Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2025 Spring

Programming Assignment 1

March 21, 2025

Student name:
Ömer Taha Gögen

Student Number:
b2220356076

1 Problem Definition

This project evaluates the performance of sorting algorithms (**Comb Sort, Insertion Sort, Shaker Sort, Shell Sort, and Radix Sort**) by measuring execution time and memory usage on the **TrafficFlowDataset**. Results are visualized using the **Java XChart** module.

2 Solution Implementation

2.1 1- Comb Sort

Comb Sort is a **comparison-based sorting algorithm** and an optimized version of **Bubble Sort**. It improves performance by using a **gap** between compared elements, which starts as the array length and shrinks by generally 1.3. When the gap==1 , Comb Sort behave like Bubble Sort. This logic is effective on large datasets because the gap moves larger elements faster, reducing swaps and improving efficiency over Bubble Sort.

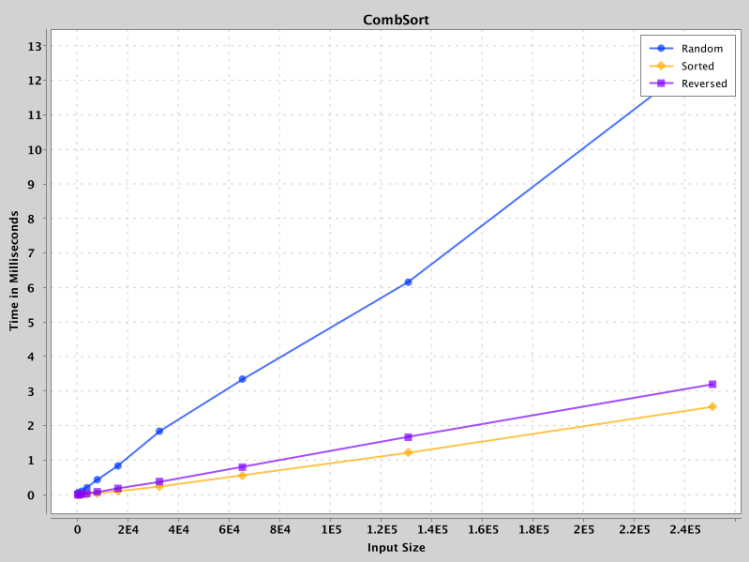

Figure 1: A smaller plot of the CombSort.

Java implementation:

```java
public static void comb(int arr[]){
        int l=arr.length;
        int gap=l;
        double shrink=1.3;
        boolean sorted=false;

        while (sorted==false){
            gap= (int ) Math.floor((gap/shrink));
```

```
            if (gap<1){
                gap=1;
            }
            if (gap==1){
                sorted=true;
            }
            for (int i=0;i<l-gap;i++){
                if (arr[i]>arr[i+gap]){
                    int temp=arr[i];
                    arr[i]=arr[i+gap];
                    arr[i+gap]=temp;
                    sorted=false;
                }
            }
        }
    }
```

2.2 2-Insertion Sort

Insertion Sort is a simple, stable, adaptive **comparison-based sorting algorithm**. Insertion Sort works by considering the left part of the array as sorted and inserting each new element into its correct position within this sorted section, shifting elements as needed.

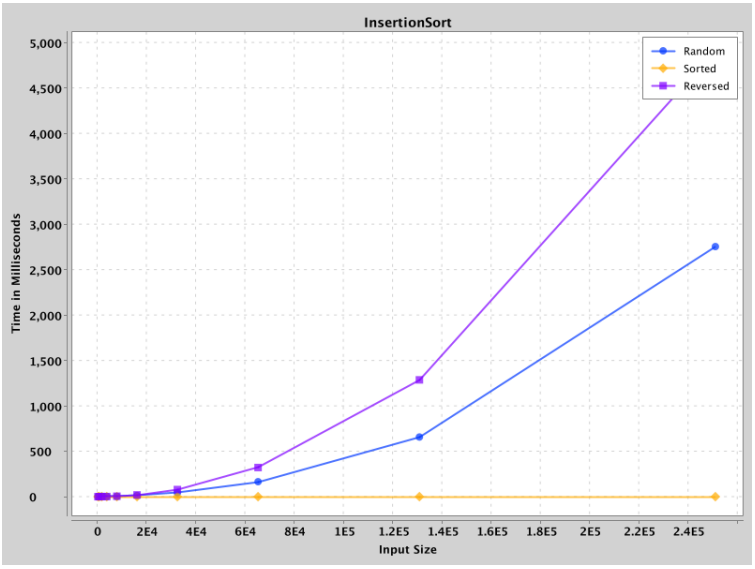

Figure 2: A smaller plot of the InsertionSort.

Java implementation:

```
25 public static void insertion(int arr[]){
26         int l=arr.length;
27         for (int i=1;i<l;i++){
28             int key=arr[i];
29             int j=i-1;
30             while (j>=0 && arr[j]>key){
31                 arr[j+1]=arr[j];
32                 j--;
33             }
34             arr[j+1]=key;
35         }
36     }
```

2.3 3-Shaker Sort

Shaker Sort is a bidirectional variant of the Bubble Sort algorithm. It improves efficiency by scanning the dataset in both forward and backward directions, moving elements more effectively and reducing the number of passes needed to sort the list.

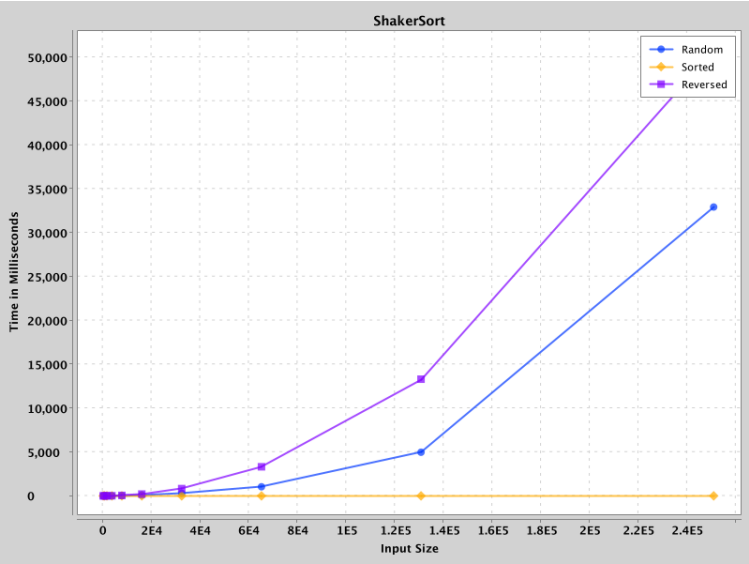

Figure 3: A smaller plot of the ShakerSort.

Java implementation:

```
37 public static void ShakerSort(int[] arr) {
38         int l=arr.length;
39         boolean swap= true;
40 
41         while (swap==true) {
42             swap = false;
43 
44             for (int i = 0; i < l-1; i++) {
45                 if (arr[i] > arr[i + 1]) {
46                     int temp = arr[i];
47                     arr[i] = arr[i + 1];
48                     arr[i + 1] = temp;
49                     swap = true;
50                 }
51             }
52 
53             if (swap==false){
54                 break;
55             }
56 
57             swap = false;
58 
59             for (int i = l-2; i >= 0; i--) {
60                 if (arr[i] > arr[i + 1]) {
61                     int temp = arr[i];
62                     arr[i] = arr[i + 1];
63                     arr[i + 1] = temp;
64                     swap = true;
65 
66                 }
67             }
68         }
69     }
```

2.4 4-Shell Sort

Shell Sort is a **comparison-based sorting algorithm** and an improved version of **Insertion Sort**. In this project, as we have observed, Shell Sort performs significantly better than Insertion Sort for large datasets. This is because **Insertion Sort only swaps adjacent elements**, while Shell Sort moves elements much further in early iterations, reducing the total number of swaps.

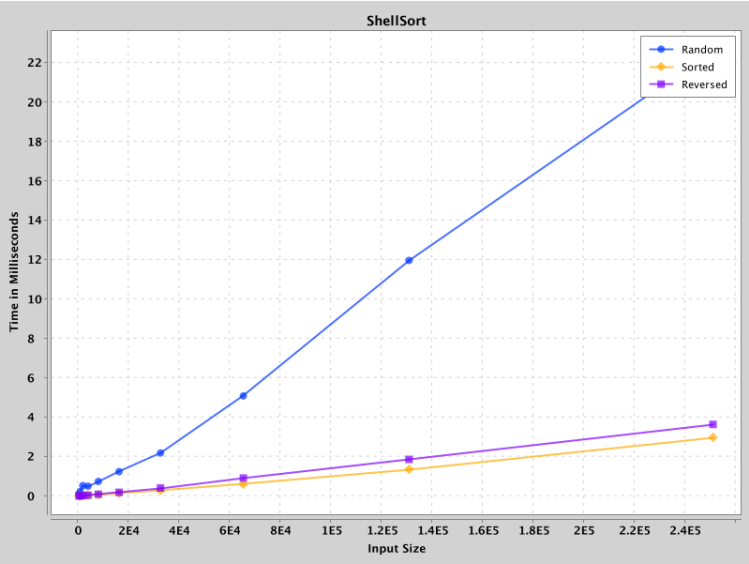

Figure 4: A smaller plot of the ShellSort.

Java implementation:

```java
public static void shellSort(int arr[]){
        int l=arr.length;
        int gap=l/2;
        while (gap>0){
            for (int i=gap;i<l;i++){
                int temp=arr[i];
                int j=i;
                while ( j>=gap && arr[j-gap]>temp){
                    arr[j]=arr[j-gap];
                    j-=gap;
                }
                arr[j]=temp;
            }
            gap=gap/2;
        }
    }
```

2.5 5-Radix Sort

Radix Sort is a **non-comparison-based algorithm** that sorts numbers by processing each digit separately. Instead of directly comparing values, it groups and orders them based on individual digits, making it particularly efficient for **large numerical datasets**.

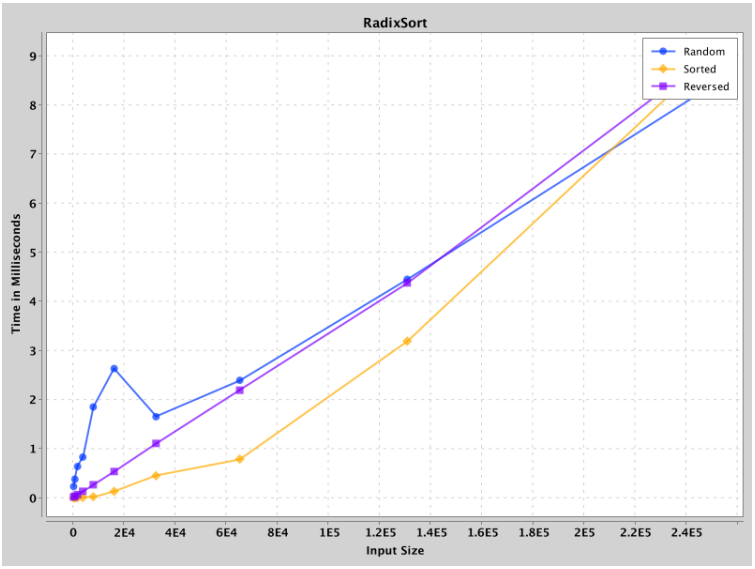

Figure 5: A smaller plot of the RadixSort.

Java implementation:

```java
public static int get_digit(int n, int index) {
        if (index < 0){
            return 0;
        }
        int divisor=1;
        for (int i=0;i<index;i++){
            divisor*=10;
        }
        return (n / divisor) % 10;
    }

    public static void countingSort(int arr[],int pos){
        int l=arr.length;
        int count[] =new int[10];
        int output[]=new int[l];

        for (int i=0;i<l;i++){
            int digit=get_digit(arr[i],pos);
            count[digit]++;
        }
```

```
106         for (int i=1;i<10;i++){
107             count[i]=count[i]+count[i-1];
108         }
109         for (int i=l-1;i>=0;i--){
110             int digit=get_digit(arr[i],pos);
111                 count[digit]--;
112                 output[count[digit]]=arr[i];
113         }
114         System.arraycopy(output, 0, arr, 0, l);
115     }
116 
117     public static void radixSort(int arr[]){
118         int maxi=0;
119         for (int number : arr){
120             maxi=Math.max(maxi,number);
121         }
122         int d = (int)Math.log10(maxi) + 1;
123         for (int i=0;i<d;i++){
124             countingSort(arr,i);
125         }
126     }
```

3 Results, Analysis, Discussion

Running time test results for sorting algorithms are given in Table 1.

Table 1: Results of the running time tests performed for varying input sizes (in ms).

	Input Size n									
Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
	Random Input Data Timing Results in ms									
Comb sort	0.02	0.03	0.07	0.15	0.32	0.62	1.33	2.83	6.03	12.37
Insertion sort	0.06	0.11	0.28	0.92	3.20	11.30	41.49	162.74	671.96	2824.81
Shaker sort	0.18	0.40	1.13	4.35	17.69	76.83	280.11	1045.22	4990.93	33241.35
Shell sort	0.03	0.07	0.14	0.24	0.51	1.05	2.32	5.20	11.63	21.55
Radix sort	0.04	0.07	0.12	0.19	0.41	0.69	1.03	2.03	3.95	8.01
	Sorted Input Data Timing Results in ms									
Comb sort	0.01	0.01	0.01	0.02	0.06	0.12	0.26	0.57	1.25	2.52
Insertion sort	0.00	0.00	0.00	0.01	0.01	0.02	0.04	0.03	0.05	0.08
Shaker sort	0.00	0.00	0.00	0.00	0.01	0.01	0.03	0.02	0.04	0.07
Shell sort	0.01	0.01	0.02	0.05	0.04	0.07	0.14	0.23	0.31	0.65
Radix sort	0.01	0.01	0.02	0.03	0.06	0.28	0.63	1.30	3.63	10.17
	Reversely Sorted Input Data Timing Results in ms									
Comb sort	0.01	0.01	0.02	0.04	0.09	0.19	0.37	0.77	1.59	3.04
Insertion sort	0.10	0.17	0.40	1.57	5.14	20.34	80.91	327.82	1312.21	4985.82
Shaker sort	0.33	0.92	3.24	12.84	51.65	208.88	829.28	3320.12	13346.64	50959.88
Shell sort	0.02	0.02	0.05	0.08	0.11	0.24	0.46	0.97	2.04	4.00
Radix sort	0.04	0.07	0.12	0.20	0.41	0.72	1.04	2.02	3.91	8.45

Complexity analysis tables to complete:

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Comb sort	$\Omega(n \log n)$	$\Theta\left(\frac{n^2}{2^p}\right)$	$O(n^2)$
Insertion sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Shaker sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Shell sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$
Radix sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$

k= Maximum digit number p=Number of increments

Table 2: Computational complexity comparison of the given algorithms.

Table 3: Auxiliary space complexity of the given algorithms.

Algorithm	**Auxiliary Space Complexity**
Comb sort	$O(1)$
Insertion sort	$O(1)$
Shaker sort	$O(1)$
Shell sort	$O(1)$
Radix sort	$O(n+k)$

Radix Sort is generally considered to have linear time complexity (O(nk)). However, unlike comparison-based sorting algorithms, it requires additional memory, which can be a drawback in certain scenarios.

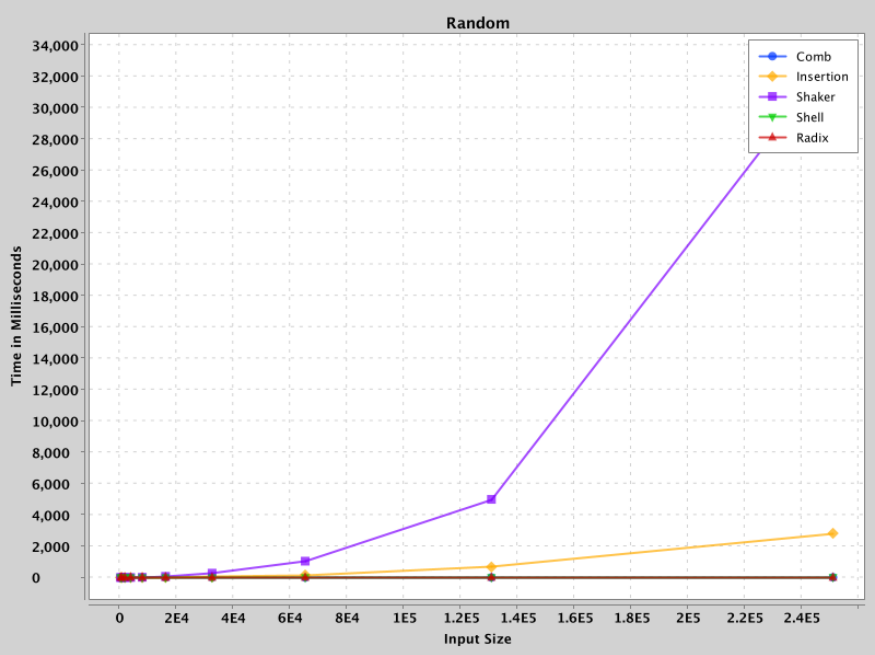

Figure 6: All algorithm for Random data.

• **Shaker Sort (Purple line)** has the worst performance. **Its runtime increases exponentially with input size**, confirming its **O(n²) complexity**.

• **Insertion Sort (Yellow line)** shows moderate performance. Since it has **O(n²) complexity**, it slows down significantly for larger inputs.

• **Shell Sort (Green line)** and **Comb Sort (Blue line)** perform significantly better than the previous algorithms. This aligns with their **O(n log n) average case complexity**.

• **Radix Sort (Red line)** is the best-performing algorithm. Its growth is **almost linear**, supporting its **O(nk) complexity**.

As a result , Theoretical complexities generally align with the plots that we generated .

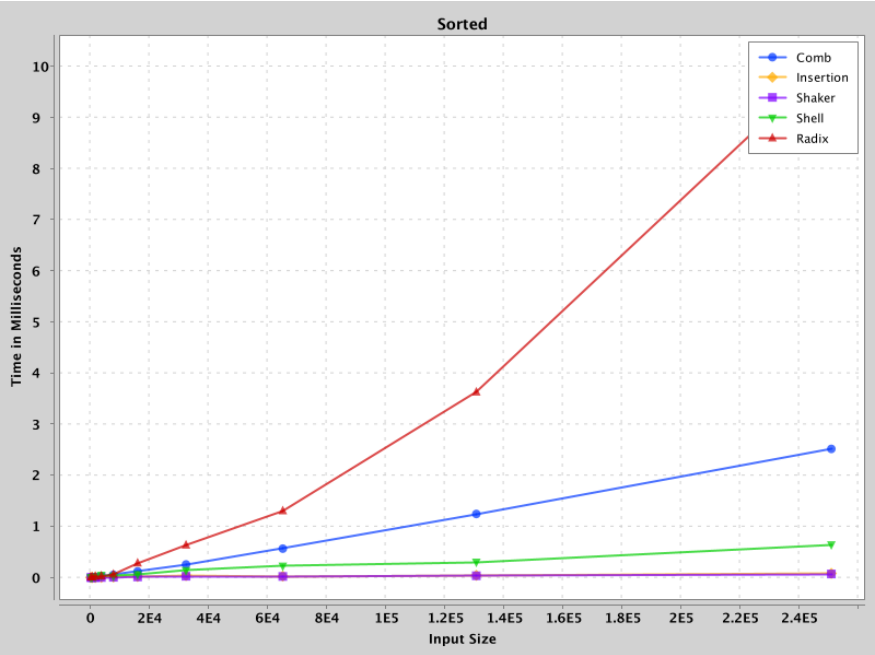

Figure 7: All algorithm for Sorted data.

Comparison-based algorithms (Insertion, Shell, Comb, Shaker Sort) are adaptive, meaning they reduce the number of operations based on input order. For example, **Insertion Sort runs in O(n) on sorted data** since it barely needs to swap elements.

However, **Radix Sort, is not adaptive** and **always processes all digits**, even if the data is already sorted. Since it does not use comparisons, it **cannot take advantage of order** and still performs unnecessary operations. Additionally, **its extra memory usage and fixed number of passes make it slower** than adaptive algorithms on sorted data.

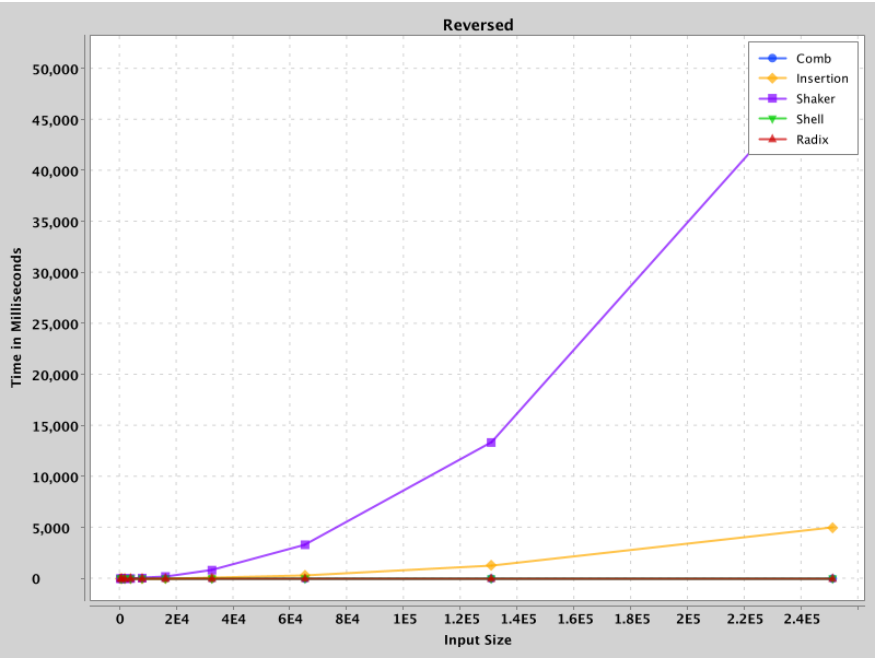

Figure 8: All algorithm for Reversed data.

4 Conclusion

The experiments showed that the theoretical complexities of sorting algorithms largely **matched** their practical running times, though some differences were observed due to factors such as computer architecture, processor optimizations, and memory management. Radix Sort was very fast for sorting large numbers because it does not compare values directly, but it was slower on already sorted data since it still processed every digit **unnecessarily**. Comb Sort **improved efficiency compared to** Bubble Sort by reducing the number of swaps through a shrinking gap, making it faster in most cases. Shell Sort **outperformed** Insertion Sort for large datasets by moving elements more efficiently, while Insertion Sort worked **exceptionally well** on nearly sorted data **as** it required very few operations. Shaker Sort, which scans both forward and backward, **showed a slight improvement over** Bubble Sort, especially on sorted data, but **remained inefficient** for large or completely unsorted datasets. These results **highlight** that while theoretical analysis **provides a useful guideline for**choosing a sorting algorithm, real-world performance depends on factors such as dataset size, initial order, and system characteristics. Therefore, selecting the best sorting algorithm requires considering both theoretical complexity and practical performance in different scenarios.

References

- https://www.geeksforgeeks.org/comb-sort/
- https://en.wikipedia.org/wiki/Comb_sort
- https://www.geeksforgeeks.org/insertion-sort-algorithm//
- https://www.geeksforgeeks.org/shell-sort/
- https://en.wikipedia.org/wiki/Shellsort#Gap_sequences
- https://www.geeksforgeeks.org/radix-sort/