Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2022 Spring

Programming Assignment 1

March 4, 2022

Student name:
Süleyman Yilmaz

Student Number:
b21946735

1 Problem Definition

Compare two different kind of sorting algorithms such as comparison based sorting algorithms (Insertion , Merge) and non-comparison based sorting algorithms (Pigeonhole , Counting) and analyze their performances in several situations.

2 Solution Implementation

Java codes of algorithms explained here.

2.1 Insertion Sort

```
1   public static void insertionSort(int[] arr){
2       for (int i = 0; i < arr.length; i++) {
3           int currentElement = arr[i];
4           for (int j = i-1; j >=0; j--) {
5               if (arr[j] > currentElement ) {
6                   arr[j+1] = arr[j];
7                   arr[j] = currentElement;
8               }
9               else{break;}
10          }
11      }
12  }
```

The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

2.2 Merge Sort

```
13      public static void mergeSort(int[] arr){
14          Integer[] sortedNums = mergeSortRecur(arr,0,arr.length-1);
15          for (int i = 0; i < sortedNums.length; i++)
16              arr[i] = sortedNums[i];
17      }
18      public static Integer[] mergeSortRecur(int[] arr,int low,int upp){
19          int mid = low + (int)((upp-low)/2); // set mid point
20          Integer[] lower; // lower array
21          Integer[] upper; // upper array
22          if (low <upp ) { // if there is more than one element in array
23              lower = mergeSortRecur(arr, low, mid);
24              upper = mergeSortRecur(arr, mid+1, upp);
25              return  merge(lower,upper); // combine 2 arrays
26          }
27          Integer [] ret = new Integer[upp-low+1]; // copy array
28          int index = 0;
29          for (int i = low; i < upp+1; i++)
30              ret[index++] = arr[i];
31          return ret;
32      }
33      public static Integer[] merge(Integer[] lower , Integer[] upper){
34          Integer[] newArr = new Integer[lower.length+upper.length];
35
36          int low =0; // current index in lower array
37          int upp = 0; // current index in upper array
38          for (int i = 0; i < newArr.length; i++) {
39              if ( low < lower.length && upp < upper.length) {
40                  if (lower[low] < upper[upp] ) {newArr[i] = lower[low];low++;}
41                  else{newArr[i] = upper[upp];upp++;}
42              }
43              else{
44                  if ( low < lower.length) {newArr[i] = lower[low];low++;}
45                  else if ( upp < upper.length){newArr[i] = upper[upp];upp++;}
46              }
47          }
48          return newArr; // return merged array
49      }
```

Divide the unsorted list into n sublists, each containing one element (a list of one element is considered sorted).
Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.

2.3 Pigeonhole Sort

```
51      public static void pigeonholeSort(int arr[]){
52          int n = arr.length;
53          int max =Collections.max(Arrays.stream(arr).boxed().collect(Collectors
                .toList()),null);
54          int min =  Collections.min(Arrays.stream(arr).boxed().collect(
                Collectors.toList()),null);
55
56          int range = max - min + 1;
57          LinkedList<Integer>[] anArray =  new LinkedList[range];
58
59          for (int i = 0; i < n; i++){
60              if (anArray[arr[i]-min] == null) {
61                  anArray[arr[i]-min] = new LinkedList<>();
62              }
63              anArray[arr[i]-min].add(arr[i]);
64          }
65
66          int ind = 0;
67          for (int i = 0; i < range; i++) {
68              if (anArray[i] !=null) {
69                  for (int j : anArray[i]) {
70                      arr[ind++] = j;
71                  }
72              }
73          }
74      }
```

Given an array of values to be sorted, set up an auxiliary array of initially empty "pigeonholes", one pigeonhole for each key in the range of the keys in the original array.
Going over the original array, put each value into the pigeonhole corresponding to its key, such that each pigeonhole eventually contains a list of all values with that key.
Iterate over the pigeonhole array in increasing order of keys, and for each pigeonhole, put its elements into the original array in increasing order.

2.4 Counting Sort

```
75      public static int[] countingSort(int[] arr){
76          int max =Collections.max(Arrays.stream(arr).boxed().collect(Collectors
                .toList()),null);
77          int min =  Collections.min(Arrays.stream(arr).boxed().collect(
                Collectors.toList()),null);
78
79          int range = max - min + 1;
80          int[] count = new int[range];
81          int[] out = new int[arr.length];
82          for (int i = 0; i < arr.length; i++) {
83              count[arr[i]-min]++;
84          }
85
86          for (int i = 1; i < count.length ; i++) {
87              count[i] +=count[i-1];
88          }
89          for (int i = 0; i < arr.length; i++) { // create output array
90              out[count[arr[i]-min]-1] = arr[i];
91              count[arr[i]-min]--;
92
93          }
94
95          return out;
96      }
```

Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

3 Results, Analysis, Discussion

Complexity analysis tables:

Table 1: 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	$\Omega(n + k)$	$\Theta(n + k)$	$O(n + k)$

Table 2: 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	$O(n + k)$

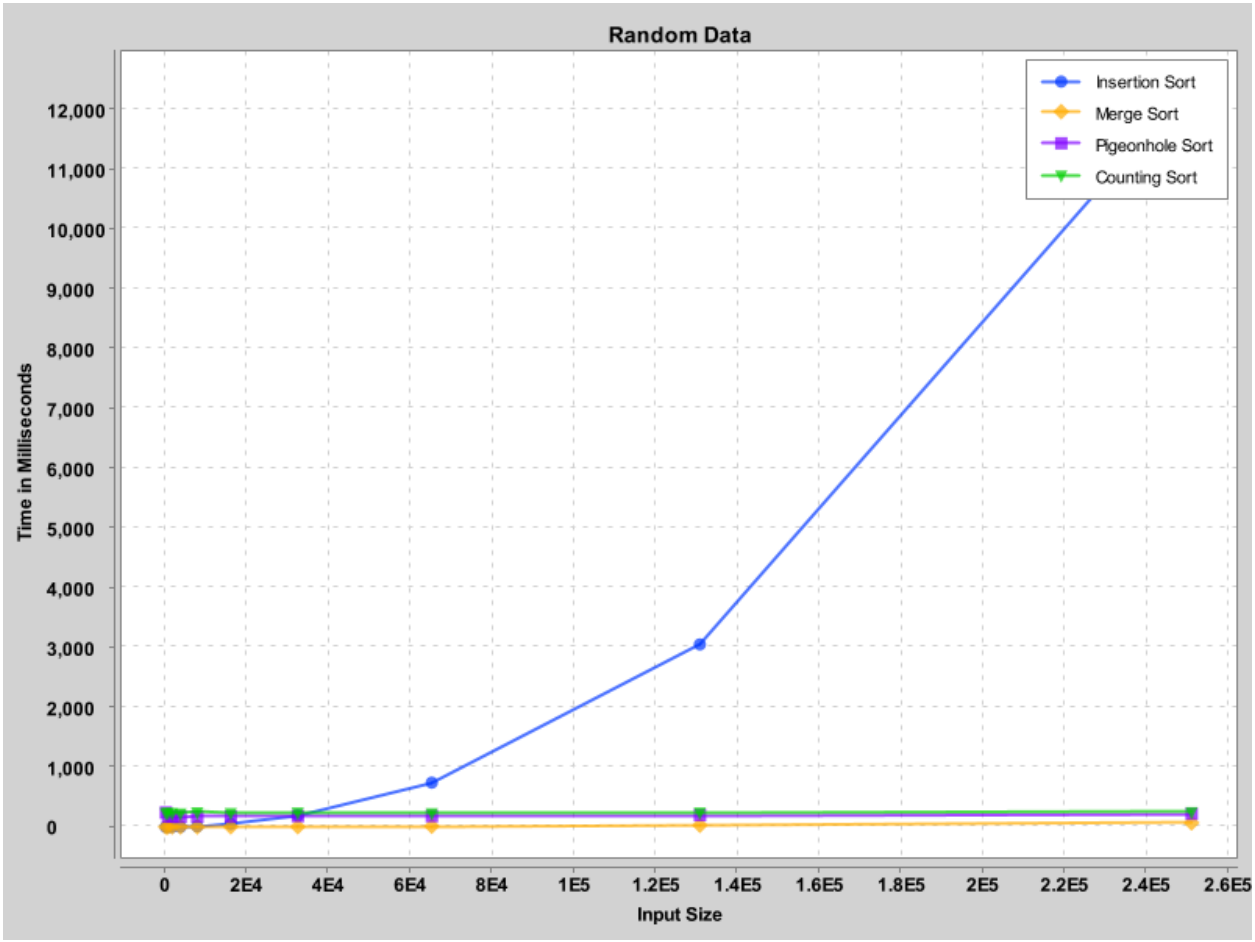

Figure 1:

Algorithm	512	1024	2048	4096	8192	16384	32768	65536	131072	251281
Insertion Sort	0	0	0	3	12	52	191	741	3053	12443
Merge Sort	0	0	0	0	1	2	5	13	28	68
Pigeonhole Sort	239	171	168	168	171	172	180	179	195	217
Counting Sort	238	240	235	231	250	239	246	236	246	255

Figure 2:
For random data:
Insertion Sort increase exponential with input increasing
Merge Sort increase logarithmic with input increasing
Pigeonhole and Counting sorts are do not change with input increasing
For first input pigeonhole is higher than other inputs, reason of that is cache misses.

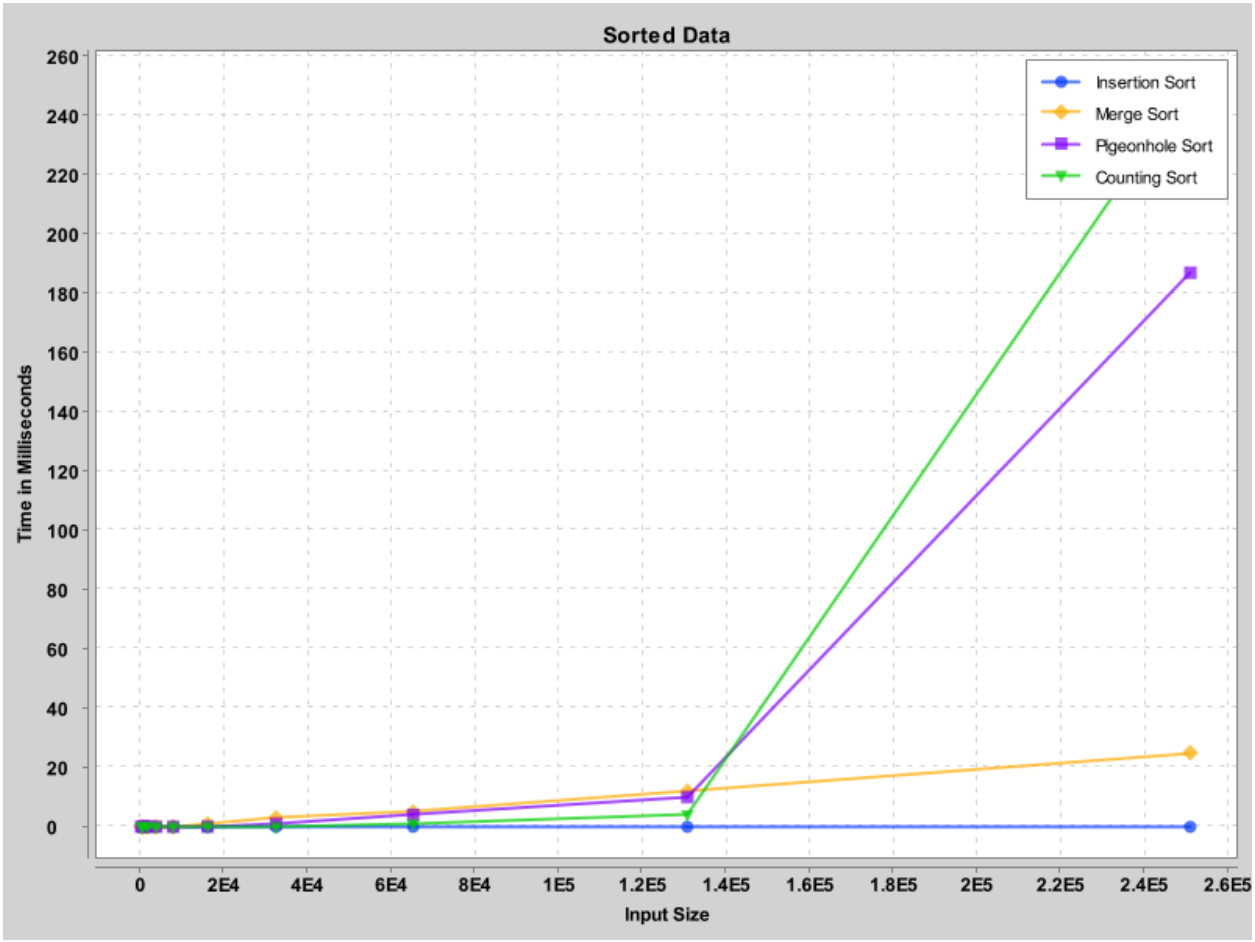

Figure 3:

Algorithm	512	1024	2048	4096	8192	16384	32768	65536	131072	251281
Insertion Sort	0	0	0	0	0	0	0	0	0	0
Merge Sort	0	0	0	0	0	1	3	5	12	25
Pigeonhole Sort	0	0	0	0	0	0	1	4	10	187
Counting Sort	0	0	0	0	0	0	0	1	4	251

Figure 4:

For sorted data:

Insertion Sort check if data is sorted immediately

Merge Sort increase logarithmic with input increasing but faster than random data because of less comparisons

Pigeonhole and Counting sorts are do not change with input increasing they change with increase of range. This shows us range difference is huge between first and second half of input data

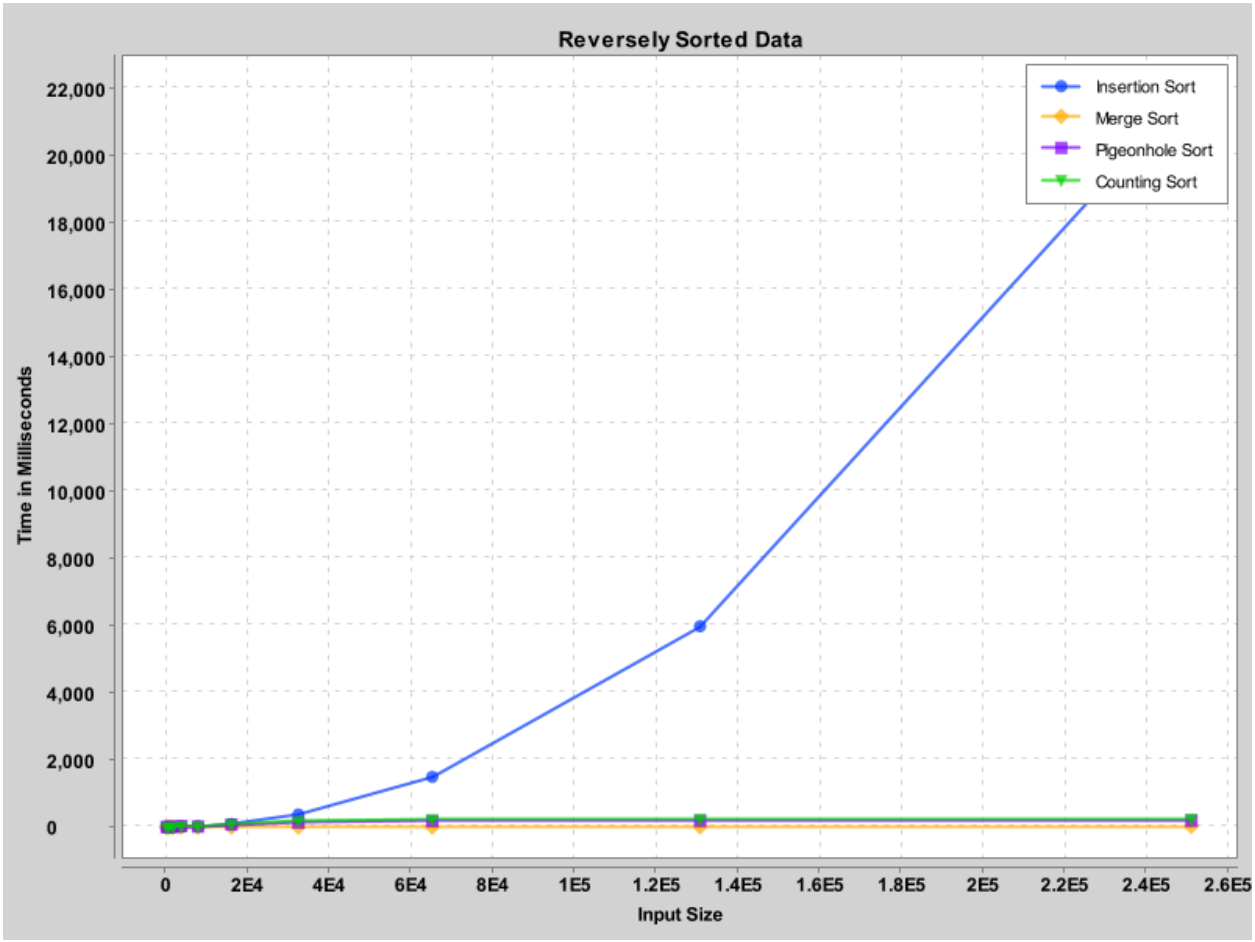

Figure 5:

4 Notes

***Reversely sorted data table is below, Latex drove me crazy so i had to leave it like that.

For merge sort I preferred to use Integer class instead of primitive int in order to see how it will effect the performance of algorithm and class type decreased performance by half.

References

- https://iq.opengenus.org/time-and-space-complexity-of-counting-sort/
- https://www.geeksforgeeks.org/counting-sort/
- https://en.wikipedia.org/wiki/Pigeonhole_sort#:~:text=Pigeonhole%20sorting%20is%20a%20sorting,(n%20%2B%20N)%20time.

Algorithm	512	1024	2048	4096	8192	16384	32768	65536	131072	251281
Insertion Sort	0	0	1	5	23	93	376	1498	5985	22034
Merge Sort	0	0	0	0	0	1	2	5	12	23
Pigeonhole Sort	0	0	2	29	15	72	134	177	184	187
Counting Sort	0	0	3	35	27	96	197	244	253	260

Figure 6:
For reversely sorted data:
Insertion Sort increase exponential with input increasing . It is even slower than random data because of reversely sorted data is worst case of insertion sort
Merge Sort increase logarithmic with input increasing
Pigeonhole and Counting sorts are do not change with input increasing they change with increase of range. This shows us range difference is increasing with increase of input and after 7. input it remains stable