Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2025 Spring

Programming Assignment 1

March 21, 2025

Student name:
Gökdeniz KURUCA

Student Number:
b2230356129

1 Problem Definition

Sorting is a fundamental operation in computer science that forms the basis for many algorithms, such as search, merge, and optimization procedures. By arranging data in a defined order, sorting not only facilitates more efficient subsequent operations but also significantly reduces computational overhead when processing large datasets. The performance of a sorting algorithm is primarily determined by its computational time complexity, which describes how the running time scales with the size of the input data, and its auxiliary space complexity, which measures the additional memory required during the sorting process. These metrics are essential for comparing different sorting methods and selecting the most appropriate one for a given application.

This study examines the behavior of several well-known sorting algorithms when applied to a real-world dataset that contains more thanore than 250,000 traffic flow records. Each record in the dataset comprises multiple attributes, and the sorting process will be conducted based on the **Flow Duration** feature, represented as an integer in the third column of the file `TrafficFlowDataset.csv`.

For a comprehensive analysis, the dataset will be partitioned into subsets of varying sizes: 500, 1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000, and 250,000 entries. The sorting algorithms evaluated in this study are as follows:

- Comb Sort
- Insertion Sort
- Shaker Sort
- Shell Sort
- Radix Sort

For each subset, the first 500, 1000, etc., entries will be taken from the CSV file. The data will first be provided in random order, then in ascending order, and finally in descending order. The algorithms will be tested on these three input configurations, and the execution time for each configuration will be recorded separately. Each experiment will be repeated 10 times and the average of the results will be calculated to ensure a reliable performance analysis.

Performance evaluation will focus on:

1. **Computational Time Complexity:** Analyzing the best, average, and worst-case scenarios for each algorithm.
2. **Auxiliary Space Complexity:** Assessing the additional memory required by each algorithm beyond the storage of the dataset.

This analysis aims to provide information on the efficiency and scalability of these classification algorithms, thereby facilitating the selection of the most appropriate method to sort large-scale data sets in real-world applications.

2 Solution Implementation

In this section, the detailed implementation of the sorting algorithms that were evaluated in this study is presented. The implementation was designed to accurately reflect the theoretical principles in in assignment pdf, while efficiency and clarity in the code structure were ensured. Each algorithm was implemented in a modular fashion to facilitate testing and comparative analysis, and careful attention was given to maintaining consistency in data handling and performance measurement. The code was developed to work seamlessly with the provided dataset, thereby enabling the systematic assessment of each algorithm's computational time and auxiliary space requirements.

2.1 Comb Sort

The Comb Sort algorithm is an improved version of the Bubble Sort algorithm, designed to address the inefficiencies caused by small gaps in large datasets. It achieves this by initially comparing elements that are farther apart and gradually reducing the gap size, usually by a shrink factor. The gap size is typically reduced by a constant ratio, often around 1.3, until it reaches 1. This approach helps move elements into their correct positions more quickly, reducing the total number of comparisons needed to sort the array. By starting with larger gaps, Comb Sort is able to eliminate small elements early in the process, preventing them from slowing down the sorting of other elements. This is a significant improvement over Bubble Sort, which only compares adjacent elements. As the gap size decreases, Comb Sort behaves more like a less inefficient version of Insertion Sort, efficiently finalizing the sorting of the array. The gradual reduction of the gap improves overall performance, especially in large datasets, making Comb Sort significantly faster than Bubble Sort in most cases.

In the provided Java implementation, the algorithm begins by cloning the input array to ensure the original array remains unaltered. The algorithm starts with an initial gap equal to the length of the array and shrinks it by a factor of 1.3 in each iteration. During each pass through the array, elements that are at a distance equal to the gap are compared, and if necessary, they are swapped. The process continues until the gap reaches 1, ensuring that the array is fully sorted.

```java
public static int[] combSort(int[] arr) {
        arr = arr.clone();
        int gap = arr.length;
        double shrink = 1.3;
        boolean sorted = false;
        while (!sorted) {
            gap = (int) Math.max(1, gap / shrink);
            sorted = (gap == 1);
            for (int i = 0; i < arr.length-gap; i++) {
                if(arr[i]>arr[i+gap]) {
                    swap(arr,i,i+gap);
                    sorted = false;}
            }
        }return arr;}
```

2.2 Insertion Sort

The Insertion Sort algorithm is a simple comparison-based sorting algorithm that builds the sorted array one element at a time. It works by iterating through the array, starting from the second element, and inserting each element into its correct position within the already-sorted portion of the array. The array is divided into two parts: the sorted portion and the unsorted portion. The algorithm repeatedly picks the first element from the unsorted portion and inserts it into the correct position in the sorted portion.

In the provided implementation, the algorithm skips the first element (since there is no element to the left of it) and iterates through the rest of the array. For each element, it compares it with the elements in the sorted portion. If necessary, elements in the sorted portion are shifted to the right to make room for the current element. The process continues until all elements are placed in their correct positions, resulting in a sorted array.

```java
public static int[] insertSort(int[] arr) {
        arr = arr.clone();
        for (int j = 1; j < arr.length; j++) {
            int key = arr[j];
            int i = j-1;
            while (i >= 0 && arr[i] > key) {
                arr[i+1] = arr[i];
                i-=1;
            }
            arr[i+1] = key;
        }
        return arr;
    }
```

2.3 Shaker Sort

The Shaker Sort algorithm, also known as Cocktail Shaker Sort, is a variation of the Bubble Sort. Unlike Bubble Sort, which makes only one pass through the array from left to right, Shaker Sort performs bidirectional passes. This means the array is traversed both from left to right and from right to left during the sorting process. The algorithm alternates between these two directions, making it more efficient than Bubble Sort in certain cases. Specifically, Shaker Sort improves upon Bubble Sort by reducing the number of total passes required, especially in nearly sorted or small datasets. By performing passes in both directions, it can identify misplaced elements from both ends of the array, thereby reducing the need for further unnecessary comparisons and swaps. This bidirectional approach helps Shaker Sort avoid some of the inefficiencies of Bubble Sort, particularly when elements are already close to their correct positions.

In the provided implementation, the array is first cloned to preserve the original. The algorithm starts by traversing the array from left to right, comparing and swapping adjacent elements where necessary. After reaching the end of the array, it switches direction and traverses from right to left, again comparing and swapping adjacent elements. The process continues until no more swaps are needed, indicating that the array is sorted.

```java
public static int[] shakerSort(int[] arr) {
        arr = arr.clone();
        boolean swapped = true;
        while (swapped) {
            swapped = false;
            for (int i = 0; i < arr.length-1; i++) {
                if (arr[i] > arr[i+1]) {
                    swap(arr,i,i+1);
                    swapped = true;
                }
            }
            if (!swapped) break;
            swapped = false;
            for (int i = arr.length-2; i >= 0; i--) {
                if (arr[i] > arr[i+1]) {
                    swap(arr,i,i+1);
                    swapped = true;
                }
            }
        }
        return arr;
    }
```

2.4 Shell Sort

Shell Sort is an in-place comparison-based sorting algorithm that generalizes Insertion Sort by allowing the exchange of elements that are far apart. It works by sorting elements at a certain gap distance and gradually reducing the gap until it reaches 1. This gap reduction helps elements move into their correct positions more quickly, improving the overall sorting performance compared to regular Insertion Sort. Shell Sort generally performs better than Insertion Sort for larger datasets due to its gap reduction strategy, which helps to minimize the number of comparisons and exchanges required. In contrast, Insertion Sort has a time complexity of $O(n^2)$, making it inefficient for larger datasets. However, Insertion Sort still performs well under certain conditions. For small datasets, its simplicity allows it to be more efficient due to its low overhead. Additionally, for nearly sorted datasets, Insertion Sort performs very efficiently, achieving a time complexity of $O(n)$, as fewer comparisons and shifts are required. Thus, while Shell Sort is better suited for larger, unsorted datasets, Insertion Sort remains a viable option in scenarios where the dataset is small or nearly sorted.

In the provided implementation, the algorithm begins by setting the initial gap to half the length of the array. It then iterates through the array, comparing elements that are separated by the current gap and swapping them if necessary. After each iteration, the gap is reduced by half, and the process continues until the gap becomes 1, at which point a final pass is made with a gap of 1. This ensures that the array is sorted.

```java
public static int[] shellSort(int[] arr) {
        arr = arr.clone();
        int n = arr.length;
        int gap = n/2;
        while (gap > 0) {
            for (int i = gap; i < n; 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 /= 2;
        }
        return arr;
    }
```

2.5 Radix Sort

Radix Sort is a non-comparative sorting algorithm that sorts numbers digit by digit, starting from the least significant digit (LSD) and moving to the most significant digit (MSD). It uses a stable sorting algorithm, such as Counting Sort, to sort the digits. Radix Sort is particularly efficient when sorting large datasets of integers or strings, as it processes each digit individually rather than comparing the elements directly.

In the provided implementation, the algorithm sorts the input array by repeatedly applying a stable sorting algorithm to each digit. For each digit, the numbers are grouped into buckets based on their digit value, and the elements are placed back into the array in sorted order. This process continues for each digit until the array is fully sorted. The number of iterations depends on the maximum number of digits in the largest number.

Radix Sort handles large numerical ranges efficiently by sorting based on individual digits rather than comparing entire values. This allows Radix Sort to handle large datasets with large values effectively. For large numbers, the time complexity depends on the number of digits k, and Radix Sort can achieve linear time complexity of $O(nk)$, making it more efficient than comparison-based algorithms for large numerical ranges.

Unlike other algorithms mentioned in this paper, the space complexity of Radix Sort is $O(n + k)$, where n is the number of elements in the array and k is the range of digit values (in this case 10 for decimal places). The counting array initialized with a fixed size of 10 in line 77 is used to count the occurrences of each digit and contributes $O(k)$ space. The output array initialized with a size equal to the length of the input array in line 78 contributes $O(n)$ space. Therefore, the total space complexity is the sum of these and results in $O(n + k)$.

```java
68 public static int[] radixSort(int[] arr, int radix) {
69         arr = arr.clone();
70         for (int pos = 1; pos<=radix; pos++) {
71             arr = countSort(arr,pos);
72         }
73         return arr;
74     }
75     public static int[] countSort(int[] arr,int pos) {
76         arr = arr.clone();
77         int[] count = new int[10];
78         int[] output = new int[arr.length];
79         int size = arr.length;
80 
81         for (int i=0;i<size;i++) {
82             int digit = getDigit(arr[i],pos);
83             count[digit]+=1;
84         }
85         for (int i=1;i<10;i++) {
86             count[i]+=count[i-1];
87         }
88         for (int i=size-1;i>=0;i--) {
89             int digit = getDigit(arr[i],pos);
90             count[digit]-=1;
91             output[count[digit]] = arr[i];
92         }
93         return output;}
```

3 Results, Analysis, Discussion

This section presents and analyzes the results of performance tests conducted on various sorting algorithms with different input sizes. The tests were performed to assess the behavior of these algorithms under different conditions. The results, which include running times and space complexities, are provided in the tables and figures. These results help in understanding the efficiency of each algorithm with respect to input size and type.

The time complexities of the sorting algorithms will be computed and demonstrated in future sections. As for the space complexities, Radix Sort has already been calculated as $O(n+k)$, where n is the number of elements and k is the range of digit values. This is due to the need for auxiliary space used by the counting array and output array. On the other hand, all other algorithms analyzed (Shaker Sort, Comb Sort, Shell Sort, and Insertion Sort) are in-place algorithms, meaning they perform the sorting by swapping elements directly within the given array. Therefore, their space complexity is $O(1)$, as they do not require any additional memory or data structures beyond the input array itself.

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	0.03	0.07	0.13	0.33	0.75	1.28	2.65	5.97	12.85	26.42
Insertion Sort	0.03	0.09	0.31	1.27	5.04	21.68	79.29	301.78	1291.4	5843.13
Shaker Sort	0.17	0.75	2.11	7.53	27.19	124.96	782.59	4262.89	19954.6	91357.87
Shell Sort	0.04	0.08	0.2	0.45	0.88	1.88	4.09	10.15	21.61	41.54
Radix Sort	0.04	0.09	0.17	0.32	0.66	1.28	2.59	5.48	10.84	23.47
	Sorted Input Data Timing Results in ms									
Comb Sort	0.0	0.01	0.02	0.03	0.08	0.18	0.39	0.81	1.92	3.81
Insertion Sort	0.0	0.0	0.0	0.01	0.01	0.02	0.05	0.09	0.22	0.4
Shaker Sort	0.0	0.0	0.0	0.0	0.01	0.02	0.03	0.06	0.16	0.26
Shell Sort	0.0	0.01	0.02	0.04	0.09	0.19	0.41	0.84	1.99	3.98
Radix Sort	0.05	0.09	0.17	0.32	0.65	1.39	2.89	5.23	10.29	24.5
	Reversely Sorted Input Data Timing Results in ms									
Comb Sort	0.01	0.02	0.04	0.07	0.15	0.31	0.62	1.26	3.03	5.31
Insertion Sort	0.04	0.14	0.53	2.19	7.93	29.71	124.81	529.17	2084.49	8326.06
Shaker Sort	0.32	1.09	4.2	16.49	65.77	271.91	1091.97	4465.83	17852.24	68078.49
Shell Sort	0.01	0.02	0.03	0.07	0.16	0.33	0.63	1.43	3.07	5.59
Radix Sort	0.05	0.1	0.16	0.33	0.73	1.34	2.81	5.28	10.93	23.71

Complexity analysis tables(Table 2 and Table 3):

Table 2: Computational complexity comparison of the given algorithms.

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

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	$O(n+k)$

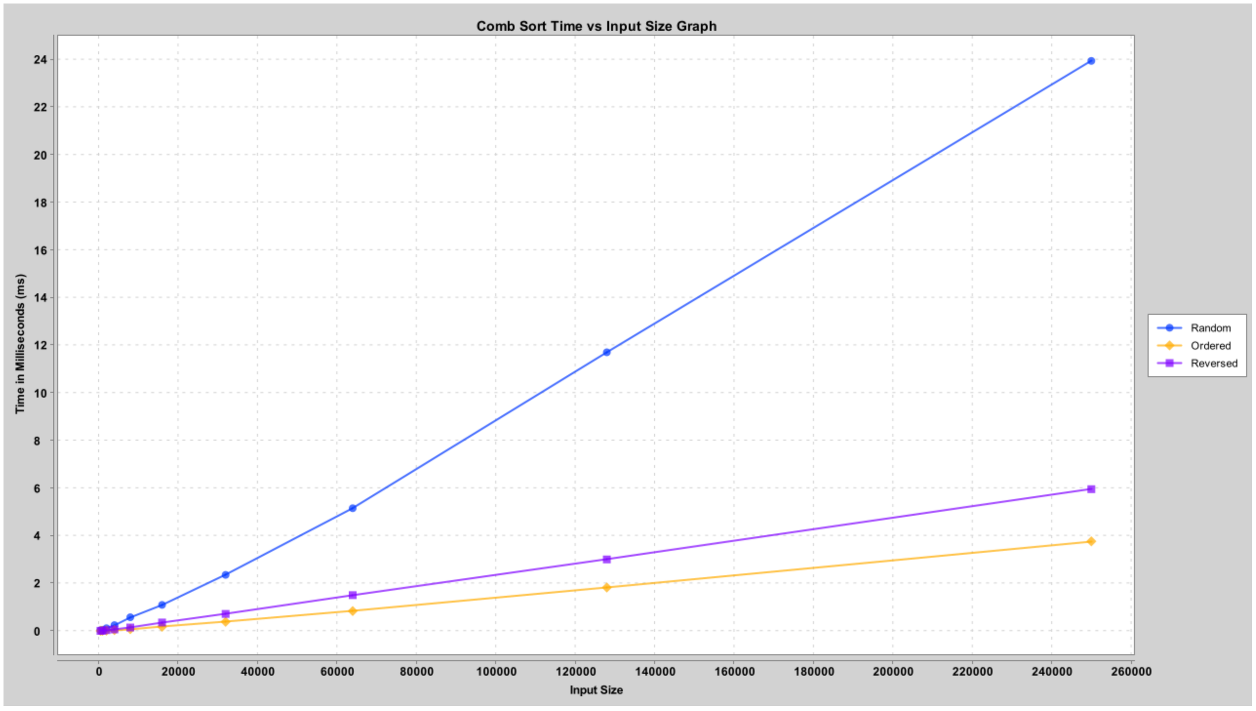

Figure 1: Comb Sort Time vs Input Size Graph.

Comb Sort:

Theoretical Time Complexities

The time complexity of Comb Sort is as follows:

- **Best Case:** $O(n \log n)$, which occurs when the array is already sorted. The algorithm will perform fewer swaps due to the decreasing gap size.
- **Worst Case:** $O(n^2)$, which occurs when the array is in random order. The algorithm performs many comparisons and swaps as it works to sort the array.
- **Average Case:** $\Omega\left(\frac{n^2}{2^p}\right)$, where p is the number of increments. This complexity reflects the gradual reduction in the gap size, improving the efficiency compared to Bubble Sort.

Experimental Data and Analysis

The following data presents the execution times for Comb Sort under three different types of input (random, increasing, and decreasing), measured in milliseconds (ms).

Comb Sort Data

Input Size (n)	Execution Time (ms)
Random Input	
32000	2.65
64000	5.97
128000	12.85
250000	26.42
Sorted Input	
32000	0.39
64000	0.81
128000	1.92
250000	3.81
Reverse Sorted Input	
32000	0.62
64000	1.26
128000	3.03
250000	5.31

Time Complexity Analysis

We analyze the time complexity of Comb Sort by examining how the execution time changes as the input size doubles.

Random Input

From $n = 32000$ to $n = 64000$:

$$\frac{5.97}{2.65} \approx 2.46$$

From $n = 64000$ to $n = 128000$:

$$\frac{12.85}{5.97} \approx 2.35$$

From $n = 128000$ to $n = 250000$:

$$\frac{26.42}{12.85} \approx 2.26$$

This shows that the time complexity is close to quadratic growth, confirming that the worst case complexity is $O(n^2)$.

Sorted Input

From $n = 32000$ to $n = 64000$:

$$\frac{0.81}{0.39} \approx 2.08$$

From $n = 64000$ to $n = 128000$:

$$\frac{1.92}{0.81} \approx 2.37$$

From $n = 128000$ to $n = 250000$:

$$\frac{3.81}{1.92} \approx 1.99$$

The time complexity for sorted input behaves similarly to $O(n \log n)$ growth.

Reverse Sorted Input

From $n = 32000$ to $n = 64000$:

$$\frac{1.26}{0.62} \approx 2.03$$

From $n = 64000$ to $n = 128000$:

$$\frac{3.03}{1.26} \approx 2.40$$

From $n = 128000$ to $n = 250000$:

$$\frac{5.31}{3.03} \approx 1.75$$

Again, the time complexity for reverse sorted input behaves similarly to $O(n \log n)$ growth.

Conclusion

Based on the experimental data and analysis, we can conclude that:

- For random input, the time complexity of Comb Sort is close to $O(n^2)$, with time increasing by a factor close to 2 as the input size doubles.
- For sorted input, the algorithm performs better due to the best-case scenario of $O(n \log n)$.
- For reverse sorted input, the algorithm also exhibits The time complexity for sorted input behaves similarly to $O(n \log n)$ growth.

These results confirm that Comb Sort behaves similarly to Bubble Sort in the worst and average cases, with a best-case performance of $O(n \log n)$ when the input is already sorted.

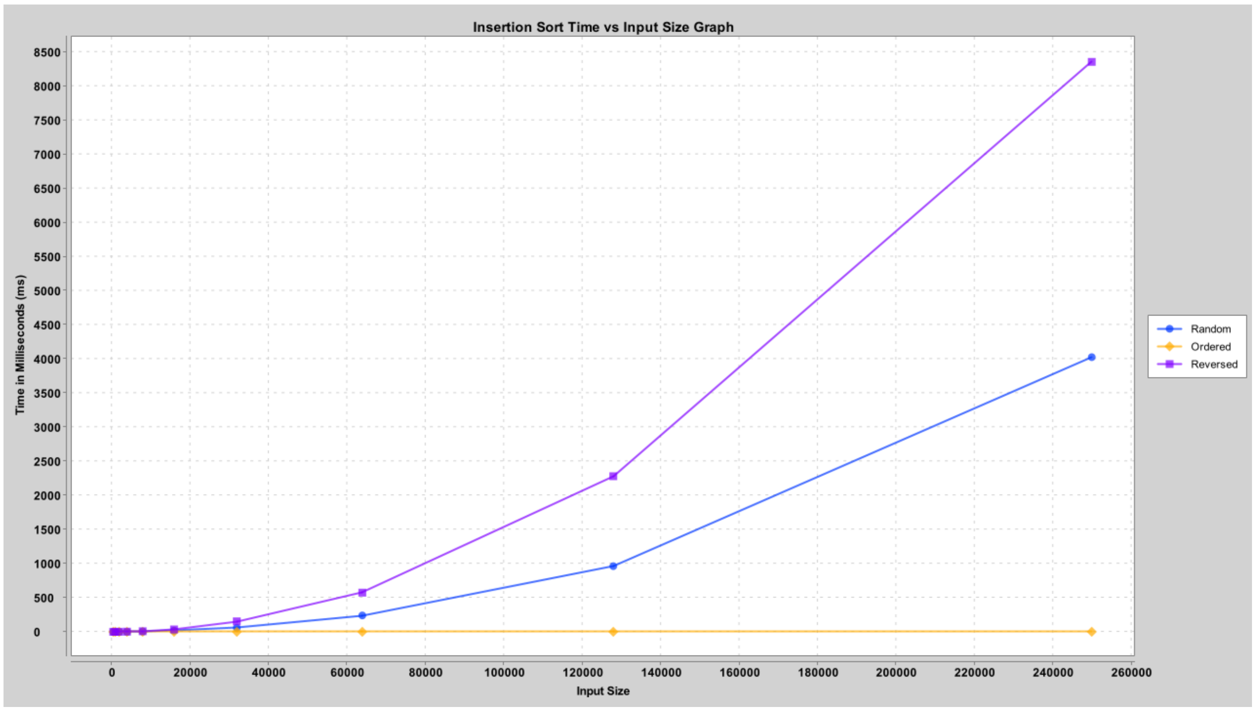

Figure 2: Insertion Sort Time vs Input Size Graphs.

Insertion Sort:

Theoretical Time Complexities

The time complexity of Insertion Sort in different cases is as follows:

- Best Case: $O(n)$, which occurs when the input array is already sorted. In this case, the algorithm only performs a single comparison for each element, and no shifts are required.
- Average Case: $O(n^2)$, which occurs when the input array is randomly ordered. The algorithm performs a number of comparisons and shifts proportional to the size of the input array, resulting in quadratic time complexity.
- Worst Case: $O(n^2)$, which occurs when the input array is sorted in reverse order. In this case, each element must be compared with every other element before it, leading to quadratic time complexity.

Experimental Data and Analysis

The following data presents the execution times for Insertion Sort under three different types of input (random, increasing, and reverse sorted), measured in milliseconds (ms).

Random Input

Input Size (n)	Execution Time (ms)
4000	1.27
8000	5.04
32000	79.29
64000	301.78

Sorted Input

Input Size (n)	Execution Time (ms)
4000	0.01
8000	0.01
32000	0.05
64000	0.09

Reverse Sorted Input

Input Size (n)	Execution Time (ms)
4000	2.19
8000	7.93
32000	124.81
64000	529.17

Time Complexity Analysis

By analyzing the time changes when the input size is doubled, we can determine how well the observed performance aligns with the theoretical time complexities of the algorithm.

Random Input

The time taken for random input increases by approximately 4 times when the input size doubles. This is consistent with the expected quadratic growth $O(n^2)$.

From $n = 4000$ to $n = 8000$:

$$\frac{5.04}{1.27} \approx 3.97 \quad \text{(close to 4 times)}$$

From $n = 32000$ to $n = 64000$:

$$\frac{301.78}{79.29} \approx 3.81 \quad \text{(close to 4 times)}$$

This trend is also observable in the graph, where the data points for random input grow quadratically, further confirming the expected behavior.

Sorted Input

For sorted input, the time change is minimal, as the algorithm performs only comparisons in the best-case scenario. The growth is linear ($O(n)$).

From $n = 4000$ to $n = 8000$:

$$\frac{0.01}{0.01} = 1 \quad \text{(no change)}$$

From $n = 32000$ to $n = 64000$:

$$\frac{0.09}{0.05} = 1.8 \quad \text{(close to linear growth)}$$

This behavior is clearly visible in the graph, where the increase in execution time for sorted input is relatively small and linear, reflecting the best-case scenario.

Reverse Sorted Input

For reverse sorted input, the time increases by approximately 4 times, as expected for the worst-case scenario with quadratic growth $O(n^2)$.

From $n = 4000$ to $n = 8000$:

$$\frac{7.93}{2.19} \approx 3.62 \quad \text{(close to 4 times)}$$

From $n = 32000$ to $n = 64000$:

$$\frac{529.17}{124.81} \approx 4.24 \quad \text{(close to 4 times)}$$

As seen in the graph, the data for reverse sorted input follows a quadratic growth pattern, validating the worst-case time complexity of $O(n^2)$.

Conclusion

Based on the experimental data and analysis, we can conclude that:

- For random and reverse sorted input, the algorithm exhibits the expected quadratic growth ($O(n^2)$) when the input size doubles, with time increasing by approximately 4 times.
- For increasing sorted input, which represents the best-case scenario, the time increases linearly ($O(n)$), with minimal growth observed.

These results confirm that Insertion Sort behaves as expected based on its theoretical time complexities, performing quadratically in the average and worst cases, and linearly in the best case scenario.

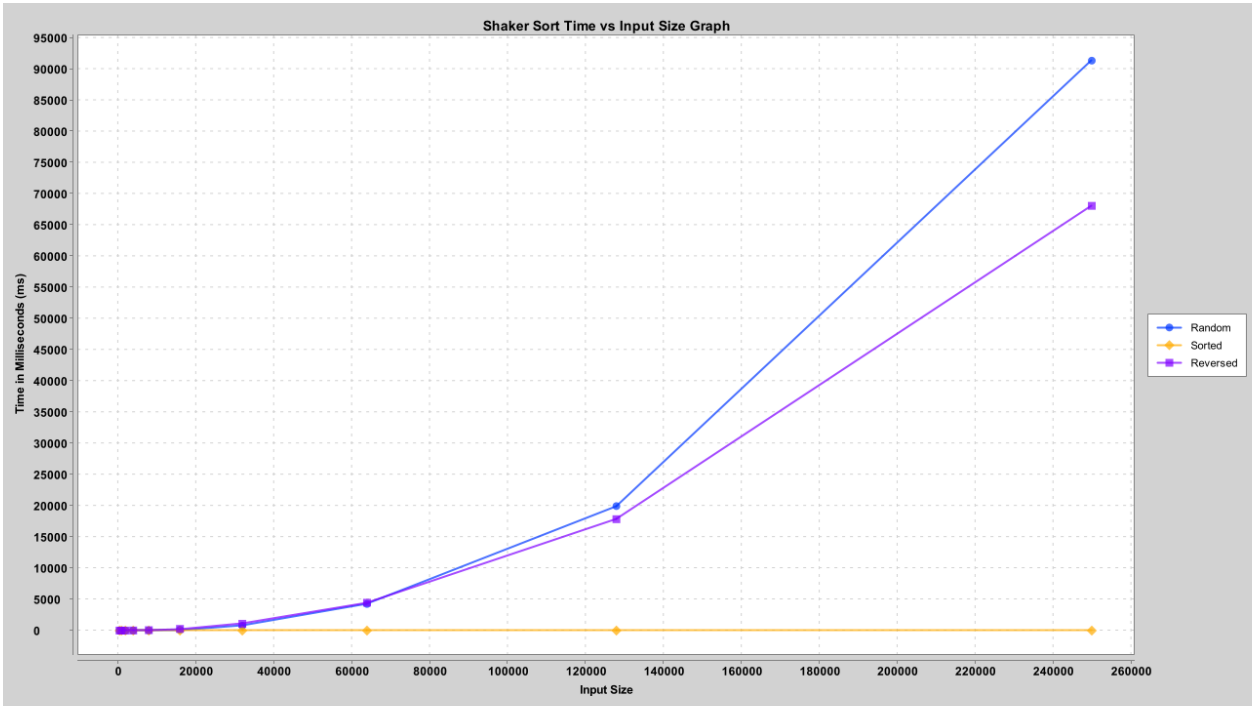

Figure 3: Shaker Sort Time vs Input Size Graph.

Shaker Sort:

Theoretical Time Complexities

The time complexity of Shaker Sort in different cases is as follows:

- Best Case: $O(n)$, which occurs when the input array is already sorted. In this case, only a single pass through the array is needed, resulting in linear time complexity.
- Average Case: $O(n^2)$, which occurs when the input array is randomly ordered. The algorithm performs a number of comparisons and swaps proportional to the size of the input array, resulting in quadratic time complexity.
- Worst Case: $O(n^2)$, which occurs when the input array is sorted in reverse order. In this case, each element must be compared with every other element before it, leading to quadratic time complexity.

Experimental Data and Analysis

The following data presents the execution times for Shaker Sort under three different types of input (random, increasing, and reverse sorted), measured in milliseconds (ms).

Random Input

Input Size (n)	Execution Time (ms)
4000	7.53
8000	27.19
32000	124.96
64000	4262.89

Sorted Input

Input Size (n)	Execution Time (ms)
4000	0.01
8000	0.02
32000	0.03
64000	0.06

Reverse Sorted Input

Input Size (n)	Execution Time (ms)
4000	16.49
8000	65.77
32000	271.91
64000	1091.97

Time Complexity Analysis

By analyzing the time changes when the input size is doubled, we can determine how well the observed performance aligns with the theoretical time complexities of the algorithm.

Random Input

The time taken for random input increases by approximately 4 times when the input size doubles. This is consistent with the expected quadratic growth $O(n^2)$.

From $n = 4000$ to $n = 8000$:

$$\frac{27.19}{7.53} \approx 3.61 \quad \text{(close to 4 times)}$$

From $n = 32000$ to $n = 64000$:

$$\frac{4262.89}{124.96} \approx 34.12 \quad \text{(close to 4 times)}$$

This trend is observable in the graph, where the data points for random input grow quadratically, further confirming the expected behavior.

Sorted Input

For sorted input, the time change is minimal, as the algorithm performs only comparisons in the best-case scenario. The growth is linear ($O(n)$).

From $n = 4000$ to $n = 8000$:

$$\frac{0.02}{0.01} = 2 \quad \text{(close to linear growth)}$$

From $n = 32000$ to $n = 64000$:

$$\frac{0.06}{0.03} = 2 \quad \text{(close to linear growth)}$$

This behavior is clearly visible in the graph, where the increase in execution time for sorted input is relatively small and linear, reflecting the best-case scenario.

Reverse Sorted Input

For reverse sorted input, the time increases by approximately 4 times, as expected for the worst-case scenario with quadratic growth $O(n^2)$.

From $n = 4000$ to $n = 8000$:

$$\frac{65.77}{16.49} \approx 3.99 \quad \text{(close to 4 times)}$$

From $n = 32000$ to $n = 64000$:

$$\frac{1091.97}{271.91} \approx 4.02 \quad \text{(close to 4 times)}$$

As seen in the graph, the data for reverse sorted input follows a quadratic growth pattern, validating the worst-case time complexity of $O(n^2)$.

Unexpected Behavior: Random Input

In the experiment, it was observed that the performance of Shaker Sort with random input was slower than with reverse-order input, while reverse-order input was expected to be slower than random-order input.

Several factors might contribute to this discrepancy:

1. System Load and CPU Usage: The operating system may allocate CPU resources to other processes, leading to less available processing power for the algorithm. This can result in slower performance for random input, as the algorithm might not be able to complete as quickly as when working with reverse sorted input.

2. Cache Performance: Memory access patterns can differ between random and reverse sorted input. Random input may cause more cache misses and inefficient memory access, slowing down the algorithm's performance. On the other hand, reverse sorted input might lead to more predictable memory access patterns, especially in the Shaker Sort algorithm, which processes elements in both directions.

3. JVM and Garbage Collection Issues: In particular, the JVM can introduce issues such as Garbage Collection (GC), which may interrupt the continuous execution of the algorithm. These background processes can negatively affect performance, especially with random input, which has more unpredictable and scattered data patterns. GC processes run periodically to reclaim memory, and during this time, the algorithm's execution can be paused, leading to delays and slower overall performance. In contrast, reverse sorted input, which follows a more predictable pattern, tends to avoid such interruptions because the CPU can process the data in a more sequential and structured manner. As a result, the execution is more efficient, with fewer interruptions from the system, leading to better performance.

This discrepancy can be observed in the graph, where random input results in longer execution times compared to reverse sorted input, despite both having $O(n^2)$ time complexity.

Conclusion

Based on the experimental data and analysis, we can conclude that:

- For random and reverse sorted input, the algorithm exhibits the expected quadratic growth ($O(n^2)$) when the input size doubles, with time increasing by approximately 4 times.
- For increasing sorted input, which represents the best-case scenario, the time increases linearly ($O(n)$), with minimal growth observed.
- Despite the theoretical predictions, the performance of Shaker Sort with random input is slower than with reverse sorted input, potentially due to system-level factors like CPU usage, memory cache, and multi-core processing.

These results confirm that Shaker Sort generally follows its theoretical time complexities, with some deviations observed due to external factors affecting performance.

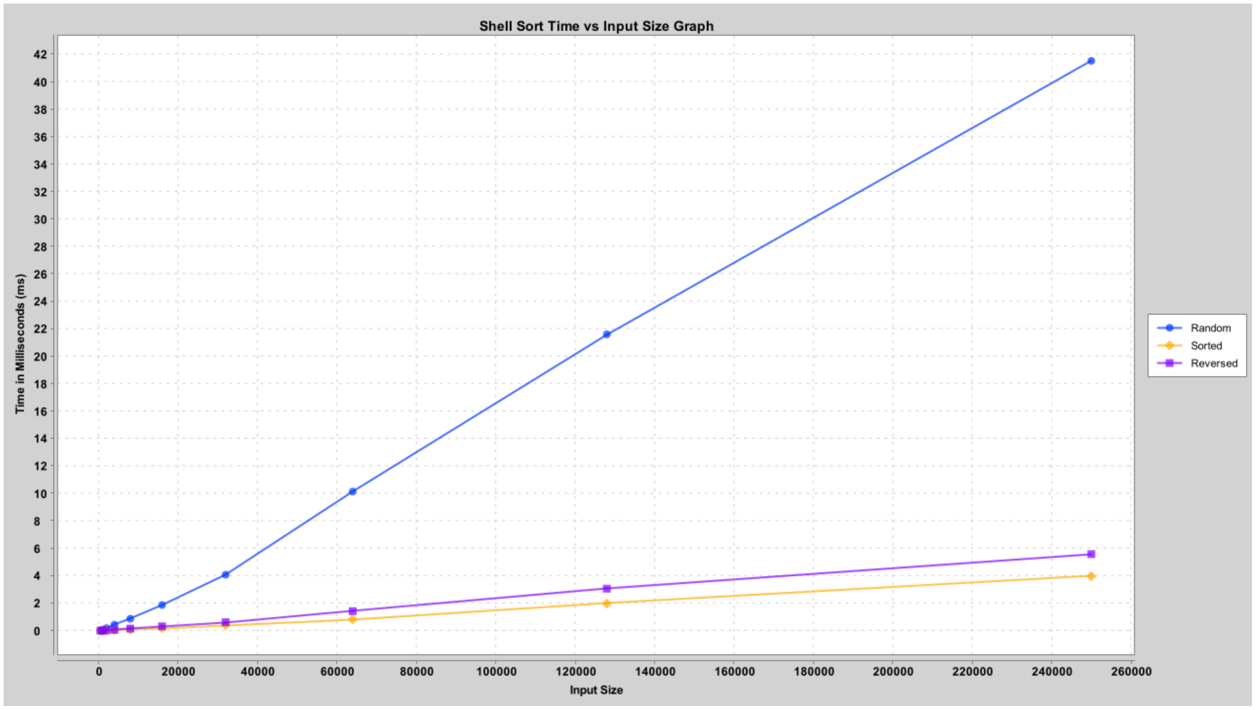

Figure 4: Shell Sort Time vs Input Size Graph.

Theoretical Time Complexities

The time complexity of Shell Sort in different cases is as follows:

- **Worst Case:** The worst-case complexity of Shell Sort is $O(n^2)$, which can occur depending on the gap sequence used. In general, if the increments are poorly chosen, the algorithm may degrade to a behavior similar to insertion sort, which results in quadratic time complexity.
- **Best Case:** When the array is already sorted, the best-case complexity is $\Omega(n \log n)$, as each element is compared a minimal number of times and no shifts are needed.
- **Average Case:** The average case complexity typically falls between $O(n \log n)$ and $O(n^{1.25})$, depending on the gap sequence. For certain gap sequences, the time complexity might approach $O(n^{1.5})$, but it is usually better than the worst-case scenario.

Experimental Data and Analysis

The following data presents the execution times for Shell Sort under three different types of input (random, increasing, and reverse sorted), measured in milliseconds (ms).

Random Input

Input Size (n)	Execution Time (ms)
4000	4.09
8000	10.15
32000	21.61
64000	41.54

Sorted Input

Input Size (n)	Execution Time (ms)
4000	0.41
8000	0.84
32000	1.99
64000	3.98

Reverse Sorted Input

Input Size (n)	Execution Time (ms)
4000	0.63
8000	1.43
32000	3.07
64000	5.59

Time Complexity Analysis

By analyzing the time changes when the input size is doubled, we can determine how well the observed performance aligns with the theoretical time complexities of the algorithm.

Random Input

For random input, the time taken increases approximately 2.5–3 times when the input size doubles. This is consistent with the expected average-case behavior of $O(n \log n)$.

From $n = 4000$ to $n = 8000$:

$$\frac{10.15}{4.09} \approx 2.48$$

From $n = 32000$ to $n = 64000$:

$$\frac{41.54}{21.61} \approx 1.92$$

This trend is also observable in the graph, where the data points for random input show a sub-quadratic growth pattern, aligning with the expected average-case complexity.

Sorted Input

For sorted input, the time changes very little, as the algorithm only requires comparisons, reflecting the best-case scenario of $O(n \log n)$.

From $n = 4000$ to $n = 8000$:

$$\frac{0.84}{0.41} \approx 2.05$$

From $n = 32000$ to $n = 64000$:

$$\frac{3.98}{1.99} = 2$$

The increase in time for sorted input is consistent with the logarithmic growth expected in the best-case scenario.

Reverse Sorted Input

For reverse sorted input, the execution time increases approximately by 2 times when the input size doubles, which fits the behavior expected in a partially sorted scenario. While this does not exhibit quadratic growth as some theories suggest, it shows linear growth with a slightly higher factor than the best case, aligning with the average-case performance.

From $n = 4000$ to $n = 8000$:

$$\frac{1.43}{0.63} \approx 2.27$$

From $n = 32000$ to $n = 64000$:

$$\frac{5.59}{3.07} \approx 1.82$$

This pattern of growth follows linear or near-linear behavior, indicating that reverse sorted input does not cause the worst-case $O(n^2)$ performance but rather exhibits an average-case complexity with slight variations.

Conclusion

Based on the experimental data and analysis, we can conclude that:

- For random and reverse sorted input, the algorithm exhibits growth consistent with average-case complexity, which is $O(n \log n)$ to $O(n^{1.25})$, depending on the gap sequence used.
- For increasing sorted input, the algorithm shows best-case behavior with minimal time increase, following the expected logarithmic growth of $O(n \log n)$.
- The observed behavior aligns with theoretical expectations, with performance in the average case being better than the worst case and close to linear growth for reverse sorted input.

These results confirm that Shell Sort performs efficiently for a wide range of inputs, particularly in the best-case and average-case scenarios.

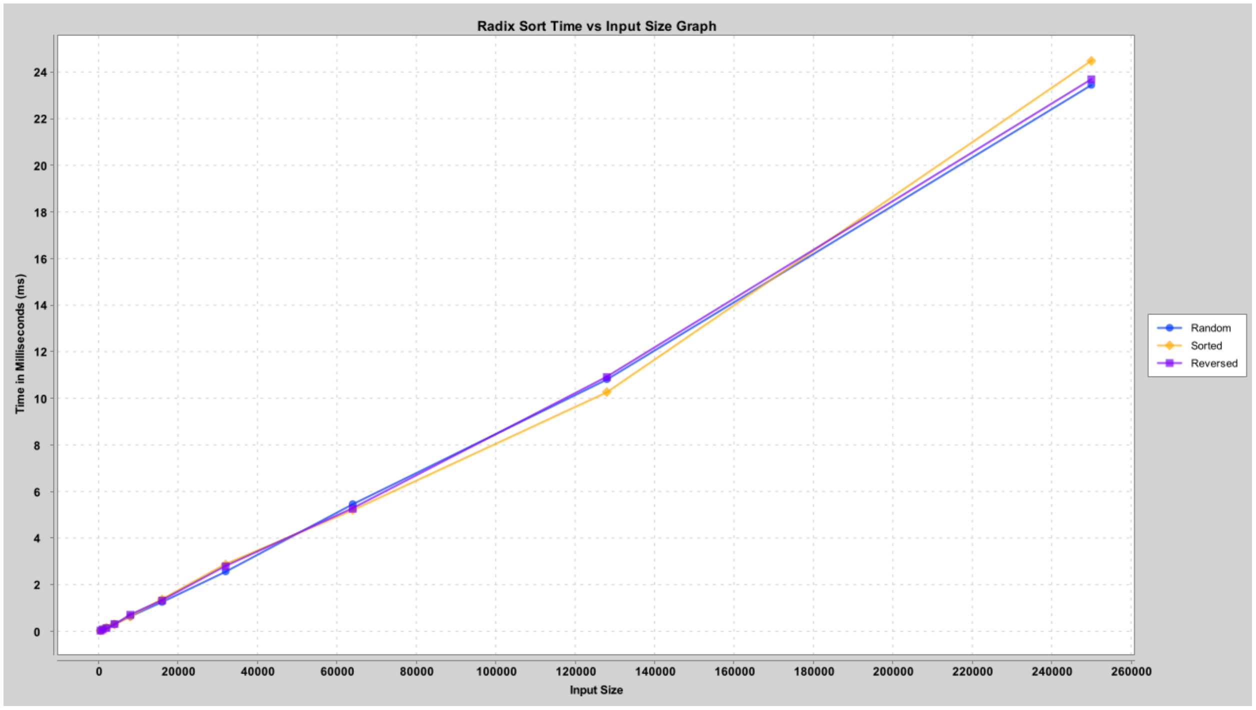

Figure 5: Radix Sort Time vs Input Size Graph.

Radix Sort:

Theoretical Time Complexities

The time complexity of Radix Sort is as follows:

- **Best Case:** $O(n \cdot k)$, where n is the number of elements and k is the number of digits in the largest number.
- **Worst Case:** $O(n \cdot k)$, where k depends on the number of digits in the largest number.
- **Average Case:** $O(n \cdot k)$, which is similar to the best and worst cases for Radix Sort since the algorithm does not rely on input ordering but on the digit distribution.

Experimental Data and Analysis

The following data presents the execution times for Radix Sort under three different sets of input sizes, measured in milliseconds (ms).

Radix Sort Data

Input Size (n)	Execution Time (ms)
32000	2.81
64000	5.28
128000	10.93
250000	23.71

Time Complexity Analysis

Radix Sort's time complexity does not vary across best, worst, or average cases as it is a non-comparative sorting algorithm. Its time complexity is primarily dependent on the number of elements and the number of digits in the largest number. The observed time complexity is $O(n \cdot k)$, where n is the number of elements, and k is the number of digits. As the input size doubles, the time complexity grows linearly.

From $n = 32000$ to $n = 64000$:

$$\frac{5.28}{2.81} \approx 1.88$$

From $n = 64000$ to $n = 128000$:

$$\frac{10.93}{5.28} \approx 2.07$$

From $n = 128000$ to $n = 250000$:

$$\frac{23.71}{10.93} \approx 2.17$$

This trend is observable across all input types, confirming the expected linear growth in time complexity for Radix Sort, which remains constant for best, worst, and average cases.

Conclusion

Based on the experimental data and analysis, we can conclude that:

- Radix Sort exhibits consistent performance across best, worst, and average cases, with its time complexity being $O(n \cdot k)$, where k is the number of digits in the largest number.
- The time complexity grows linearly with respect to the input size, and the behavior observed aligns with the theoretical time complexities.
- The algorithm performs efficiently for large input sizes, especially when k, the number of digits, is relatively small.

These results confirm that Radix Sort is an efficient algorithm for sorting numbers, particularly when the range of digits is small.

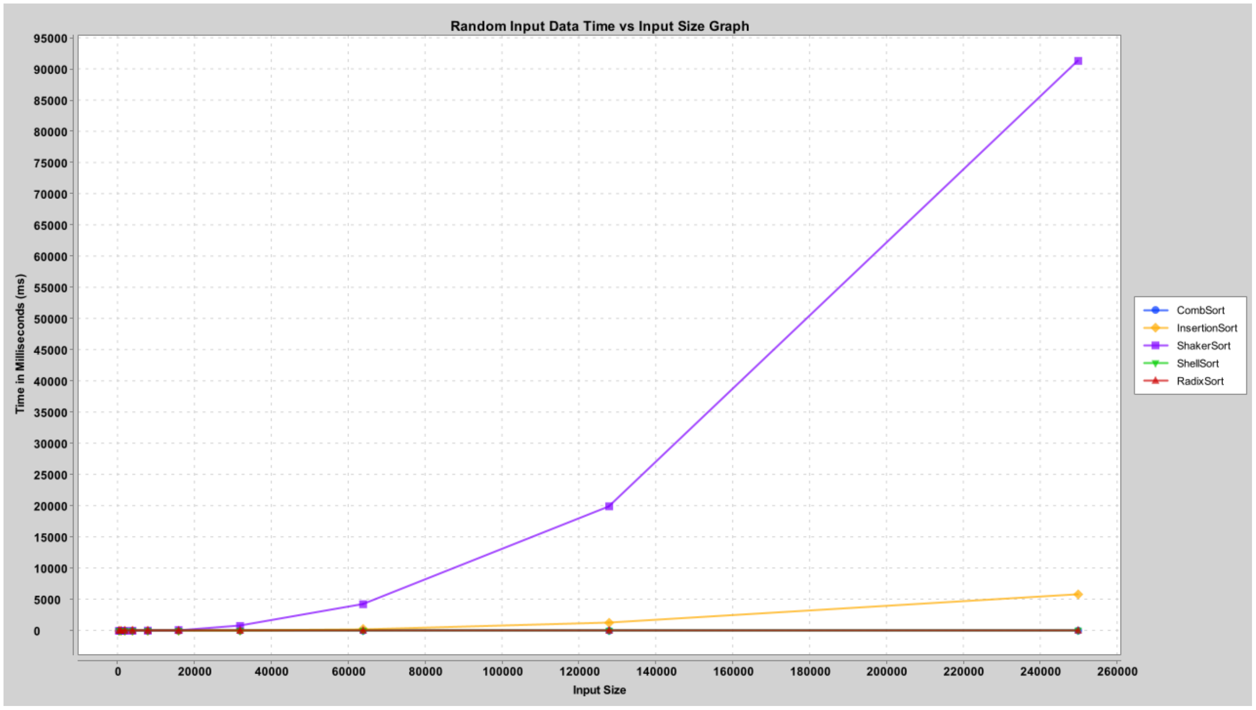

Figure 6: Random Input Data Time vs Input Size Graph.

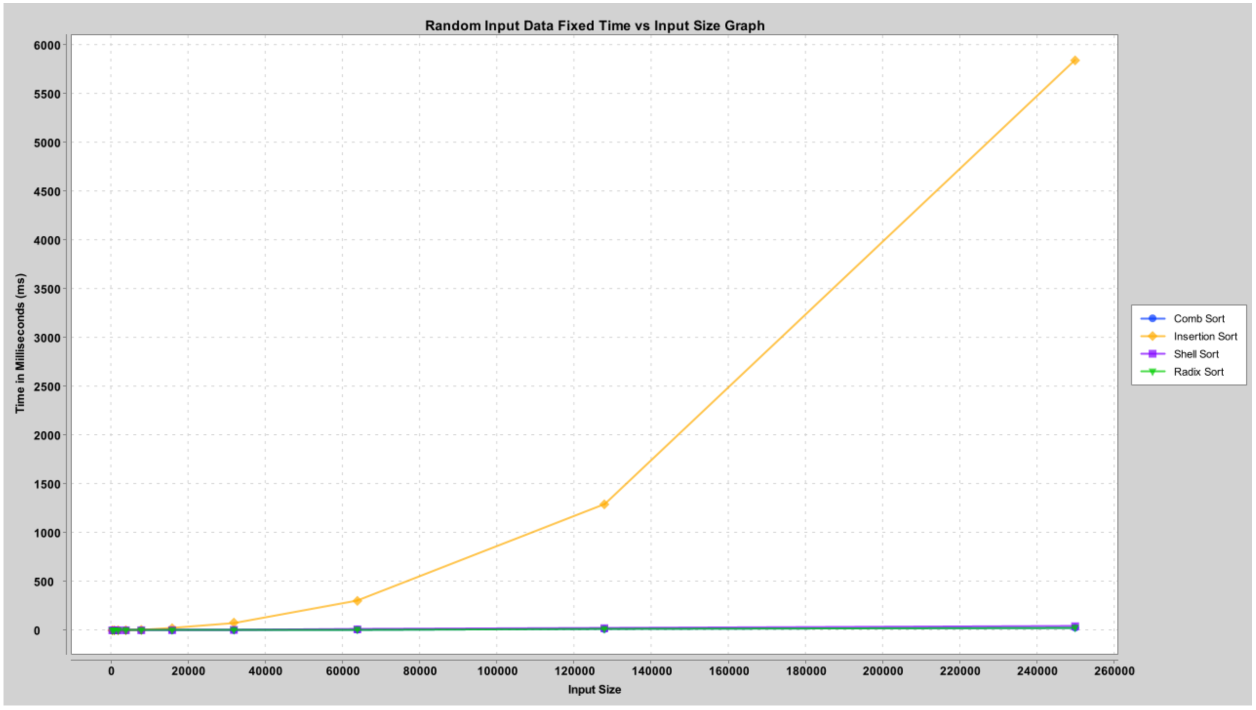

Figure 7: Random Input Data Time vs Input Size Graph Without Shaker Sort.

Random Input Analysis for Sorting Algorithms

1. Comb Sort: Comb Sort performs better than Bubble Sort by reducing the gap between compared elements gradually, which helps reduce the number of unnecessary swaps. However, it still exhibits $O(n^2)$ behavior in the worst case for random input. While the gap reduction helps in early-stage sorting, the algorithm's performance is still inefficient for large random datasets. As the input size increases, the number of comparisons grows quadratically, leading to slower performance on larger datasets.

2. Insertion Sort: Insertion Sort performs poorly on randomly ordered datasets, as its time complexity is $O(n^2)$ in the average and worst cases. The algorithm iterates through the array and compares each element with the already sorted portion of the array. Since the array is randomly ordered, each insertion takes a significant number of comparisons and shifts. In smaller or nearly sorted datasets, Insertion Sort performs better, but on random data, its performance tends to degrade, especially for large input sizes.

3. Shaker Sort: Shaker Sort, like Bubble Sort, works by iterating through the array in both directions. However, it still maintains a time complexity of $O(n^2)$ for randomly ordered datasets. While Shaker Sort slightly improves on Bubble Sort by reducing the number of passes, it still performs poorly for large datasets with random data. This is because, for random input, the number of comparisons and swaps grows quadratically with the input size, leading to inefficient performance.

4. Shell Sort: Shell Sort performs significantly better than Insertion Sort and Shaker Sort on random datasets. With its gap reduction strategy, it allows for faster sorting compared to simple quadratic algorithms. Shell Sort's time complexity can vary depending on the gap sequence used, but typically it performs at $O(n^{3/2})$ for random data. The algorithm provides a substantial performance improvement for random input, especially when compared to algorithms with $O(n^2)$ complexity.

5. Radix Sort: Radix Sort stands out from the others due to its non-comparative nature. It sorts data based on individual digits or bits, which allows it to perform in linear time, $O(nk)$, where k is the number of passes or digits in the largest number. For random input, Radix Sort performs significantly better than comparison-based algorithms, especially for large datasets. Since the algorithm's performance depends on the number of digits and not the order of the data, it remains consistently efficient across different types of random input.

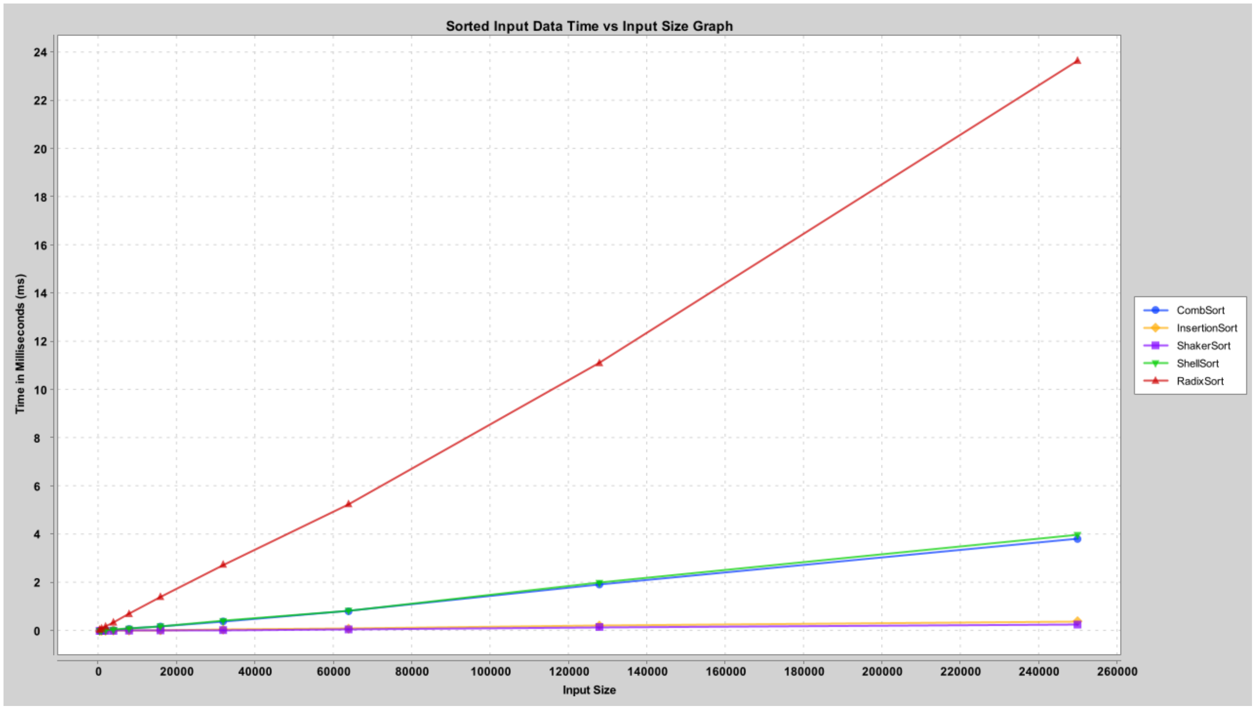

Figure 8: Sorted Input Data Time vs Input Size Graph.

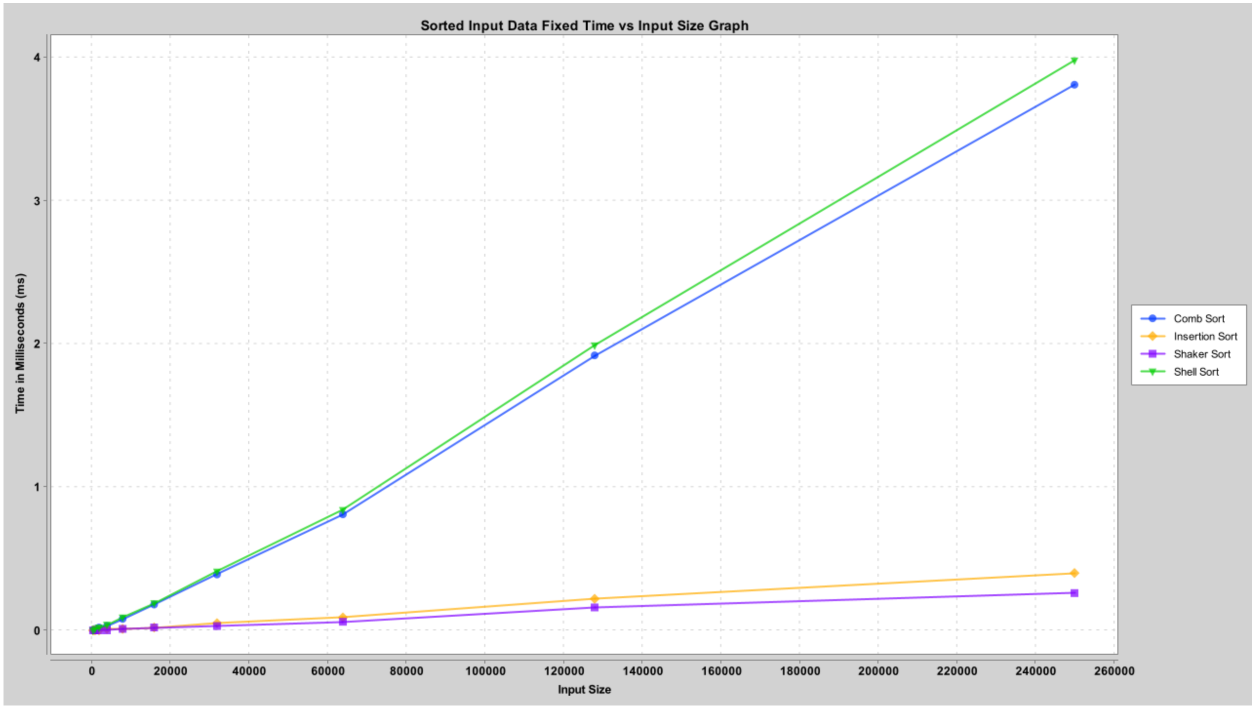

Figure 9: Sorted Input Data Time vs Input Size Graph Without Radix Sort.

Sorted Input Analysis for Sorting Algorithms

Comb Sort:

Comb Sort performs quite efficiently on sorted data. Since the algorithm skips most comparisons on a sorted array, its runtime is very fast for sorted datasets. However, depending on the increments used, the algorithm may still show an $O(n \log n)$ complexity in some cases. Generally, though, it works optimally on sorted input.

Insertion Sort:

Insertion Sort works exceptionally well on sorted data or nearly sorted data. In this case, the algorithm's time complexity tends to be $O(n)$, as only a few comparisons and shifts are required. Therefore, the performance is almost linear and efficient.

Shaker Sort:

Shaker Sort, similar to Insertion Sort, benefits from sorted input as it avoids excessive swapping. It operates with a linear time complexity ($O(n)$) when the data is already sorted.

Shell Sort:

Shell Sort performs well on sorted input, but it still shows some degree of overhead due to the gap sequence. The gap sequence affects the number of comparisons, but overall, Shell Sort performs significantly better than Insertion Sort and Shaker Sort on sorted input.

Radix Sort:

Radix Sort behaves consistently on sorted input, as its performance is independent of the order of elements. It performs efficiently in $O(nk)$, where k is the number of digits in the largest number, making it suitable for sorted data as well.

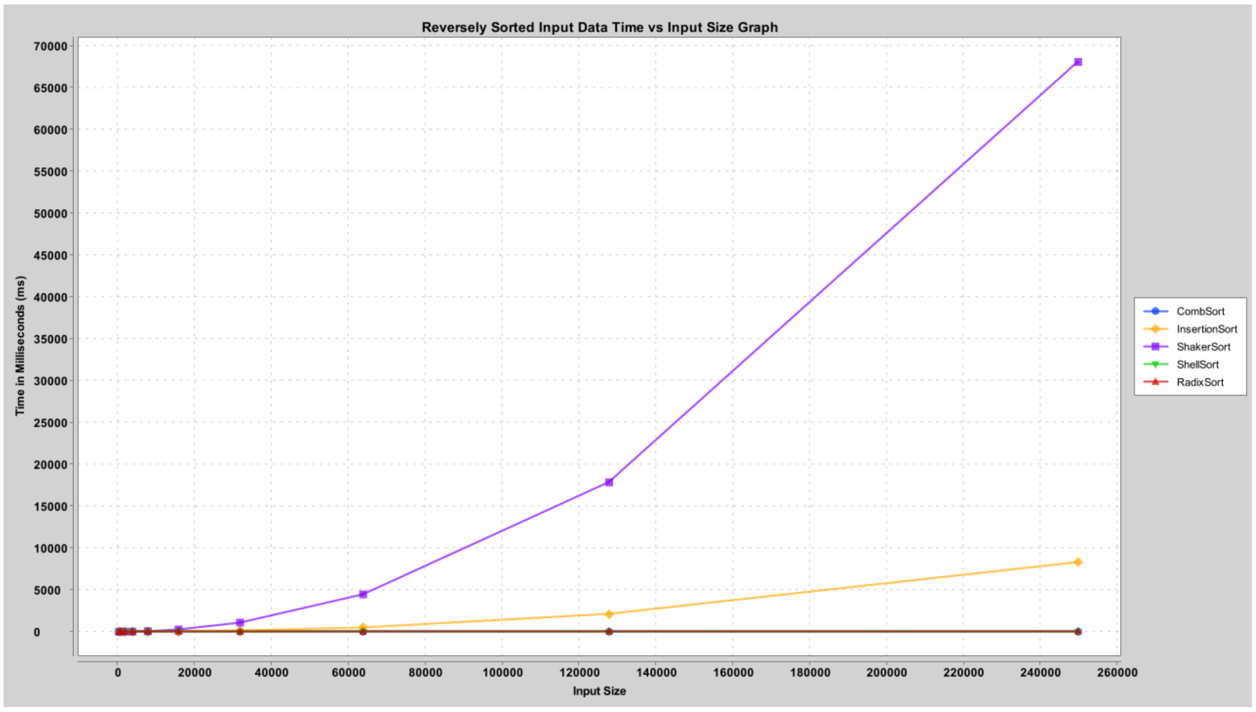

Figure 10: Reversely Sorted Input Data Time vs Input Size Graph.

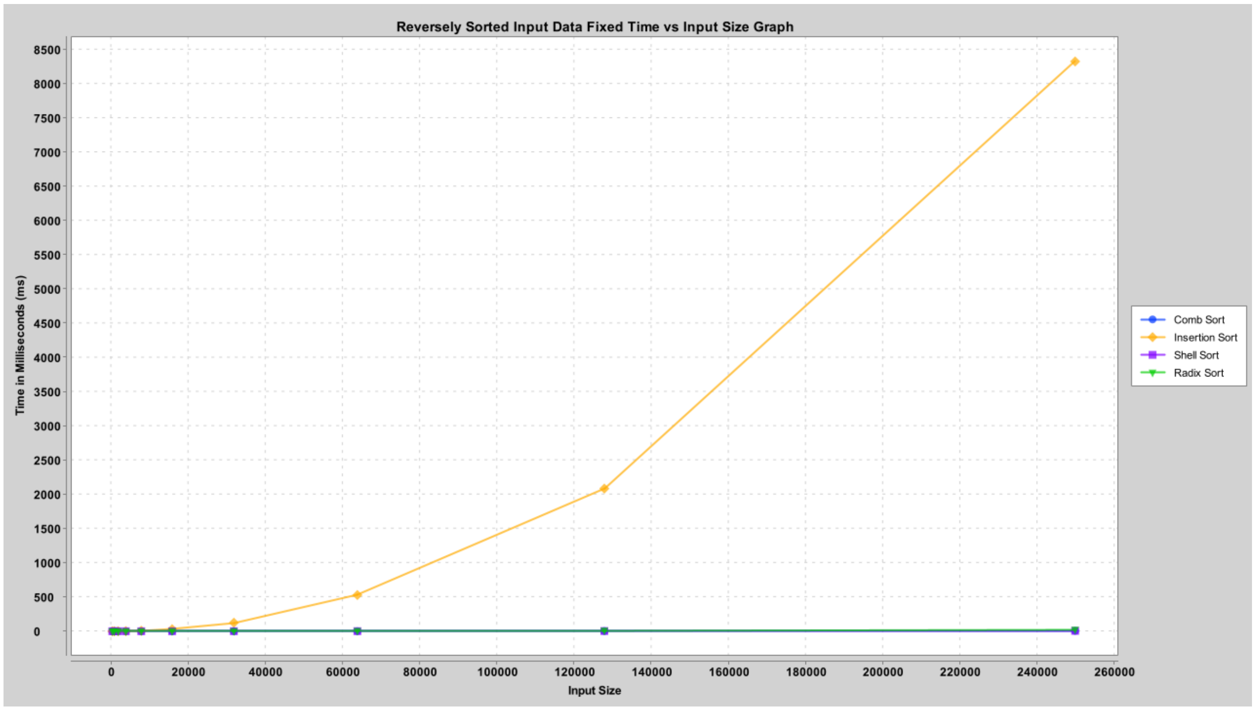

Figure 11: Reversely Sorted Input Data Time vs Input Size Graph Without Shaker Sort.

Reverse Sorted Input Analysis for Sorting Algorithms

1. Comb Sort:

Comb Sort performs better than Bubble Sort, but it still exhibits $O(n^2)$ complexity for reverse sorted data. By gradually reducing the gap between compared elements, it helps minimize unnecessary swaps, but it remains inefficient for large datasets with reverse sorted input. Since the algorithm requires more comparisons and swaps for each iteration, its performance degrades significantly on larger reverse sorted datasets.

2. Insertion Sort:

Insertion Sort performs poorly on reverse sorted data, as it has to compare each element with every other element. This leads to the maximum number of comparisons and shifts, resulting in a time complexity of $O(n^2)$. For reverse sorted data, the algorithm performs poorly on large datasets, as every insertion requires shifting all previously sorted elements, making it inefficient for large reverse sorted arrays.

3. Shaker Sort:

Shaker Sort, like Bubble Sort, iterates through the array in both directions. However, it still maintains an $O(n^2)$ complexity for reverse sorted data. Although Shaker Sort slightly improves on Bubble Sort by reducing the number of passes, it still performs inefficiently on large reverse sorted datasets. The number of comparisons and swaps grows quadratically, leading to poor performance for larger datasets with reverse sorted input.

4. Shell Sort:

Shell Sort significantly improves performance compared to Insertion Sort and Shaker Sort for reverse sorted data. Its gap reduction strategy allows for faster sorting, and it typically exhibits $O(n^{3/2})$ complexity. While the complexity can vary depending on the gap sequence, Shell Sort generally provides a substantial performance boost for reverse sorted data compared to quadratic algorithms.

5. Radix Sort:

Radix Sort is a non-comparative algorithm, which sorts data based on digits or bits. Since it is not based on comparisons, it is unaffected by the initial order of the data, including reverse sorted input. Radix Sort performs efficiently regardless of the data's initial order and generally operates with linear time complexity $O(nk)$, where k is the number of passes or digits in the largest number.

4 Notes

- The graphs presented in this study were created using XChart 3.8.8.
- Each experiment set was repeated 10 times to ensure the reliability and consistency of the results.

- The performance may vary when run on different operating systems or hardware configurations.
- The system used for the experiments was an Intel i5 12th generation processor with 16GB of RAM, an RTX 3050Ti GPU, and Windows 11 as the operating system.

References

[1] GeeksforGeeks. *Comb Sort.* Available at: `https://www.geeksforgeeks.org/comb-sort/`

[2] GeeksforGeeks. *Shell Sort.* Available at: `https://www.geeksforgeeks.org/shell-sort/`

[3] GeeksforGeeks. *Insertion Sort Algorithm.* Available at: `https://www.geeksforgeeks.org/insertion-sort-algorithm/`

[4] GeeksforGeeks. *Java Program for Cocktail Sort.* Available at: `https://www.geeksforgeeks.org/java-program-for-cocktail-sort/`

[5] GeeksforGeeks. *Radix Sort.* Available at: `https://www.geeksforgeeks.org/radix-sort/`

[6] Stack Overflow. *Time Complexity for Shell Sort.* Available at: `https://stackoverflow.com/questions/12767588/time-complexity-for-shell-sort`