Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2022 Spring

Programming Assignment 1

March 9, 2022

Student name:
Diren Boran Sezen

Student Number:
b21946553

1 Problem Definition

In this assignment we are going to measure and record 4 different algorithms' time and memory usage. We are going to run these algorithms with different input sizes over and over until we see an average result of how fast or how slow that algorithm is working. What we actually want to emphasise here is that, there are different computers in the world that has different powers. But a good, efficient algorithm will always be more accurate choice than a powerful machine. So we don't have to build new super computers every time as the problems go bigger and bigger. Instead, we analyze the algorithms (like the ones we are going to analyze in this experiment) and experiment on them, understand how they behave in different input sizes. Are they fast or what are their complexities? What can be their upper and lower bound? How much additional memory are we using? We point these questions, we repeat the experiments and as result we understand the code better. We solve the problems with better understanding and we don't have to live a couple more lifetimes to see a problem to be solved.

2 Solution Implementation

2.1 Insertion Sort

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

2.2 Mergesort

```
17 public void mergeSort(int[] arr, int left, int right){
18
19       if(left>=right){
20           return;
21       }else{
22           int mid= left + (right-left)/2;
23
24           mergeSort(arr, left, mid);
25           mergeSort(arr, mid + 1, right);
26           merge(arr, left, mid, right);
27
28       }
29
30    }
31
32    public void merge(int[] arr, int left, int mid, int right){
33
34        int[] left_half = new int[mid-left+1];
35        int[] right_half = new int[right-mid];
36
37        int l1 = mid-left+1;
38        int l2 = right-mid;
39
40        //data copying
41        for(int i=0; i<l1; i++){
42            left_half[i] = arr[left+i];
43        }
44
45        for(int j=0; j<l2; j++){
46            right_half[j] = arr[mid + 1 + j];
47        }
48
49
50        int i=0, j=0;
51        int a = left;
52
53        while(i<l1 && j<l2){
54
55            if(left_half[i] < right_half[j]){
56                arr[a] = left_half[i];
57                i++;
58            }
59            else{
60                arr[a] = right_half[j];
61                j++;
62            }
```

```
            a++;
        }

        while(i<l1){
            arr[a] = left_half[i];
            i++;
            a++;
        }

        while(j<l2){
            arr[a] = right_half[j];
            j++;
            a++;
        }

    }
```

2.3 Pigeonhole Sort

```
public void pigeonholeSort(int[] arr, int l){
        int min = arr[0], max = arr[0];

        for (int j : arr) {
            if (j > max) {
                max = j;
            }
            if (j < min) {
                min = j;
            }
        }
        int range = max - min +1;
        int index=0;
        int[] holes = new int[range];
        Arrays.fill(holes, 0);
        for (int i=0; i<l; i++) {
            holes[arr[i] - min]++;
            index=0;
        }
        for(int j=0; j<range; j++){
            while(holes[j]-- > 0){
                arr[index] = j + min;
                index++;
            }
        }
    }
```

2.4 Counting Sort

```
106  public void countingSort(int[] arr,int k, int l){
107
108          int[] output = new int[l];
109          int[] count = new int[k+1];
110          Arrays.fill(count, 0);
111
112          for(int i=0; i<l; i++){
113              ++count[arr[i]];
114          }
115
116
117          for(int i=1; i<=k; i++){
118              count[i] += count[i-1];
119          }
120
121          for(int i=l-1; i>=0; i--){
122              output[count[arr[i]] - 1] = arr[i];
123              --count[arr[i]];
124          }
125
126          for (int i = 0; i < l; ++i)
127              arr[i] = output[i];
128      }
```

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	118140	69280	45910	153560	604450	2617600	9632580	37461720	166025890	787743650
Merge sort	137110	172370	371490	287490	639470	1483610	2465000	4810110	9280750	23370640
Pigeonhole sort	254340000	192017600	132596660	124471920	134194180	124208490	128439800	126936680	133514510	128105630
Counting sort	386648270	332129950	337775740	343633420	332217030	352754210	364285010	336711390	295267900	303621250

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	15840	5230	6610	29380	10350	20590	41180	78740	173230	342590
Merge sort	74390	159370	143150	255050	527490	1581610	1826470	4871570	8387020	17312010
Pigeonhole sort	255156650	130134350	120204030	118313400	130657950	124686470	120335860	128930700	124885950	135850890
Counting sort	326632220	338769530	336657080	329261090	348937340	348937340	332457980	307213370	282502990	293212270

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

Algorithm	Input Size									
	512	1024	2048	4096	8192	16384	32768	65536	131072	251281
Insertion sort	75440	128340	65530	212490	1169760	5045390	19474330	81546950	317537130	1489925020
Merge sort	75950	163710	124890	259530	691310	1191940	1751400	5255970	8590660	17237480
Pigeonhole sort	273589160	125005920	123131280	116737000	125593540	126492910	124780950	124882290	128598100	136797740
Counting sort	326416760	343815380	336106210	324310410	342898190	340339310	350264210	300615240	288253840	304226890

Complexity analysis tables to complete:

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

Table 5: Auxiliary space complexity of the given algorithms.

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