Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2022 Spring

Programming Assignment 1

March 9, 2022

Student name:
Ecem ÇIRAKOĞLU

Student Number:
b21821665

1 Problem Definition

The aim of this assignment is to compare the sorting algorithms and to examine the relationship between the running time of the sorting algorithm implementations with their theoretical asymptotic complexities.

2 Solution Implementation

Let me briefly explain the implementation of my code, I created classes for each sorting algorithm. Then in the 'Main' class, I applied each sorting algorithm 10 times with the given input size. I recorded every sorting algorithm's run time then I took the average. And I did this the whole process for random, sorted, and reverse sorted data of CSV file which is read by the program.

2.1 Insertion Sort Algorithm

```
1  import java.util.Arrays;
2  class InsertionSort{
3      public void insertionSort(int[] arr){
4          int i;
5          int key;
6          for(int j=1; j<arr.length; j++){
7              key = arr[j];
8              i= j-1;
9              while(i>0 && arr[i]>key){
10                 arr[i+1] = arr[i];
11                 i=i-1;
12             }
13             arr[i+1] = key;
14         }
15     }
16 }
```

This algorithm assumes key as a second element. And starts to traverse in first element in the array. Compares the key with the first element, if the first element is greater than key, then places key in front of the first element. Repeats this step until it reach the end of the array.
Time complexity of insertion sort for best case is $O(n)$ n-1 compares and 0 exchange, for average and worst case $O(n^2)$ because we have second loop. Auxiliary space complexity is $O(1)$ because no array was created.
*I applied this codes from the pseudo codes.

2.2 Merge Sort Algorithm

```
18 import java.util.ArrayList;
19 class MergeSort{
```

```
20      public static void Merge(int[] arr,int startIndex,int endIndex){
21          ArrayList<Integer> mergedSortedArray = new ArrayList<Integer>();
22
23          int midIndex = (endIndex + startIndex) / 2;
24          int leftIndex = startIndex;
25          int rightIndex = midIndex + 1;
26
27          while (leftIndex <= midIndex && rightIndex <= endIndex) {
28              if (arr[leftIndex] <= arr[rightIndex]) {
29                  mergedSortedArray.add(arr[leftIndex]);
30                  leftIndex++;
31              } else {
32                  mergedSortedArray.add(arr[rightIndex]);
33                  rightIndex++;
34              }
35          }
36          while (leftIndex <= midIndex) {
37              mergedSortedArray.add(arr[leftIndex]);
38              leftIndex++;
39          }
40          while (rightIndex <= endIndex) {
41              mergedSortedArray.add(arr[rightIndex]);
42              rightIndex++;
43          }
44          int i = 0;
45          int j = startIndex;
46          while (i < mergedSortedArray.size()) {
47              arr[j] = mergedSortedArray.get(i++);
48              j++;
49          }
50      }
51      public static void mergeSort(int[] arr,int lo,int hi){
52          if (lo < hi && (hi - lo) >= 1) {
53              int mid = (hi + lo) / 2;
54              mergeSort(arr,lo, mid);
55              mergeSort(arr,mid + 1, hi);
56              Merge(arr,lo, hi);
57          }
58      }
59  }
```

Recursively divides the array into halves until we reach the base case of the array with one element and each step gives $\log(n)$ component. After that, the merge function takes the sorted sub-arrays gradually merges to sort the entire array and the n component comes from here because its comparisons at each step. So if we combine this, we get $O(n \log(n))$. In merge sort best,average and worst cases are same: $O(n \log(n))$ And auxiliary space complexity is $O(n)$ because code creates arrays and sizes goes like n, n/2, n/4... *Some of the parts I took it from the pseudo codes.

2.3 Pigeonhole Sort Algorithm

```
class PigeonholesSort{
   public void pigeonholesSort(int[] arr){
        int n=arr.length;
        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);
        //An element arr[i] is put in phole at index arr[i]-min.
        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;
        }
    }
}
```

Find min and max elements in the array. Then find the range. Create an empty array with a length of the range. Traverse the array and put each element in its pigeonhole. Then put every element to the original array by their order. Best case, if the data is sorted $O(n+k)$, worst case, if the data is length is small but the range is too large $O(n+k)$ and average case is $O(n+k)$ *k is range And auxiliary space complexity is $O(k)$ I created a array with range size.

2.4 Counting Sort Algorithm

```
class CountingSort{
    public void countingSort(int[] arr,int size){
        int[] output = new int[size+1];
        int max= arr[0];
        for(int i = 1; i<size; i++){
            if(arr[i]>max)
                max=arr[i];
        }
        int[] count = new int[max+1];
```

```
96          for(int i=0; i<max; ++i){
97              count[i] =0;
98          }
99          for(int i =0; i<size; ++i){
100             count[arr[i]]++;
101         }
102         for(int i=1; i<=max; ++i){
103             count[i] += count[i-1];
104         }
105         for(int i=size-1;i>=0;i--){
106             output[count[arr[i]]-1] =arr[i];
107             count[arr[i]]--;
108         }
109     }
110 }
```

Stored the count of each element at their respective index in the count array. If the ith an element is not present in the array, then 0 is stored in ith position. Stored the cumulative sum of the elements of the count array. (It helps in placing the elements into the correct index of the sorted array.) Found the index of each element of the original array in the count array then put it to the output array and decrease the count of that element in that index. Counting sort has the same best,worst and the average time complexity: $O(n+m)$ because my code have loops with max number(m) and the number of given array size. And auxiliary space complexity is $O(n+m)$ because I created two array 'count' and 'output' one of them size is length of the array and the other ones is maximum number(m) in that array.

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	0.2	0.1	0.6	2.8	12.2	53.6	199.5	764.5	3186.4	13211.3
Merge sort	0.2	0.3	0.3	0.6	1.0	2.5	4.6	11.4	24.1	44.8
Pigeonhole sort	121.4	76.8	76.8	74.0	73.6	73.5	79.8	71.6	78.8	72.5
Counting sort	223.6	166.0	167.2	165.4	163.9	170.7	180.9	185.2	198.4	177.2

In insertion time taken for sorting 512 inputs is on average 0.2 millie-seconds and the time taken for sorting 251281 inputs is 13211.3 Millie-seconds.The number of inputs increases, the time is taken to sort the data also increases.Merge sort in random data runs less than $O(n \log n)$ In Pigeonhole sort and Counting sort when the given input size increases, mostly sorts more faster.

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	0.1	0.0	0.2	0.0	0.0	0.0	0.1	0.1	0.2
Merge sort	0.2	0.1	0.2	0.9	1.7	3.1	7.0	16.3	41.8	75.0
Pigeonhole sort	0.0	0.1	0.3	0.1	0.0	0.0	0.0	0.1	0.9	122.6
Counting sort	0.1	0.1	0.1	0.2	0.1	0.0	0.1	0.1	1.5	250.1

When the data is sorted in insertion sort even if the number of inputs increases, the run time does not change that much and it works very fast. Merge sort increases when the input increase but when the input size is too large Pigeonhole and counting works very fast but as you can see with largest size run time increases a lot.

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	0.2	0.2	1.0	4.4	21.0	83.4	358.1	1361.4	5445.2	20791.0
Merge sort	0.3	0.1	0.1	0.4	1.2	2.5	5.2	9.2	20.0	42.9
Pigeonhole sort	0.1	0.1	1.8	9.0	9.6	35.2	155.4	67.3	87.7	72.3
Counting sort	349.5	160.3	171.1	159.0	160.5	163.3	225.2	156.9	191.8	185.7

In reverse sorted data I firstly stored reversely the whole elements and then created array as many as the number of entries. As we can see in insertion sort when the input size increase a lot, then program runs slower.For reversely sorted and large data, Insertion sort is not a good option.

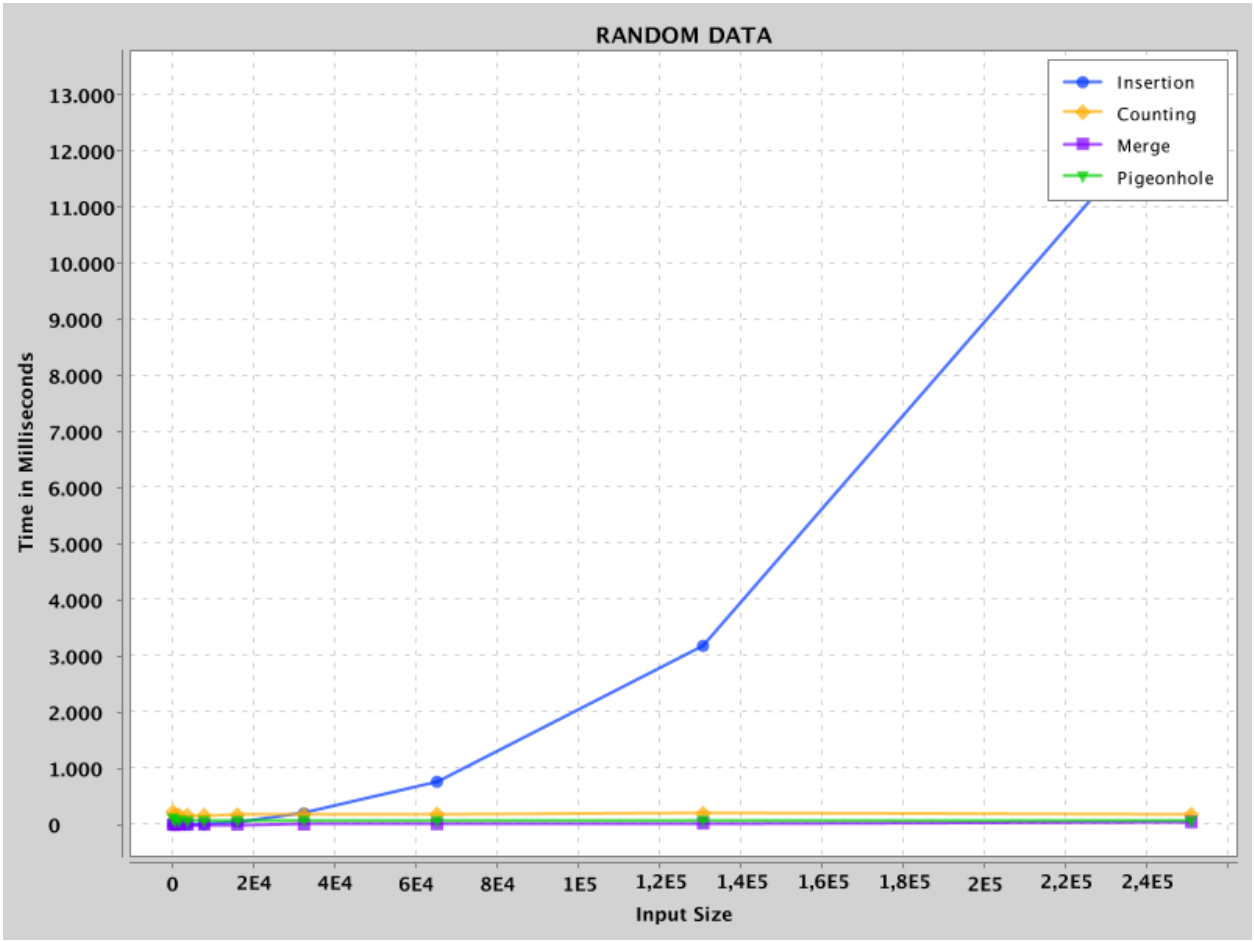

Figure 1: Plot of the average running time of each sorting algorithm with random data

Plots shows that the Insertion sort which is the slowest is approximately $O(n^2)$ and other sorting algorithms are faster.
*According to this plots Merge is faster than Pigeonhole, Pigeonhole is faster than Counting and Counting is faster than Insertion sort.

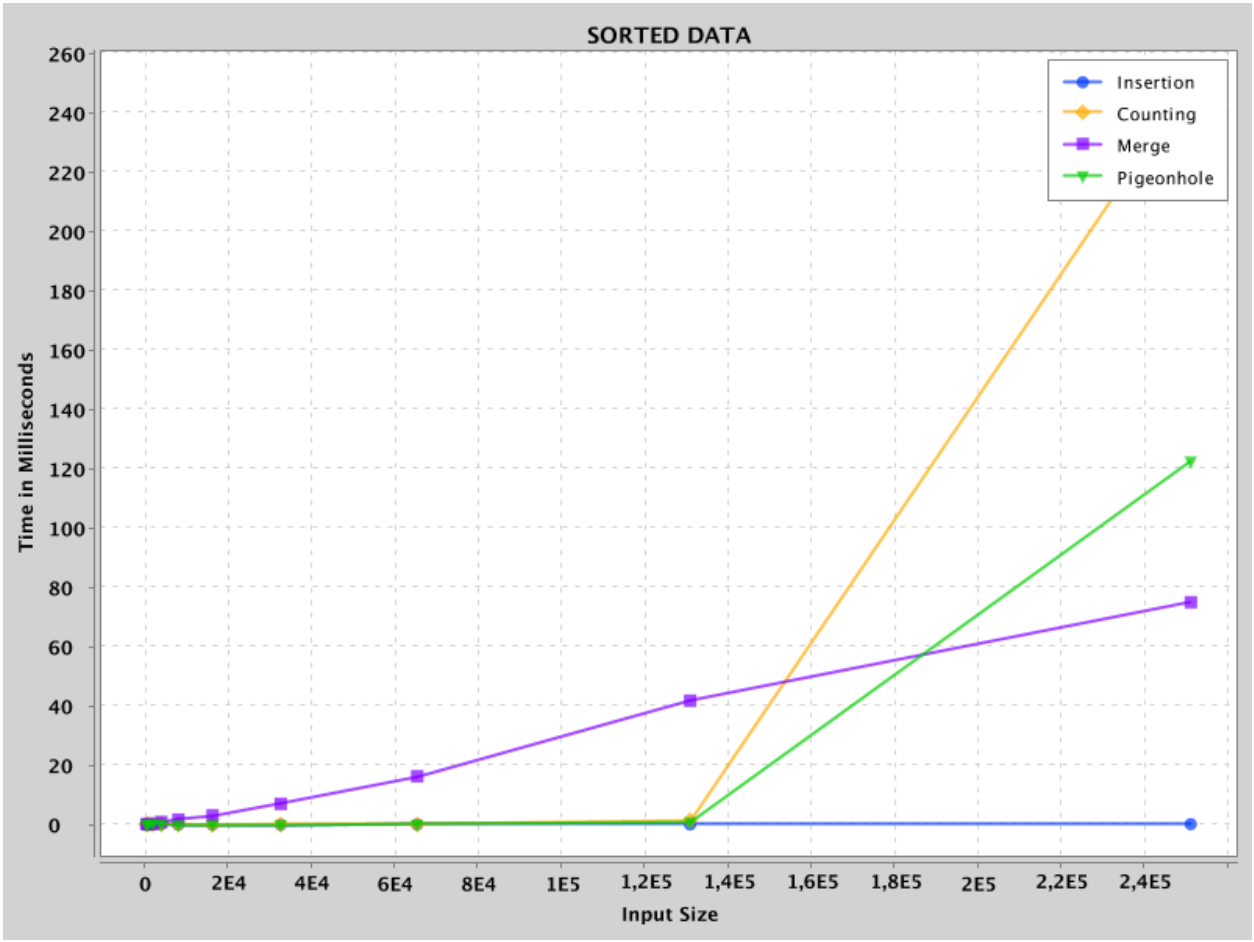

Figure 2: Plot of the average running time of each sorting algorithm with sorted data

As we can see in this graphic Merge sort is $n \log(n)$, Insertion sort runs in constant time n-1 compares 0 change, Pigeonhole and Counting sorts' run time suddenly increases a lot when input size increases.
*This happens because first I sorted the whole array and then picked number of given element size. So this reason in first steps in my array there where just like 1,2, etc little numbers. The plots seems like constant at this time then large number come to the array plots increase suddenly.

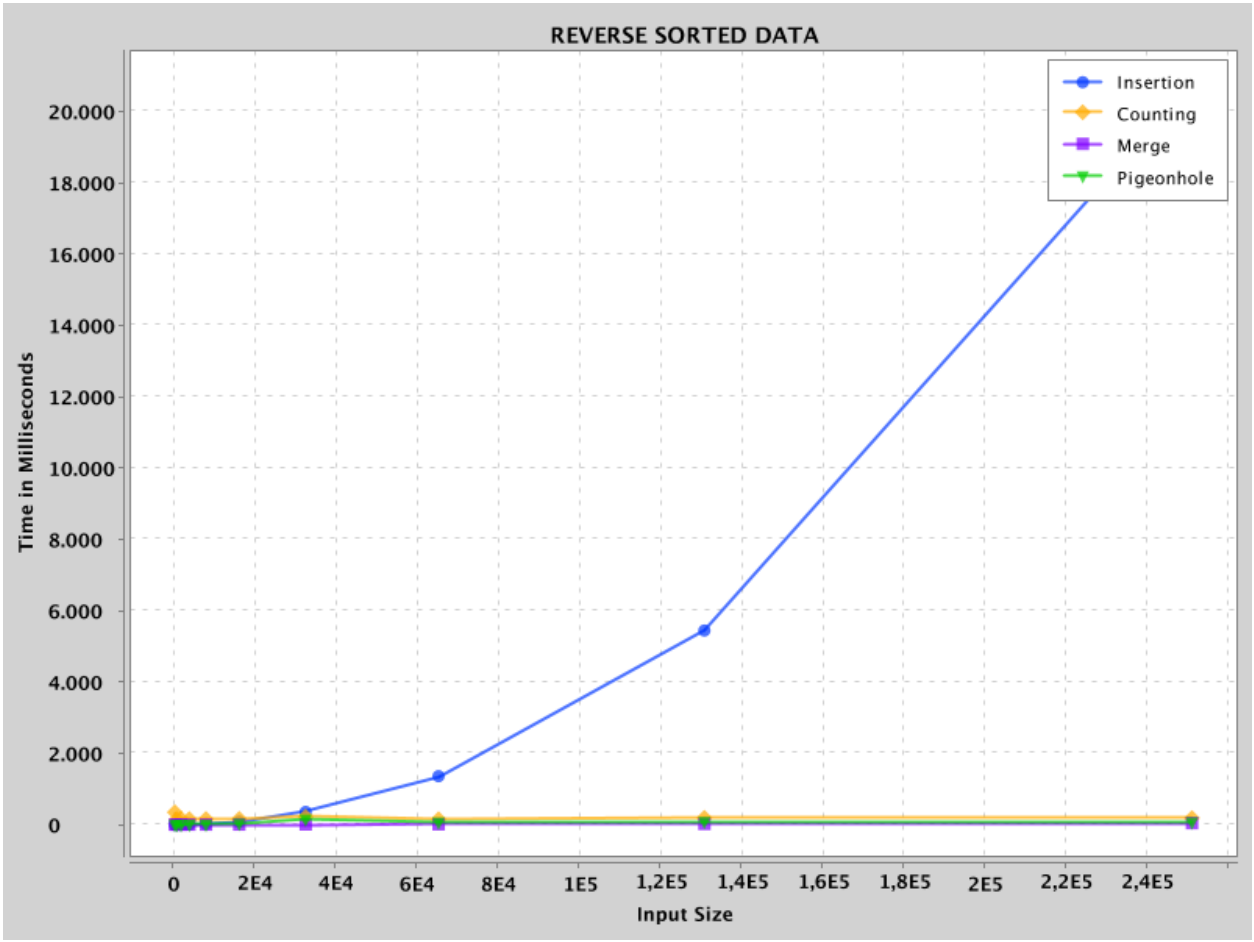

Figure 3: Plot of the average running time of each sorting algorithm with reverse sorted data.

When the data is reverse ordered insertion sort is approximately $O(n^2)$ complexity and as we can see insertion sort is more slowly than other sorting algorithms.It's not useful when the array is too large and reverse ordered.
*According to this plots when the data is reverse ordered Merge is faster than Pigeonhole, Pigeonhole sort is faster than Counting and Counting sort is faster than Insertion sort.

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	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$
Counting Sort	$\Omega(n+m)$	$\Theta(n+m)$	$O(n+m)$

*k is the range of input,m is the maximum element of the array

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	$O(n+m)$

*Created two arrays ('output' array with the size of array(n) and 'count' array with the maximum element(m) size) in counting sort algorithm and in pigeonhole sort algorithm I created one array with the size of the range(k).

3.1 Analysis and Discussion

• What are the best, average, and worst cases for the given algorithms in terms of the given input data to be sorted?

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

Insertion Sort

Best Case: When the data is sorted it runs at constant time.n-1 compares and 0 exchanges.

Average Case:When the data is random $O(n^2)$ time. We determine what is the average number of inversions will be for the record in position i.This could be 0,1 or 2 or anything up to i it is i/2 positions out order. And this had to be done from 1 to n-1 $\sum_{i=1}^{n-1} i/2 \cong \mathrm{O}(\mathrm{n}^2)$

Worst Case:When the data is reversely sorted (n-1)*(n-1)/2 $\cong \mathrm{O}(\mathrm{n}^2)$

Merge Sort

The divide-and-conquer contributes a log(n) factor. You divide the array in half ($\log n$) times, and each time you do, for each segment, you have to do a merge on two sorted array. Merging two sorted arrays is $O(n)$ Best Case: $O(n \log n)$ Average Case: $O(n \log n)$ Worst Case: $O(n \log n)$ *For sorted data, Merge Sort is about three times faster than for random data.

The obtained results match their theoretical asymptotic complexities.

Pigeonhole Sort

Best Case: When the data reversely sorted or sorted data.It is very efficient for sorting lists of

elements where the number of elements and the number of possible key values are approximately the same. $O(n+k)$ Average Case:When the k and n equally likely we can get this case $O(n+k)$ but it is not happening in this practice. Worst Case: When the data set length is small and k value is very big. $O(n+k)$
*We can't exactly can say the worst and average cases are happening in practice.
The obtained results match their theoretical asymptotic complexities.When the data sorted or reversely sorted and the input size is not very big we can get the best case.

Counting Sort

Best Case: When the data presorted, the measurements correspond to the expected linear time complexity $O(n+m)$.
Average Case:Fix N and take various values of m from 1 to infinity; in this scenario, m computes to (m+1/2), and the average case is n+(m+1)/2. However, because k approaches infinity, k is the essential element. Similarly, varying n reveals that both n and k are equally dominating, resulting in $O(n+m)$ as the average case.
Worst Case:When the data is skewed that is the largest element is significantly large than other elements.$O(n+m)$
*We can't exactly can say the worst and average cases are happening in practice.
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. So we can say sorted data is more faster than reverse sorted data in practice.
The obtained results match their theoretical asymptotic complexities.In practice when the data sorted or reversely sorted and the input size is not very big we can get the best case.

4 Notes

-Although Pigeon sort and Counting sort have equal complexity in theory, they are different in practice.Counting sort is more faster than pigeonhole sort in practice.
-Pigeonhole sort is a non-comparison based sort making it faster in application.It isn't easy to know the range of the numbers to sort.This number might only work with zero and positive integers
-When range is comparable to number of input elements Pigeonhole sort has lesser time complexity than Merge sort.
-Counting sort and pigeonhole sort are in-place algorithms.
-Compare based algorithm has worst case at least $\log n! \cong n \log n (Stirling's Formula)$
-One of the biggest lessons I learned in this assignment was that the sorting algorithms can change from computer to computer, but after a good algorithm is established, it works quickly even on a bad computer.

References

- https://stackoverflow.com/questions/1051640/correct-way-to-add-external-jars-lib-jar-to-an-intellij-idea-project

- https://www.geeksforgeeks.org/counting-sort/
- https://www.jetbrains.com/help/idea/
- https://www.geeksforgeeks.org/pigeonhole-sort/
- https://www.sanfoundry.com/pigeonhole-sort-multiple-choice-questions-answers-mcqs/
- https://iq.opengenus.org/insertion-sort-analysis/
- https://tr.overleaf.com/latex/templates/symbol-table/fhqmttqvrnhk