Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2025 Spring

Programming Assignment 1

March 15, 2025

Student name:
Yusuf Demiray

Student Number:
b2220356016

1 Problem Definition

The main objective of this assignment is to analyze the relationship between the running times of sorting algorithms and their theoretical asymptotic complexities. To achieve this, we will implement the given 5 sorting algorithms in Java and test them on *TrafficFlowDataset.csv*. In order to examine the consistency of empirical results with theoretical growth functions, we will run each test 10 times, calculating the average execution time.

In this assignment, we will focus on the following questions:

- How do different sorting algorithms perform as the input size increases?
- How do these algorithms behave on random, sorted, and reversely sorted input data?
- To what extent do the empirical results align with theoretical complexities?

By analyzing the results through graphical representations, we show the asymptotic behavior of the algorithms.

2 Solution Implementation

In this section, we will implement the five sorting algorithms provided in pseudocode using the Java programming language. These sorting algorithms, in order, are **Comb Sort, Insertion Sort, Shaker Sort, Shell Sort, and Radix Sort**. The first four algorithms are **comparison-based sorting techniques**, whereas **Radix Sort is a non-comparison-based algorithm**.

2.1 Comb Sort Algorithm

Comb sort algorithm java code :

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

        while (sorted == false) {
            gap = Math.max(1, (int) Math.floor(gap / shrink));
            if(gap == 1){
                sorted = true;
            }
            else{
                sorted = false;
            }

            for (int i = 0; i < arr.length - 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 Insertion Sort Algorithm

Insertion sort algorithm java code :

```
public class InsertionSort {
    public static void insertionSort(int[] arr) {
        int len = arr.length;
        int i ;
        for (int j = 1; j < len; j++) {
            int key = arr[j];
            i = j - 1;

            while (i >= 0 && arr[i] > key) {
                arr[i + 1] = arr[i];
                i--;
            }
            arr[i + 1] = key;
        }
    }
}
```

2.3 Shaker Sort Algoritm

Shaker sort algorithm java code :

```
public class ShakerSort {
    public static void shakerSort(int[] arr) {
        boolean swapped = true;
        while (swapped == true) {
            swapped = false;
            for (int i = 0; i <= arr.length - 2; i++) {
                if (arr[i] > arr[i + 1]) {
                    int sw = arr[i];
                    arr[i] = arr[i + 1];
                    arr[i + 1] = sw;
                    swapped = true;
```

```
                }
            }
            if (swapped == false){
                break;
            }
            swapped = false;
            for (int i = arr.length - 2; i >= 0; i--) {
                if (arr[i] > arr[i + 1]) {
                    int sw = arr[i];
                    arr[i] = arr[i + 1];
                    arr[i + 1] = sw;
                    swapped = true;
                }
            }
        }
    }
}
```

2.4 Shell Sort Algorithm

Shell sort algorithm java code :

```
public class ShellSort {
    public static void shellSort(int[] arr) {
        int len = arr.length;
        int gap = len / 2;

        while (gap > 0) {
            for (int i = gap; i <= len - 1 ; i++) {
                int temp = arr[i];
                int j = i;

                while (j >= gap && arr[j - gap] > temp) {
                    arr[j] = arr[j - gap];
                    j = j - gap;
                }

                arr[j] = temp;
            }
            gap = gap / 2;
        }
    }

}
```

2.5 Radix Sort Algorithm

Radix sort algorithm java code :

Note : I am only adding the implementation of the pseudo code given in the assignment to the report. I did not add the auxiliary function (getDigit) for the report layout.

```
95  public class RadixSort {
96
97      public static int[] radixSort(int[] arr, int digit) {
98          int pos;
99          for (pos = 1; pos <= digit; pos++) {
100             arr = countingSort(arr, pos);
101         }
102         return arr;
103     }
104
105
106     private static int[] countingSort(int[] arr, int pos) {
107         int[] count = new int[10];
108         for (int i = 0; i < 10; i++) {
109             count[i] = 0;
110         }
111
112         int size = arr.length;
113         int[] output = new int[arr.length];
114
115         for (int i = 1; i <= size; i++) {
116             int digit = getDigit(arr[i - 1], pos);
117             count[digit]++;
118         }
119
120         for (int i = 1; i < 10; i++) {
121             count[i] += count[i - 1];
122         }
123
124         for (int i = size; i >= 1; i--) {
125             int digit = getDigit(arr[i - 1], pos);
126             count[digit]--;
127             output[count[digit]] = arr[i - 1];
128         }
129
130         return output;
131     }
132 }
```

3 Results, Analysis, Discussion

In this section, we will analyze the performance of the implemented sorting algorithms.

The measurements will be conducted for three different data distributions:

- Randomly ordered data
- Sorted data
- Reverse-sorted data

Each algorithm will be executed **10 times** for each input size, and the average execution times will be recorded. Through this analysis, we will observe the **best-case, average-case, and worst-case scenarios** of each sorting algorithm.

In addition to the experimental analysis of the sorting algorithms, we will also examine their **theoretical computational complexity** and **auxiliary space complexity**. To achieve this, we will complete the given two tables.

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	0.177	0.063	0.089	0.081	0.258	0.569	1.335	3.215	6.311	12.212
Insertion sort	0.230	0.188	0.383	0.785	2.845	11.600	42.226	167.353	711.175	2927.693
Shaker sort	0.464	0.627	1.345	5.341	19.634	87.011	314.670	1188.367	4773.599	35433.691
Shell sort	0.197	0.053	0.129	0.271	0.731	1.559	4.530	10.424	23.182	23.339
Radix sort	0.286	0.143	0.321	0.186	0.330	0.609	1.428	2.497	4.912	9.699
	Sorted Input Data Timing Results in ms									
Comb sort	0.002	0.005	0.012	0.031	0.063	0.146	0.298	0.624	1.337	2.948
Insertion sort	4.414	7.082	0.001	0.002	0.005	0.0011	0.021	0.036	0.086	0.183
Shaker sort	2.625	3.540	7.790	0.001	0.002	0.004	0.009	0.017	0.035	0.207
Shell sort	0.003	0.007	0.017	0.037	0.077	0.184	0.368	0.750	1.592	3.180
Radix sort	0.002	0.003	0.005	0.014	0.027	0.141	0.399	0.788	0.081	9.737
	Reversely Sorted Input Data Timing Results in ms									
Comb sort	0.003	0.008	0.018	0.037	0.075	0.159	0.332	0.734	1.616	3.138
Insertion sort	0.024	0.099	0.374	1.372	5.257	20.845	83.164	354.025	1424.738	5221.751
Shaker sort	0.249	1.073	3.735	14.193	56.465	224.856	898.628	3605.380	14389.001	42040.530
Shell sort	0.005	0.012	0.022	0.044	0.096	0.218	0.470	1.016	2.170	4.165
Radix sort	0.021	0.044	0.076	0.150	0.308	0.637	1.253	2.445	4.969	9.659

3.1 Analysis of Table 1: Running Time Comparison

- **Random Input Data:** Radix Sort is the fastest for large inputs, while Shaker Sort performs the worst. Shell Sort outperforms comparison-based algorithms.
- **Sorted Input Data:** Comb Sort, Shell Sort, and Radix Sort perform optimally for small inputs. For large inputs Insertion sort or Shaker sort is the best option.

- **Reversely Sorted Input Data:** Shaker and Insertion Sort degrade significantly, while Comb , Shell and Radix Sort maintain efficiency.

Briefly Radix Sort is the most efficient for large datasets. Shell Sort performs well in structured data, while Insertion and Shaker Sort should be avoided for large reverse-ordered datasets.

3.2 Computational Time Complexity and Auxiliary Space Complexity

In this section, we analyze the computational complexity and auxiliary space complexity of the sorting algorithms used in our study. The focus is on additional memory usage during execution rather than overall space complexity. Below, we provide the theoretical complexity values and justify them based on the pseudo-codes.

Table 2: Computational complexity comparison of the given algorithms.

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Comb sort	$\Omega(nlogn)$	$\Theta(n^2)$	$O(n^2)$
Insertion sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Shaker sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Shell sort	$\Omega(nlogn)$	$\Theta(n \log^2 n)$	$O(n \log^2 n)$
Radix sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$

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	$O(n+k)$

- **Comb Sort:**
  - **Best Case:** When the input is nearly sorted, the gap reduction allows efficient sorting, resulting in $\Omega(n \log n)$.
    Pseudo-code reference: Lines 5-6 (gap reduction).
  - **Average Case:** Due to its similarity to Bubble Sort, it performs $\Theta(n^2)$ swaps.
    Pseudo-code reference: Lines 8-13 (nested loop with swaps).
  - **Worst Case:** When the array is in reverse order, the sorting process requires $O(n^2)$ comparisons.
    Pseudo-code reference: Lines 8-13 (maximum swaps needed).
- **Insertion Sort:**

  - **Best Case:** If the array is already sorted, each element is placed correctly in $O(n)$.
    Pseudo-code reference: Lines 5-6 (while loop exits immediately).
  - **Average Case:** Each element may shift multiple times, leading to $\Theta(n^2)$.
    Pseudo-code reference: Lines 5-9 (nested shifting).
  - **Worst Case:** When the array is in reverse order, all elements must be shifted, resulting in $O(n^2)$.
    Pseudo-code reference: Lines 5-9 (maximum shifting).

- **Shaker Sort:**
  - **Best Case:** If the array is already sorted, only one pass is required, leading to $\Omega(n)$.
    Pseudo-code reference: Lines 4-5, 11-12 (break condition when no swaps occur).
  - **Average Case:** Similar to Bubble Sort, multiple passes are required, leading to $\Theta(n^2)$.
    Pseudo-code reference: Lines 5-10, 15-20 (forward and backward passes).
  - **Worst Case:** In a reversed array, the algorithm requires $O(n^2)$ swaps.
    Pseudo-code reference: Lines 5-10, 15-20 (maximum swaps).

- **Shell Sort:**
  - **Best Case:** An optimal gap sequence allows sorting in $\Omega(n \log n)$.
    Pseudo-code reference: Lines 4-5, 8-11 (gap reduction).
  - **Average Case:** Due to decreasing gaps, it operates in $\Theta(n \log^2 n)$.
    Pseudo-code reference: Lines 4-5, 8-11 (nested loops with decreasing gaps).
  - **Worst Case:** A non-optimal gap sequence results in $O(n \log^2 n)$.
    Pseudo-code reference: Lines 4-5, 8-11 (inefficient gap reduction).

- **Radix Sort:**
  - **Best, Average, and Worst Case:** Since it depends on the number of digits, it runs in $\Theta(nk)$.
    Pseudo-code reference: Lines 2-4, 8-22 (COUNTINGSORT execution per digit).

3.3 Auxiliary Space Complexity Analysis

- **Comb Sort:** Uses only integer variables such as ‘gap‘, ‘sorted‘, and ‘shrink‘.
  Pseudo-code reference: Lines 2-4.

- **Insertion Sort:** Uses only a single variable ‘key‘, making it an in-place sorting algorithm.
  Pseudo-code reference: Lines 3-4.

- **Shaker Sort:** Uses a single boolean variable ‘swapped‘ to track swaps.
  Pseudo-code reference: Line 2.

- **Shell Sort:** Uses only a temporary variable ‘temp‘ for swapping elements.
  Pseudo-code reference: Lines 6-7.

- **Radix Sort:** Uses additional arrays (‘count[]‘, ‘output[]‘), requiring $O(n+k)$ memory. **Pseudo-code reference:** Lines 8-23.

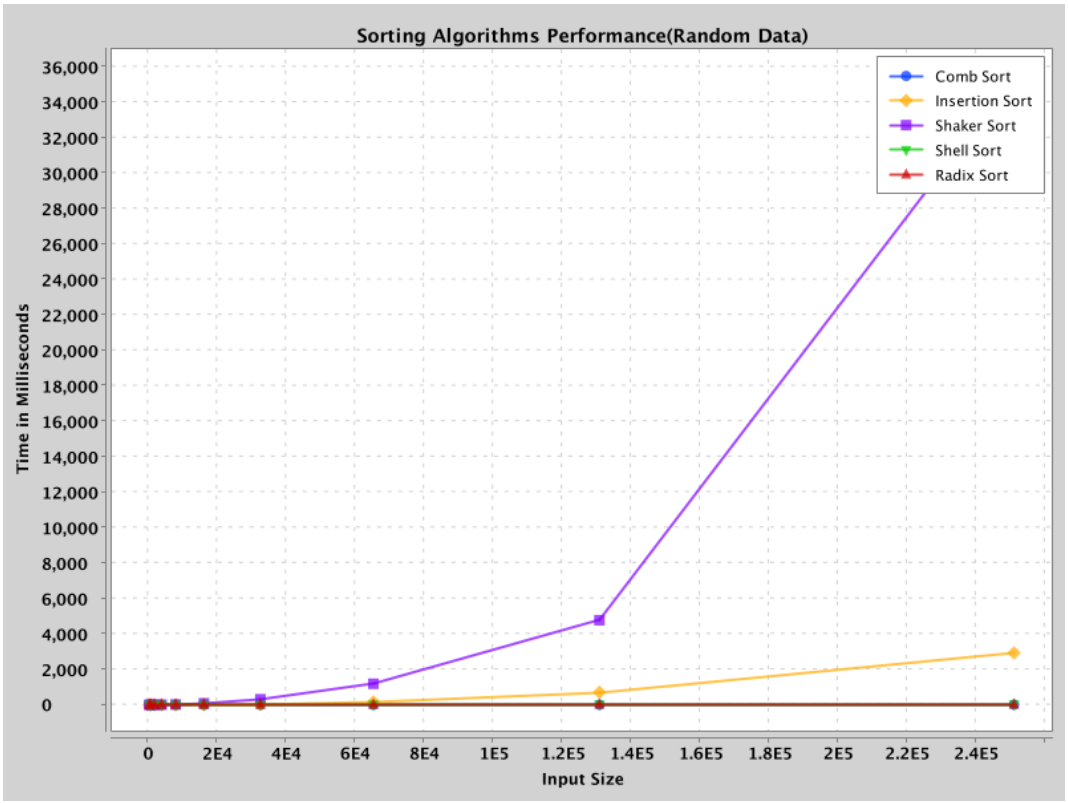
Conclusion:

- The **comparison-based sorting algorithms** (Comb, Insertion, Shaker, Shell Sort) all use **O(1) auxiliary space**, as they operate in-place.
- The **non-comparison-based Radix Sort** requires additional arrays, leading to **O(n + k) space complexity**.

3.4 Plotting the Results

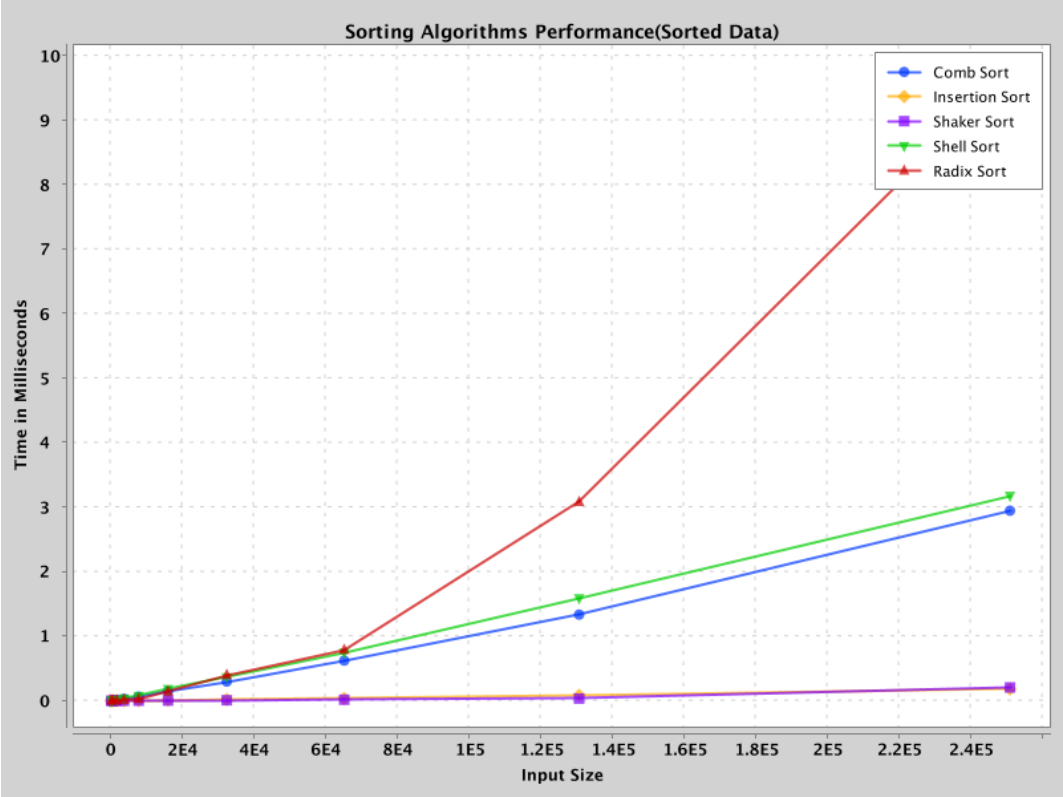
In this section, we present the plots obtained from the experimental results. Each plot compares the performance of the given sorting algorithms under different input distributions. The X-axis represents the input size (n), while the Y-axis represents the execution time in milliseconds.

3.4.1 Performance on Random Input Data



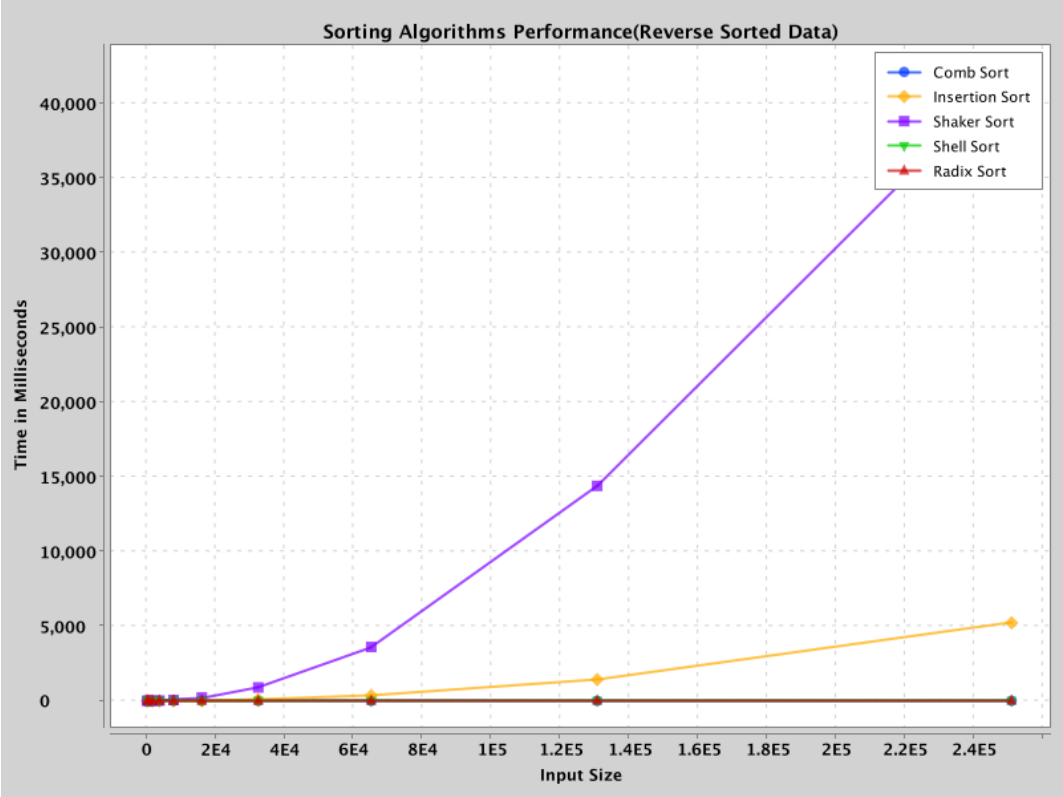
The plot shows the execution time of 5 sorting algorithms on randomly ordered input data. As expected, non-comparison-based algorithms like Radix Sort perform significantly better on large datasets, while Shaker Sort struggles with increased input sizes.

3.4.2 Performance on Sorted Input Data



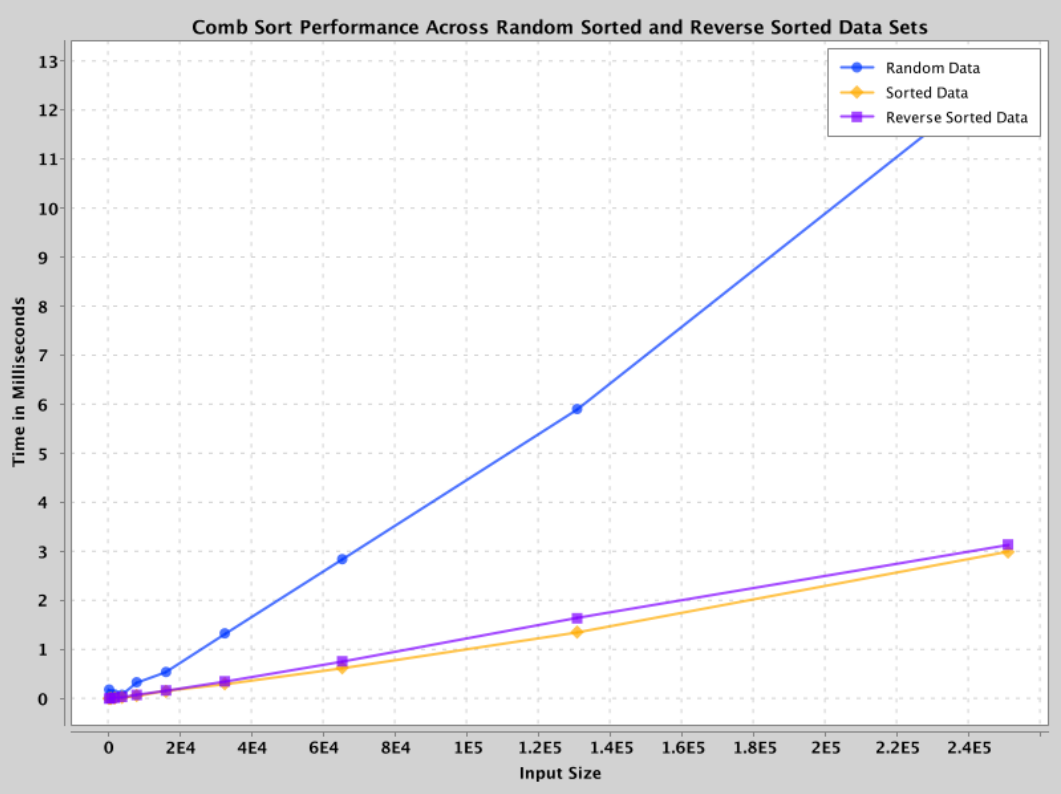
The results confirm that algorithms like Shell Sort and Radix Sort maintain optimal performance, while comparison-based algorithms such as Insertion Sort and Shaker Sort do not always take full advantage of the sorted nature of the input.

3.4.3 Performance on Reversely Sorted Input Data



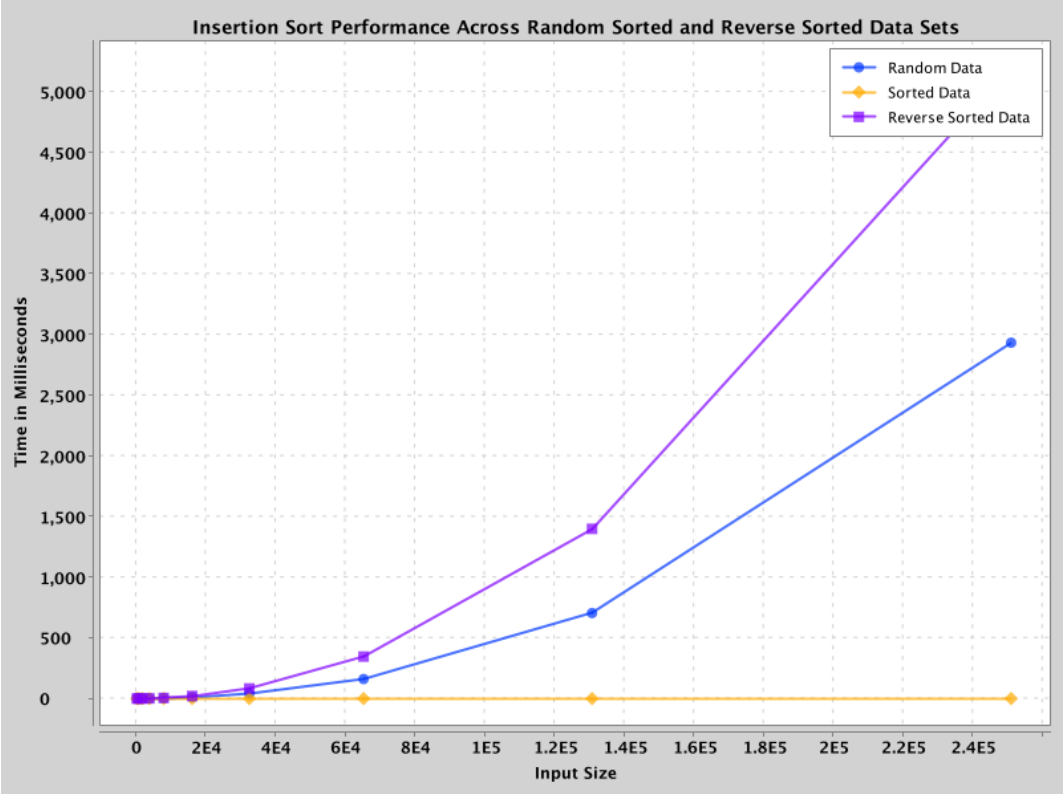
The plot shows the impact of reverse-ordered input on sorting algorithms. As expected, Insertion Sort and Shaker Sort perform poorly due to excessive shifting, while Radix Sort remains efficient.

3.4.4 Comb Sort Performance



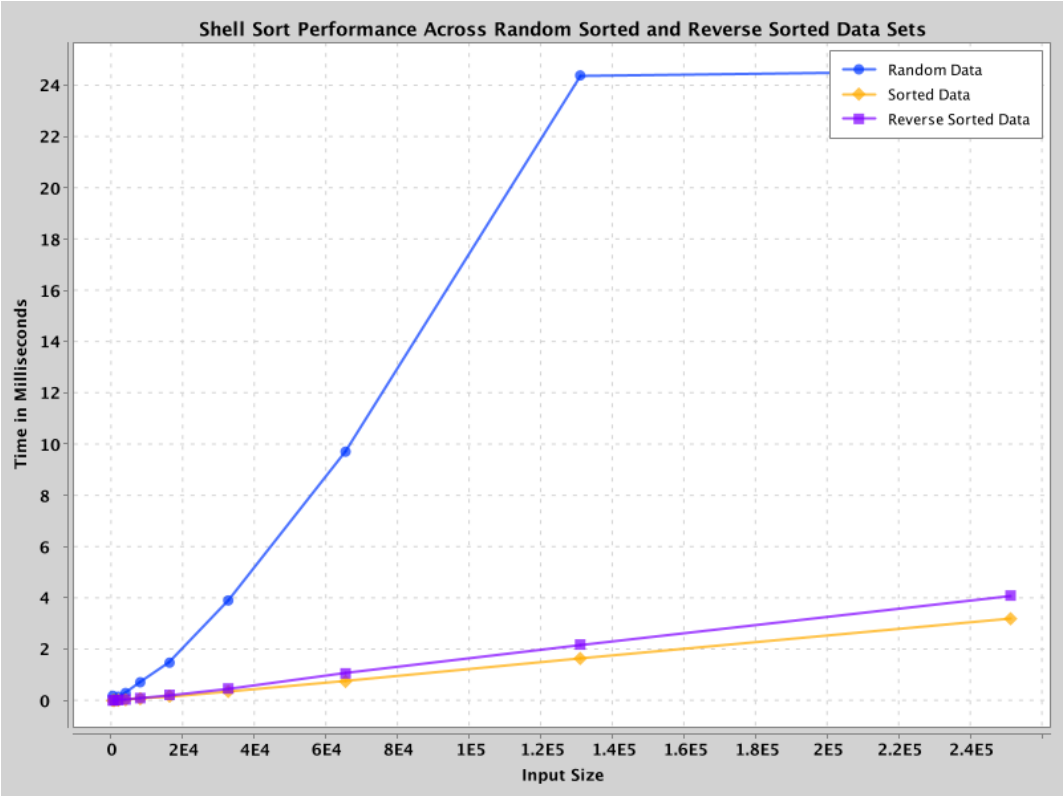
The plot illustrates the performance of Comb Sort across given input data. As seen, the algorithm performs significantly worse on randomly shuffled data, showing an increasing time complexity. However, for already sorted and reverse sorted data, the time complexity remains relatively lower.

3.4.5 Insertion Sort Performance



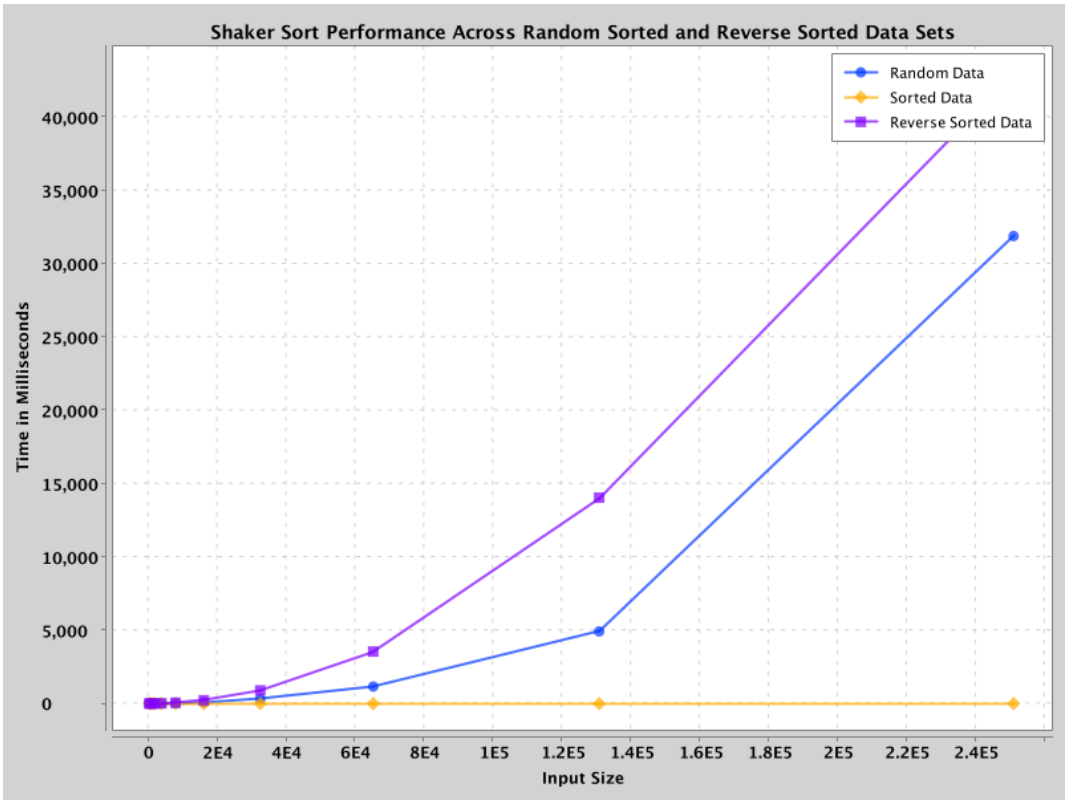
Insertion Sort exhibits its well-known behavior: performing exceptionally well on already sorted data while struggling significantly with reversed and random data.

3.4.6 Shell Sort Performance



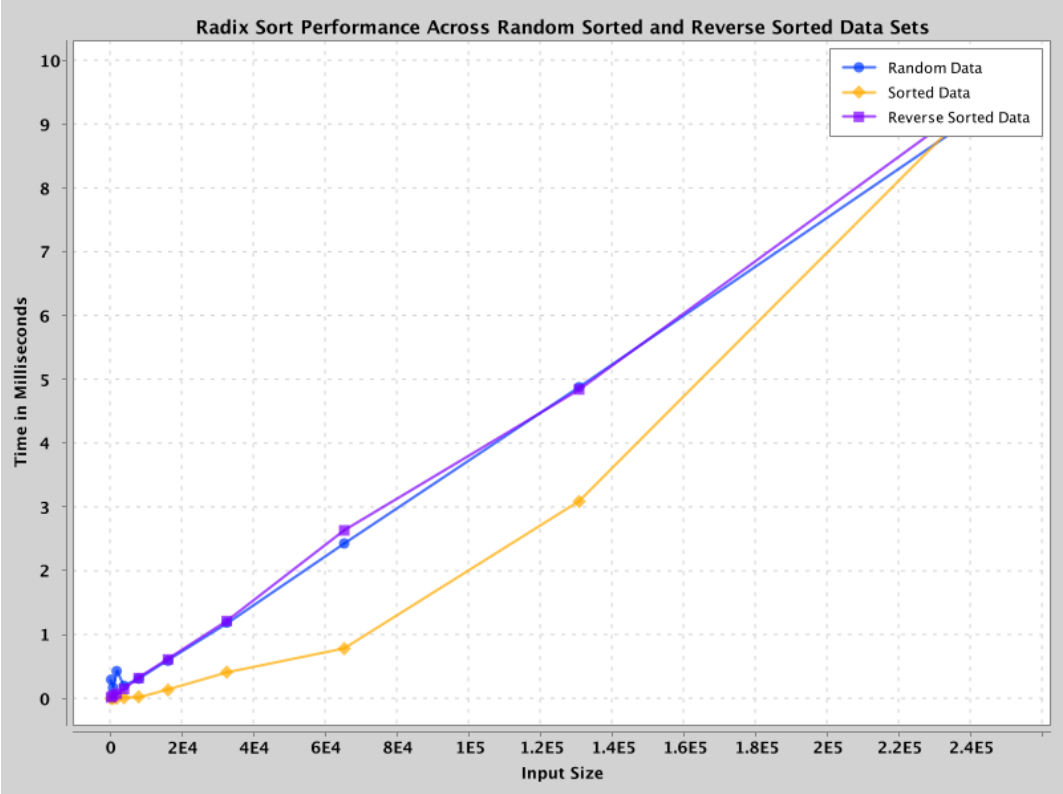
Shell Sort shows a substantial performance boost compared to Insertion Sort.

3.4.7 Shaker Sort Performance



Shaker Sort demonstrates similar performance trends as Insertion Sort but with slightly better results for certain cases. However, its inefficiency is evident when handling reverse-sorted data, where it exhibits extreme slowdowns due to extensive swapping operations.

3.4.8 Radix Sort Performance



Radix Sort outperforms all other algorithms, maintaining consistent performance across different input types. Unlike comparison-based sorts, its time complexity is linear, making it ideal for large datasets. The results confirm its suitability for efficiently sorting numerical data regardless of initial order.

4 Discussion

- **What are the best, average, and worst cases for the given algorithms in terms of the given input data to be sorted?** - The **best case** occurs when the input is already sorted, leading to minimal comparisons and swaps for algorithms like Insertion Sort and Shaker Sort. - The **average case** depends on the randomness of input distribution and generally follows the expected asymptotic complexity. - The **worst case** occurs in reverse-sorted data, significantly impacting Insertion and Shaker Sort due to excessive shifts and swaps.

- **Do the obtained results (running times of your algorithm implementations) match their theoretical asymptotic complexities?** - Yes, the experimental results align with the theoretical complexities. - Radix Sort consistently shows $O(nk)$ efficiency, while comparison-based sorts exhibit their expected $O(n^2)$ or $O(n \log n)$ complexities.

- **Do Shaker Sort and Comb Sort improve performance compared to Bubble Sort? If so, how do their approaches contribute to this improvement?** - Both improve over Bubble Sort by reducing unnecessary swaps. - **Shaker Sort** moves in both directions, preventing unnecessary iterations. - **Comb Sort** reduces the gap between compared elements, leading to faster convergence towards a sorted state.
- **Does Shell Sort perform better than Insertion Sort for larger datasets? Under what conditions does Insertion Sort still perform well?** - Shell Sort outperforms Insertion Sort in larger datasets due to reduced swaps and gap-based sorting. - Insertion Sort is still efficient on **small** or **nearly sorted** datasets due to its simple structure and minimal element movement.
- **Given that Radix Sort is a non-comparison-based sorting algorithm, how does it handle large numerical ranges efficiently?** - Radix Sort efficiently processes large numbers by sorting digits independently, making it highly efficient for integer sorting without direct element comparisons.

References

- https://bilgisayarkavramlari.com/2008/08/09/siralama-algoritmalari-sorting-algorithms/
- Sedgewick, R. (1998). *Algorithms in C*. Addison-Wesley.
- GeeksForGeeks. (2023). *Sorting Algorithm Comparisons*. Retrieved from `https://www.geeksforgeeks.org/sorting-algorithms/`
- VisuAlgo. (2023). *Sorting Algorithm Visualizations*. Retrieved from `https://visualgo.net/en/sorting`