size because range of data greatly exceeds input size so we should get values so close to each other. In my random and reverse sorted data plots and tables results are as expected. But in sorted data after first data size (512) there is a deviation which i think is because of cache.
For my counting sort results:
Because of memory and cache issues, counting sort results there are some deviations but generally the results are as expected. for the same reason (doesn't solely depend on the input size because range of data greatly exceeds input size) as pigeonhole sort we should get values so close to each other.

Obtained results matched their theoretical asymptotic complexities with some minor exceptions.

References

- https://www.happycoders.eu/algorithms/
- https://iq.opengenus.org/
- https://en.wikipedia.org/wiki/Sorting$_a lgorithm$/
- https://piazza.com/class/kyzqhsc5lck76h?cid=27
- https://www.c-sharpcorner.com/forums/read-column-wise-in-csv-file-using-java/