Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2022 Spring

Programming Assignment 1

March 4, 2022

Student name:
Halil Bülke

Student Number:
b21945944

1 Problem Definition

The main objective of this assignment is to show the relationship between the running time of the algorithm implementations with their theoretical asymptotic complexities. We implement the 4 sorting algorithms and perform a set of experiments on the given datasets to show that the empirical data follows the corresponding asymptotic growth functions. We tried 4 different sorting algorithms for 10 different data sizes.

2 Solution Implementation

2.1 Insertion Sort

```
1    public static void InsertionSort(int[] array){
2        for (int i = 1; i < array.length; ++i) {
3            int keyItem = array[i];
4            int j = i - 1;
5            while (j >= 0 && array[j] > keyItem) {
6                array[j + 1] = array[j];
7                j = j - 1;
8            }
9            array[j + 1] = keyItem;
10       }
11   }
```

Insertion sort iterates, consuming one input element each repetition, and grows a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

The best case input is an array that is already sorted. In this case insertion sort has a linear running time O(n). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

The simplest worst case input is an array sorted in reverse order. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time $O(n^2)$.

The average case is also quadratic which makes insertion sort impractical for sorting large arrays. However, insertion sort is one of the fastest algorithms for sorting very small arrays.

2.2 Merge Sort

```
12      public static void mergeSort(int a[], int beginning, int end) {
13          if (beginning < end) {
14              int mid = (beginning + end) / 2;
15              mergeSort(a, beginning, mid);
16              mergeSort(a, mid + 1, end);
17              merge(a, beginning, mid, end);
18          }
19      }
20
21      public static void merge(int a[], int beg, int mid, int end) {
22          int i, j, k;
23          int n1 = mid - beg + 1;
24          int n2 = end - mid;
25
26          int[] LeftArray = new int[n1];
27          int[] RightArray = new int[n2];
28
29
30          for (i = 0; i < n1; i++) {
31              LeftArray[i] = a[beg + i];
32          }
33          for (j = 0; j < n2; j++) {
34              RightArray[j] = a[mid + 1 + j];
35          }
36
37          i = 0;
38          j = 0;
39          k = beg;
40
41          while (i < n1 && j < n2) {
42              if(LeftArray[i] <= RightArray[j]) {
43                  a[k] = LeftArray[i];
44                  i++;
45              }
46              else {
47                  a[k] = RightArray[j];
48                  j++;
49              }
50              k++;
51          }
52          while (i<n1) {
53              a[k] = LeftArray[i];
54              i++;
55              k++;
56          }
57
```

```
58          while (j<n2) {
59              a[k] = RightArray[j];
60              j++;
61              k++;
62          }
63      }
```

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.Most implementations produce a stable sort, which means that the order of equal elements is the same in the input and output.Merge sort algorithm that recursively splits the list (called runs in this example) into sublists until sublist size is 1, then merges those sublists to produce a sorted list. The copy back step is avoided with alternating the direction of the merge with each level of recursion.

2.3 Pigeonhole Sort

```
64      public static void pigeonholeSort(int[] array, int n) {
65
66          int min = array[0];
67          int max = array[0];
68          int range, i, j, index;
69
70          for(int a=0; a<n; a++)
71          {
72              if(array[a] > max)
73                  max = array[a];
74              if(array[a] < min)
75                  min = array[a];
76          }
77
78          range = max - min + 1;
79          int[] pigeonhole = new int[range];
80          Arrays.fill(pigeonhole, 0);
81
82          for(i = 0; i<n; i++)
83              pigeonhole[array[i] - min]++;
84          index = 0;
85          for(j = 0; j<range; j++)
86              while(pigeonhole[j]-->0)
87                  array[index++]=j+min;
88
89      }
```

This was my first pigeonhole sort algorithm but I couldn't run this because of the memory leak

```
90      public static void pigeonholeSort(int[] array) {
91          int n= array.length;
92          int min = array[0];
93          int max = array[0];
94          for(int a=0; a<n; a++){
95              if(array[a] > max)
96                  max = array[a];
97              if(array[a] < min)
98                  min = array[a];
99          }
100         int range =max - min +1;
101
102         ArrayList<Integer> output = new ArrayList<Integer>();
103         LinkedList<Integer>[] holes = new LinkedList[range];// I used array of
                linked lists
104
105         for (int i = 0; i < range; i++) {
106             if (holes[i] == null) {
107                 holes[i] = new LinkedList<Integer>();
108             }
109         }
110         for(int i =0;i<n;i++){
111             holes[array[i] - min].add(array[i]);
112         }
113         for(int j=0;j<range;j++){
114             if(holes[j]!=null) {
115                 output.addAll(holes[j]);
116             }
117         }
118         for (int i = 0; i < output.size(); i++)
119             array[i] = output.get(i);
120     }
```

Pigeonhole sorting is a sorting algorithm that is suitable for sorting lists of elements where the number of elements n and the length of the range of possible key values N are approximately the same. It requires O(n + N) time. It is similar to counting sort, but differs in that it "moves items twice: once to the bucket array and again to the final destination whereas counting sort builds an auxiliary array then uses the array to compute each item's final destination and move the item there.

Firstly for pigeonhole sort implementation I used array of linked list. But it requires too much memory. Even I increase the heap-size in JVM it couldn't run this algorithm so I had to use array. I think it takes a bit more time to use array instead of linked list or Arraylist but it took much less memory, and at least it runs.

2.4 Counting Sort

```
121     public static void countSort(int[] array, int size) {
122         int[] output = new int[size + 1];
123 
124         int max = array[0];
125         for (int i = 1; i < size; i++) {
126             if (array[i] > max)
127                 max = array[i];
128         }
129         int[] count = new int[max + 1];
130 
131         for (int i = 0; i < max; ++i) {
132             count[i] = 0;
133         }
134         for (int i = 0; i < size; i++) {
135             count[array[i]]++;
136         }
137 
138         for (int i = 1; i <= max; i++) {
139             count[i] += count[i - 1];
140         }
141         for (int i = size - 1; i >= 0; i--) {
142             output[count[array[i]] - 1] = array[i];
143             count[array[i]]--;
144         }
145 
146         for (int i = 0; i < size; i++) {
147             array[i] = output[i];
148         }
149     }
```

Counting sort is an algorithm for sorting a collection of objects according to keys that are small positive integers; that is, it is an integer sorting algorithm. It operates by counting the number of objects that possess distinct key values, and applying prefix sum on those counts to determine the positions of each key value in the output sequence. Its running time is linear in the number of items and the difference between the maximum key value and the minimum key value, so it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items.

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	8	8.8	9.3	17.3	42.3	130.9	487.9	2035.5	8395.4
Merge sort	0	0	1.6	4.8	6.4	8	11.2	20.3	31.2	47
Pigeonhole sort	428	429	431	428.4	437	438	437	438	453	469
Counting sort	480	485	484	486	485	484	500	515	520	562

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	0.2	0.2	0.2	0.3	0.7	0.8	1.8	3.4
Merge sort	0	0	0	0	1.6	6.4	8	9.6	12.8	16
Pigeonhole sort	349	350	352	337.6	330.6	328	340	345	358	360
Counting sort	266	266	265	266	265	281	281	282	300	310

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

Algorithm	Input Size									
	512	1024	2048	4096	8192	16384	32768	65536	131072	251281
Insertion sort	0	7.5	7.7	9.4	24.8	65.5	254	970.6	4240	16260.4
Merge sort	0	0	1.6	2.5	6.4	8	9.4	12.8	16	31
Pigeonhole sort	256	266	268	264	265	281	281	281	453	460
Counting sort	300	297	281	296	281	297	280	281	350	455

For each of the 3 experiments I get expected results. For insertion sort the best case input is an array that is already sorted. In this case insertion sort has a linear running time O(n). Worst is reverse sorted data The simplest worst case input is an array sorted in reverse order. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time $O(n^2)$. The average case is also quadratic which makes insertion sort impractical for sorting large arrays.

For other sorting algorithms, data sizes do not matter. Merge sort works O(nlogn) in every case, pigeonhole works O(n + N) and counting sort works O(n+k).

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

Table 5: Auxiliary space complexity of the given algorithms.

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

Plot1.

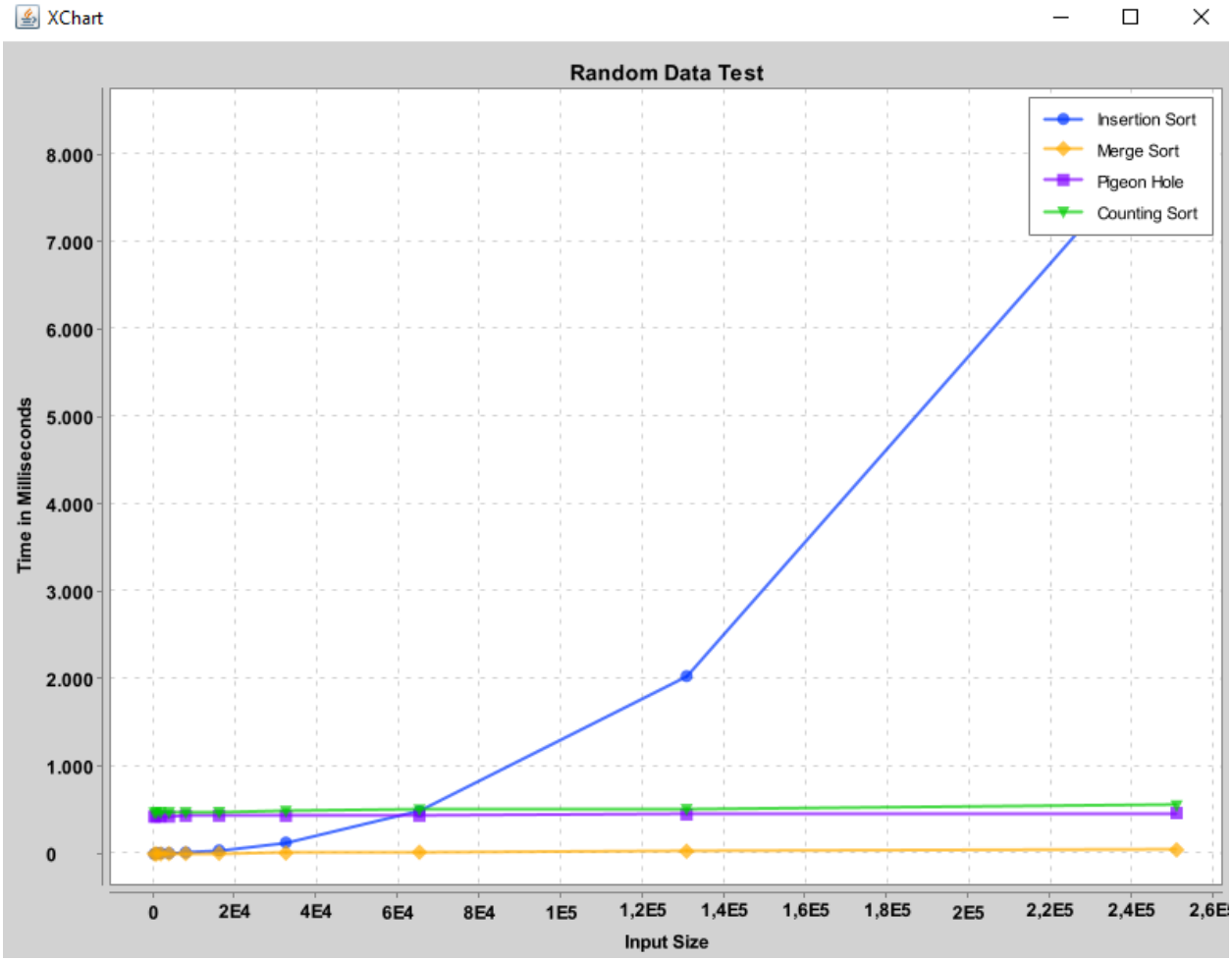

Figure 1: Plot of the random data sorting algorithms.

As shown in the plot in random data case, merge , pigeonhole and counting sort runs linear but insertion sort runs quadratic. As the size of the data increases, the insertion sort takes much more time than the others.

Plot2.

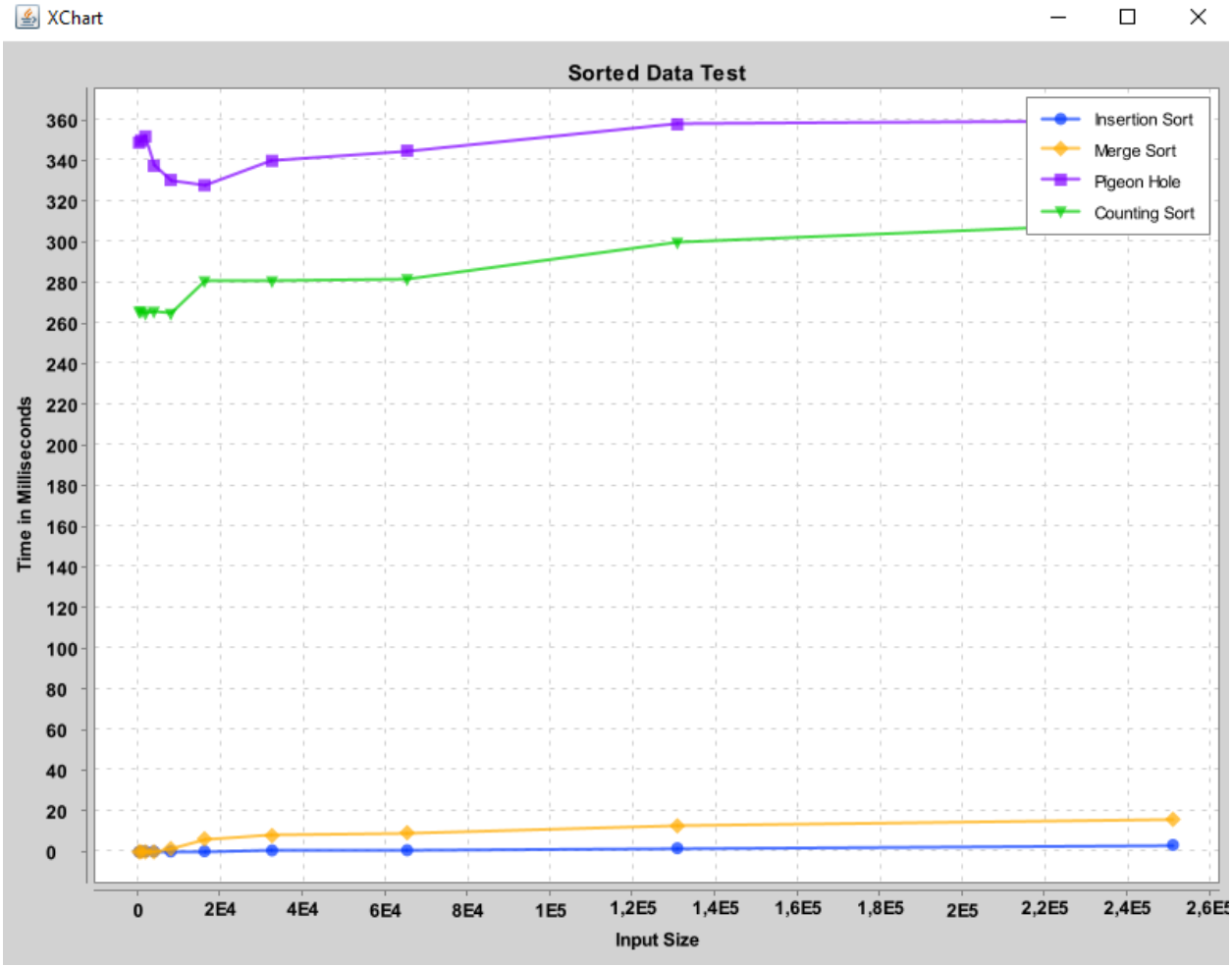

Figure 2: Plot of the sorted data sorting algorithms.

In sorted data case, Insertion sort runs in O(n) and merge sort runs in O(nlogn). They both take less time than the pigeonhole and counting because pigeonhole and counting sorts uses extra memory and creating and filling (or just leaving it empty) this extra memory costs more time.

Plot3.

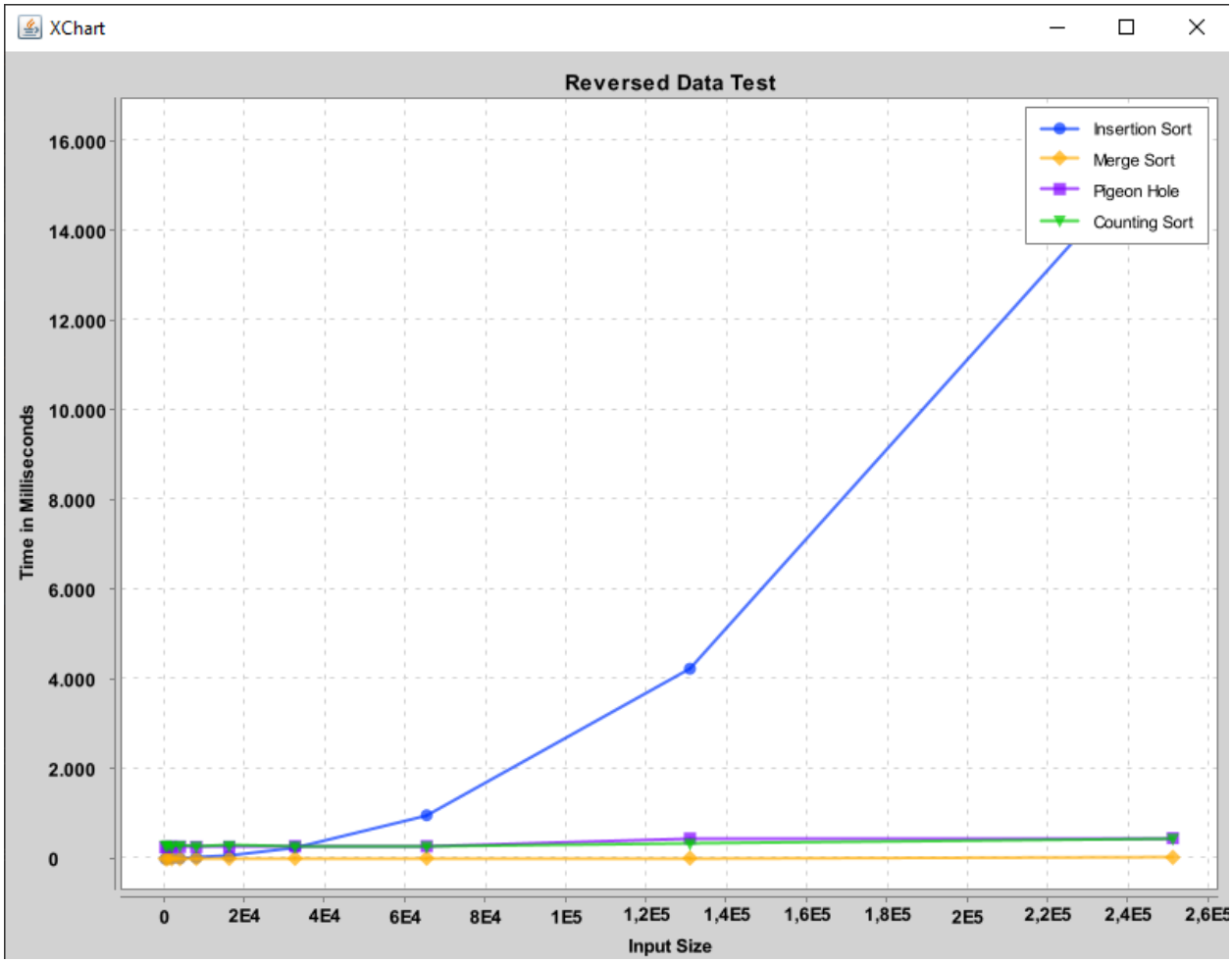

Figure 3: Plot of the reversed data sorting algorithms.

Reversed data case is worst case for insertion sort it works in $O(n^2)$. It's going very badly, especially with the array size increasing. But for small size of data insertion sort works better than the other algorithms.

4 Notes

As a conclude if we compare 4 sorting algorithm:

We can use insertion sort especially when we need to be quick to sort small size of datas or if we have limited memory.

We can use merge sort when we want stable time for sorting any size of data and we don't have memory problems.

We can use pigeonhole sort when we want sort big size of data and number of data (n) and the length of the range of possible key values (N) are approximately the same and we don't have memory problems.

We can use counting sort when we want sort big size of data and number of data (n) and the max value of the data (k) are approximately the same and we don't have memory problems.

References

- $https://en.wikipedia.org/wiki/Insertion_sort$
- $https://en.wikipedia.org/wiki/Merge_sort$
- $https://www.geeksforgeeks.org/merge-sort/$
- $https://my.eng.utah.edu/\ lakhani/Assignment5/assignment5.html$
- $https://en.wikipedia.org/wiki/Pigeonhole_sort$
- $https://en.wikipedia.org/wiki/Counting_sort$
- $https://www.geeksforgeeks.org/counting-sort/$
- $https://www.geeksforgeeks.org/pigeonhole-sort/$