Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2025 Spring

Programming Assignment 1

March 18, 2025

Student name: Erkan TAN

Student Number: b2220356098

1 Problem Definition

This project aims to analyze the performance of different sorting algorithms. We will implement **Comb Sort, Insertion Sort, Shaker Sort, Shell Sort, and Radix Sort** using Java and evaluate their efficiency.

The dataset used for sorting is **TrafficFlowDataset.csv**, where sorting will be performed based on the ”Flow Duration” column. The experiments will involve measuring execution times for different input types: **random, sorted, and reversely sorted data**.

The primary objective of this analysis is to compare the time complexity and auxiliary space complexity of each algorithm. The results will be visualized using charts to better understand the efficiency of these sorting algorithms when applied to large datasets.

2 Solution Implementation

2.1 Reading CSV File

This readCSV method reads a CSV file and extracts integer values from a specified column. It initializes a list to store the extracted values, reads the file line by line while skipping the header, splits each line into tokens, and checks if the specified column exists and is not empty. If valid, it converts the value to an integer and adds it to the list. Any invalid values are skipped to prevent errors. Finally, the list is converted into an integer array and returned.

```java
public static int[] readCSV(String filePath, int columnIndex) {
        List<Integer> data = new ArrayList<>();

        try (BufferedReader br = new BufferedReader(new FileReader(filePath)))
             {
            br.readLine();
            String line;

            while ((line = br.readLine()) != null) {
                String[] tokens = line.split(",", -1);
                if (tokens.length > columnIndex && !tokens[columnIndex].
                    isEmpty()) {
                    try {
                        data.add(Integer.parseInt(tokens[columnIndex].trim()))
                            ;
                    } catch (NumberFormatException e) {

                    }
                }
            }
        } catch (IOException e) {
            System.err.println("OOPS! Error occurred: " + e.getMessage());
        }

        return data.stream().mapToInt(i -> i).toArray();
    }
```

2.2 Comb Sort Algorithm

This method implements Comb Sort, an improvement over Bubble Sort that reduces the number of swaps by using a gap between elements. The gap starts as the array length and gradually shrinks by a factor of 10/13 in each iteration until it reaches 1. In each pass, elements at a distance of gap are compared and swapped if needed. The process continues until no swaps occur with a gap of 1, ensuring the array is fully sorted.

```java
public static void combSort(int[] arr) {
        int n = arr.length;
        int gap = n;
        boolean swapped = true;

        while (gap != 1 || swapped) {
            gap = (gap * 10) / 13;
            if (gap < 1) gap = 1;

            swapped = false;

            for (int i = 0; i < n - gap; i++) {
                if (arr[i] > arr[i + gap]) {
                    int temp = arr[i];
                    arr[i] = arr[i + gap];
                    arr[i + gap] = temp;
                    swapped = true;
                }
            }
        }
    }
```

2.3 Insertion Sort Algorithm

This method implements Insertion Sort, a simple sorting algorithm that builds the sorted array one element at a time. It starts from the second element and compares it with the previous elements, shifting them to the right if they are larger. This process continues until the correct position for the current element (key) is found. The algorithm repeats for all elements, ensuring the array becomes sorted.

```java
public static void insertionSort(int[] arr) {
        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 = i - 1;
            }
            arr[i + 1] = key;
        }
    }
```

2.4 Shaker Sort Algorithm

This method implements Shaker Sort (or Cocktail Sort), a variation of Bubble Sort that sorts in both directions. It first moves from left to right, swapping adjacent elements if they are in the wrong order. Then, it moves from right to left, performing the same swaps. This bidirectional approach helps bring both the largest and smallest elements to their correct positions faster, reducing the number of passes needed compared to Bubble Sort.

```
public static void shakerSort(int[] arr) {
        boolean swapped = true;

        while (swapped) {
            swapped = false;
            for (int i = 0; i < arr.length - 1; i++) {
                if (arr[i] > arr[i + 1]) {
                    int temp = arr[i];
                    arr[i] = arr[i + 1];
                    arr[i + 1] = temp;
                    swapped = true;
                }
            }

            if (!swapped) break;

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

2.5 Shell Sort Algorithm

This method implements Shell Sort, an optimized version of Insertion Sort that allows elements to move farther distances initially. It starts with a large gap (half the array size) and reduces it in each iteration. For each gap, elements are sorted using an insertion-like approach. This helps move elements into their correct positions faster, improving efficiency compared to standard Insertion Sort.

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

2.6 Radix Sort Algorithm

This method implements Radix Sort, a non-comparative sorting algorithm that sorts numbers digit by digit from the least significant to the most significant.

getMaxDigits(A): Determines the number of digits in the largest number. The main loop sorts the array d times, where d is the number of digits in the largest number. countingSort(A, pos): Performs Counting Sort based on the current digit position (pos). It counts occurrences of each digit, computes cumulative positions, and reconstructs the sorted array accordingly. getDigit(num, pos): Extracts the digit at a given position from a number. This ensures numbers are sorted step-by-step based on each digit, making Radix Sort efficient for large numbers.

```java
public static void radixSort(int[] A) {
        int d = getMaxDigits(A);

        for (int pos = 1; pos <= d; pos++) {
            A = countingSort(A, pos);
        }
    }


    private static int getMaxDigits(int[] A) {
        int max = Arrays.stream(A).max().orElse(0);
        return (int) Math.log10(max) + 1;
    }


    private static int[] countingSort(int[] A, int pos) {
        int size = A.length;
        int[] output = new int[size];
        int[] count = new int[10];


        for (int i = 0; i < size; i++) {
            int digit = getDigit(A[i], pos);
            count[digit]++;
        }


        for (int i = 1; i < 10; i++) {
            count[i] += count[i - 1];
        }


        for (int i = size - 1; i >= 0; i--) {
            int digit = getDigit(A[i], pos);
            output[count[digit] - 1] = A[i];
            count[digit]--;
        }

        return output;
    }


    private static int getDigit(int num, int pos) {
        return (num / (int) Math.pow(10, pos - 1)) % 10;
    }
```

3 Results, Analysis, Discussion

Insertion Sort and Shaker Sort performed well on nearly sorted datasets but were inefficient on large, random, or reversed datasets due to their quadratic time complexity. Comb Sort showed improvement over traditional Bubble Sort but still suffered in worst-case scenarios. Shell Sort offered a good balance, significantly improving over Insertion Sort, especially on larger datasets. Radix Sort outperformed the others on large datasets with integers, as it avoids direct comparisons. However, it is less effective for floating-point numbers or large integer ranges.

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

Complexity analysis tables to complete (Table 2 and Table 3):

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	0.20	0.02	0.06	0.17	0.42	0.74	1.61	3.36	7.39	15.54
Insertion sort	0.34	0.33	0.50	1.02	4.08	27.38	55.48	210.40	891.40	3789.43
Shaker sort	0.64	0.84	1.17	4.26	15.17	72.89	233.02	1465.74	8657.78	40201.22
Shell sort	0.21	0.19	0.11	0.23	0.52	1.25	2.49	5.50	11.86	23.07
Radix sort	0.47	0.75	0.82	1.06	2.17	4.81	9.08	16.36	33.15	69.14
	Sorted Input Data Timing Results in ms									
Comb sort	0.03	0.01	0.02	0.02	0.05	0.44	0.26	0.57	1.30	3.40
Insertion sort	0.01	0.00	0.00	0.01	0.01	0.02	0.06	0.07	0.14	0.73
Shaker sort	0.00	0.05	0.00	0.00	0.01	0.02	0.04	0.05	0.10	0.32
Shell sort	0.03	0.11	0.02	0.03	0.07	0.15	0.30	0.71	1.42	3.68
Radix Sort	0.21	0.80	0.67	1.05	2.31	4.18	8.09	16.69	33.32	64.39
	Reversely Sorted Input Data Timing Results in ms									
Comb sort	0.11	0.02	0.02	0.04	0.09	0.20	0.42	0.79	1.76	3.71
Insertion sort	0.24	0.26	0.77	1.95	7.45	25.53	101.09	400.75	1604.82	6619.18
Shaker sort	0.82	0.61	2.18	6.92	26.44	103.92	417.19	1696.16	6803.93	26499.17
Shell sort	0.31	0.02	0.03	0.05	0.11	0.27	0.53	1.01	2.16	4.29
Radix sort	0.31	0.47	0.60	1.20	2.38	4.09	8.31	16.98	33.11	64.27

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	$\Omega(nlogn)$	$\Theta(n^{3/2})$	$O(n^2)$
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	$O(n+k)$

Example how to include and reference a figure: Fig. 1.

3.1 Charts of Average Running Time by Datas

3.1.1 Random Data Analysis

This graph shows the execution time (in milliseconds) of different sorting algorithms on random data as the input size increases.

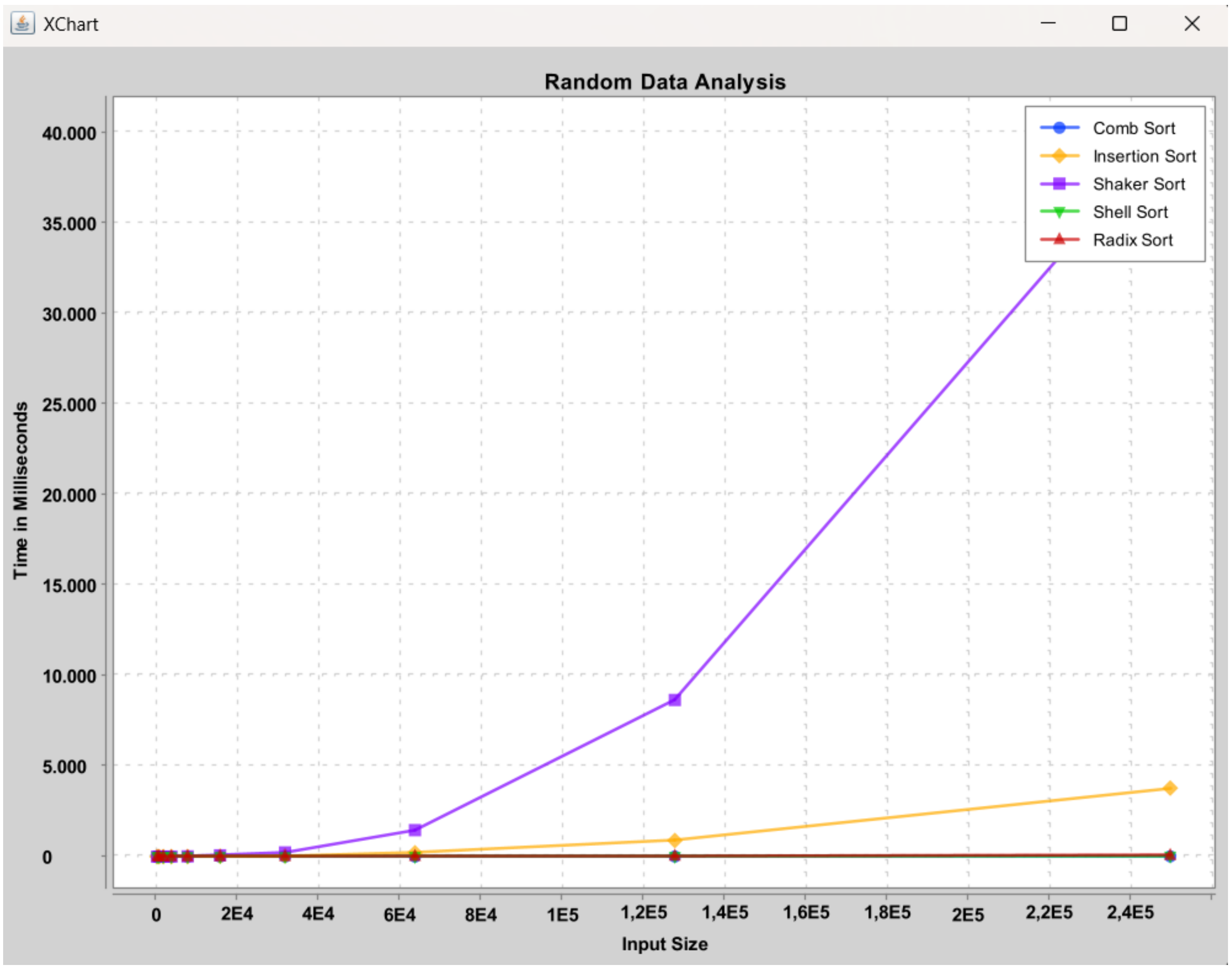

Figure 1: Plot of the functions.

Shaker Sort (Purple) performs the worst, with execution time increasing exponentially. Insertion Sort (Yellow) also slows down with larger inputs but is better than Shaker Sort. Comb Sort (Blue), Shell Sort (Green), and Radix Sort (Red) perform significantly better, maintaining low execution times even for large inputs. Radix Sort shows the best performance, with almost no noticeable increase in execution time. Overall, Shaker and Insertion Sort are inefficient for large datasets, while Radix, Shell, and Comb Sort scale much better. Fig. 1.

3.1.2 Sorted Data Analysis

This graph shows the execution time of sorting algorithms on already sorted data as the input size increases. Radix Sort (Red) performs the worst, with execution time increasing significantly. This suggests that Radix Sort does not take advantage of the already sorted order. Insertion Sort (Yellow), Shaker Sort (Purple), and Comb Sort (Blue) perform very well, maintaining extremely low execution times. This is expected because Insertion Sort and Shaker Sort run in nearly O(n) time on sorted data. Shell Sort (Green) also performs well but slightly worse than Insertion and Shaker Sort.

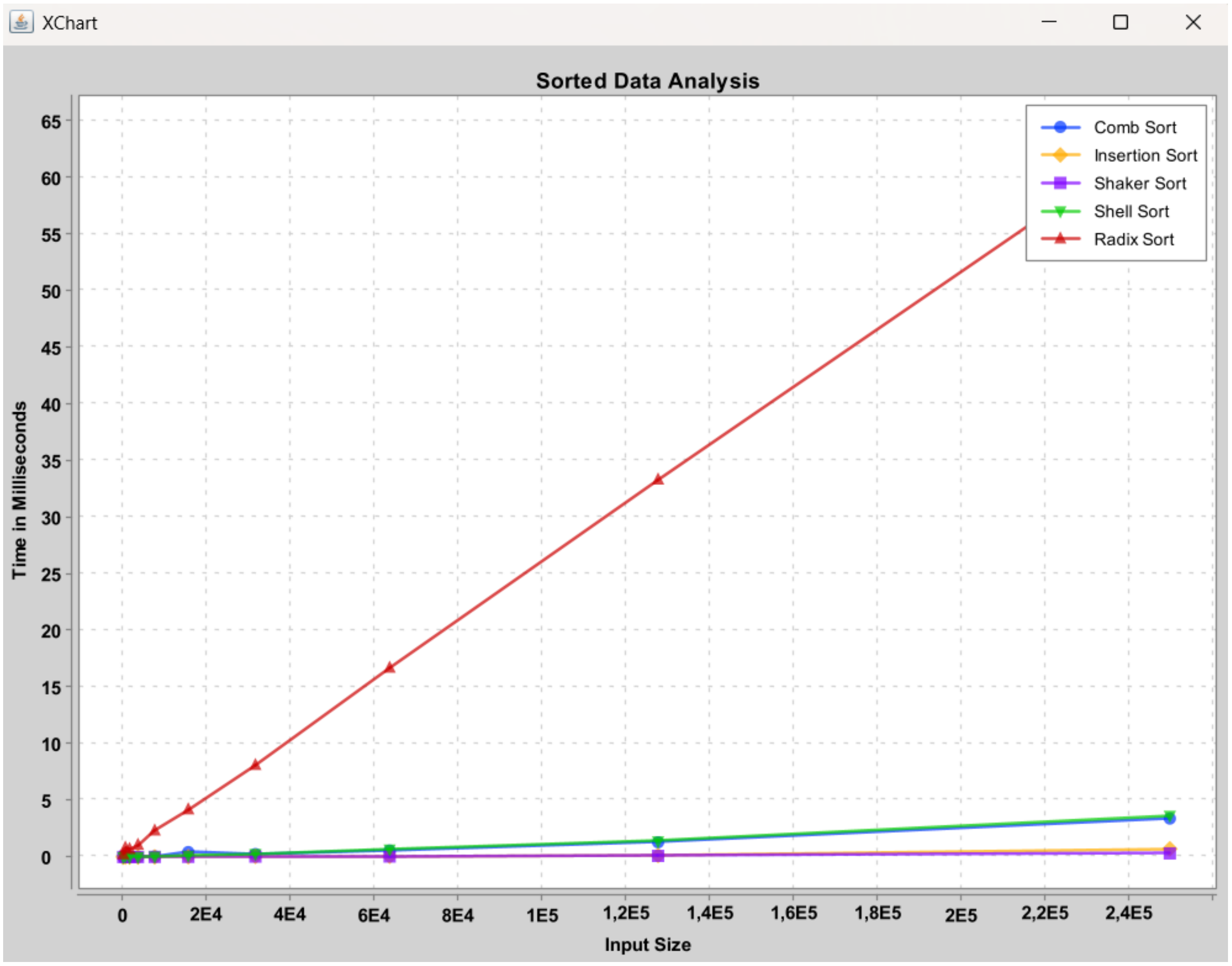

Figure 2: A smaller plot of the functions.

3.1.3 Reversed Sorted Data Analysis

This graph shows the execution time of sorting algorithms on reversely sorted data as the input size increases. Shaker Sort (Purple) performs the worst, showing a sharp increase in execution time. This is expected because Shaker Sort works similarly to Bubble Sort, which has a worst-case complexity of O(n^2) when the data is in reverse order. Insertion Sort (Yellow) also struggles, though it performs better than Shaker Sort. Insertion Sort has a worst-case complexity of O(n^2) for reversed data, requiring many swaps to sort the elements. Comb Sort (Blue), Shell Sort (Green), and Radix Sort (Red) perform significantly better, maintaining near-linear execution times. Radix Sort (Red) is the best performer, as expected, because it is not affected by data order, making it an efficient choice for sorting large datasets.

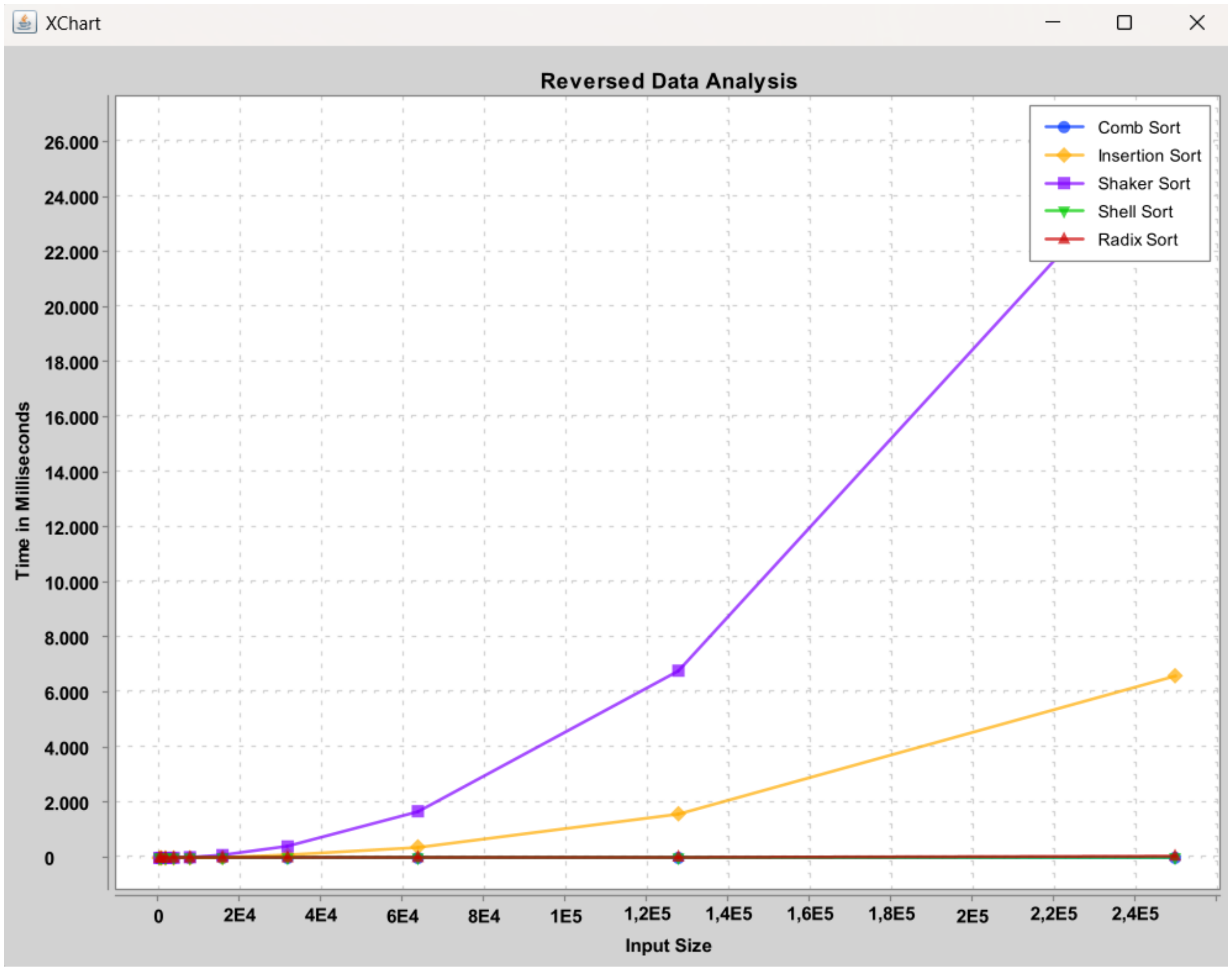

Figure 3: A smaller plot of the functions.

3.1.4 Comb Sort Analysis

Random data (blue line) exhibits the highest execution time, increasing significantly as the input size grows. This aligns with expectations since Comb Sort struggles more with unsorted data. Sorted data (orange line) has the lowest execution time, as Comb Sort performs efficiently when the input is nearly sorted. Reversed data (purple line) follows a similar trend to the sorted data but takes slightly longer, which is expected since reversed order requires more swaps. Overall, the graph confirms that Comb Sort handles sorted and reversed inputs efficiently but performs poorly on random data, demonstrating its efficiency characteristics.

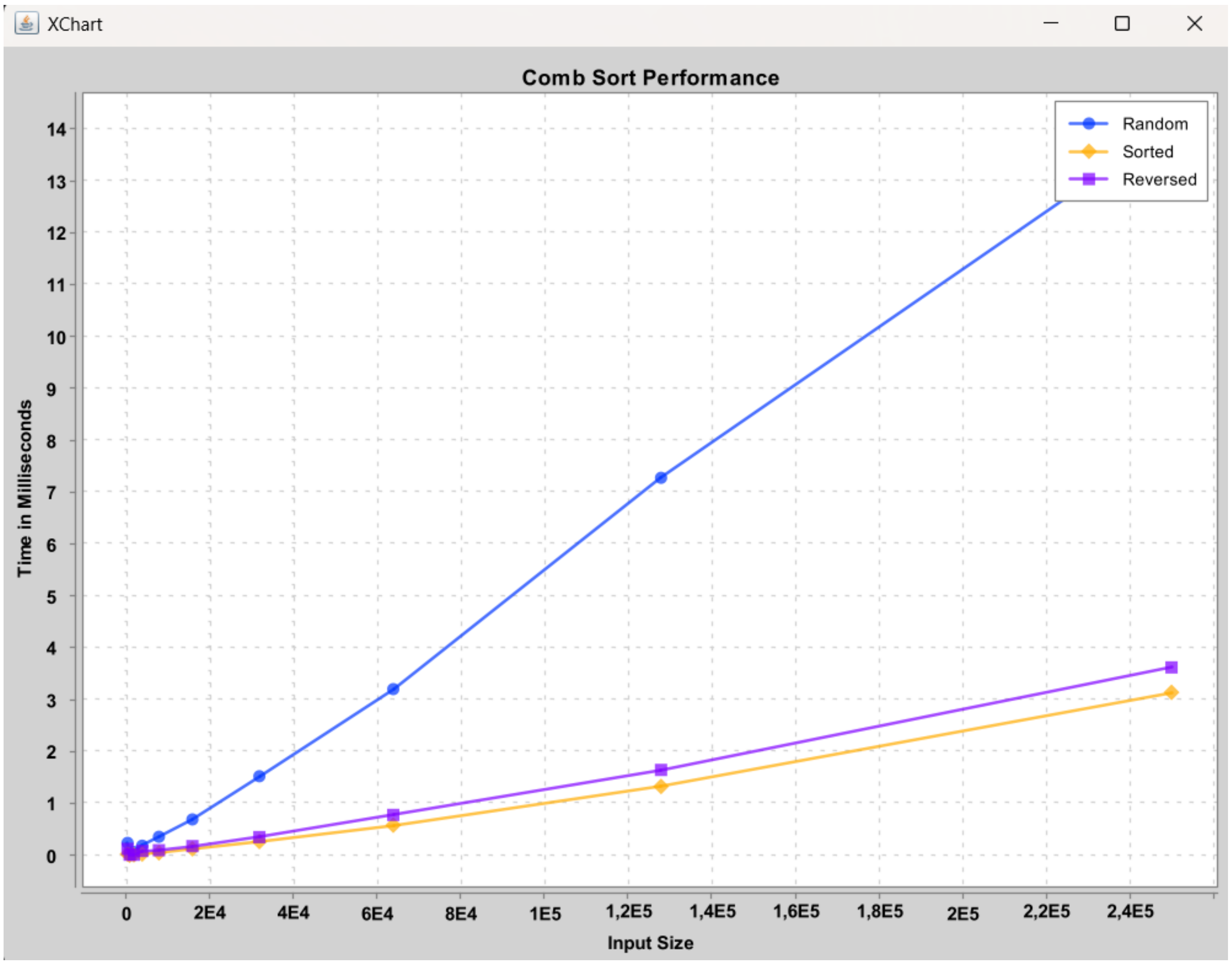

Figure 4: A smaller plot of the functions.

3.1.5 Insertion Sort Analysis

Sorted data (orange line) has a near-zero execution time, indicating that Insertion Sort performs exceptionally well on already sorted input, running in O(n) time complexity. Random data (blue line) shows a steady increase in execution time as the input size grows, reflecting its average O(n^2) complexity. Reversed data (purple line) takes the longest time, significantly increasing with input size. This is expected since Insertion Sort performs worst when the input is in reverse order, requiring the maximum number of shifts. Overall, the graph confirms that Insertion Sort is highly efficient for nearly sorted data but performs poorly on random and reversed data due to its quadratic time complexity.

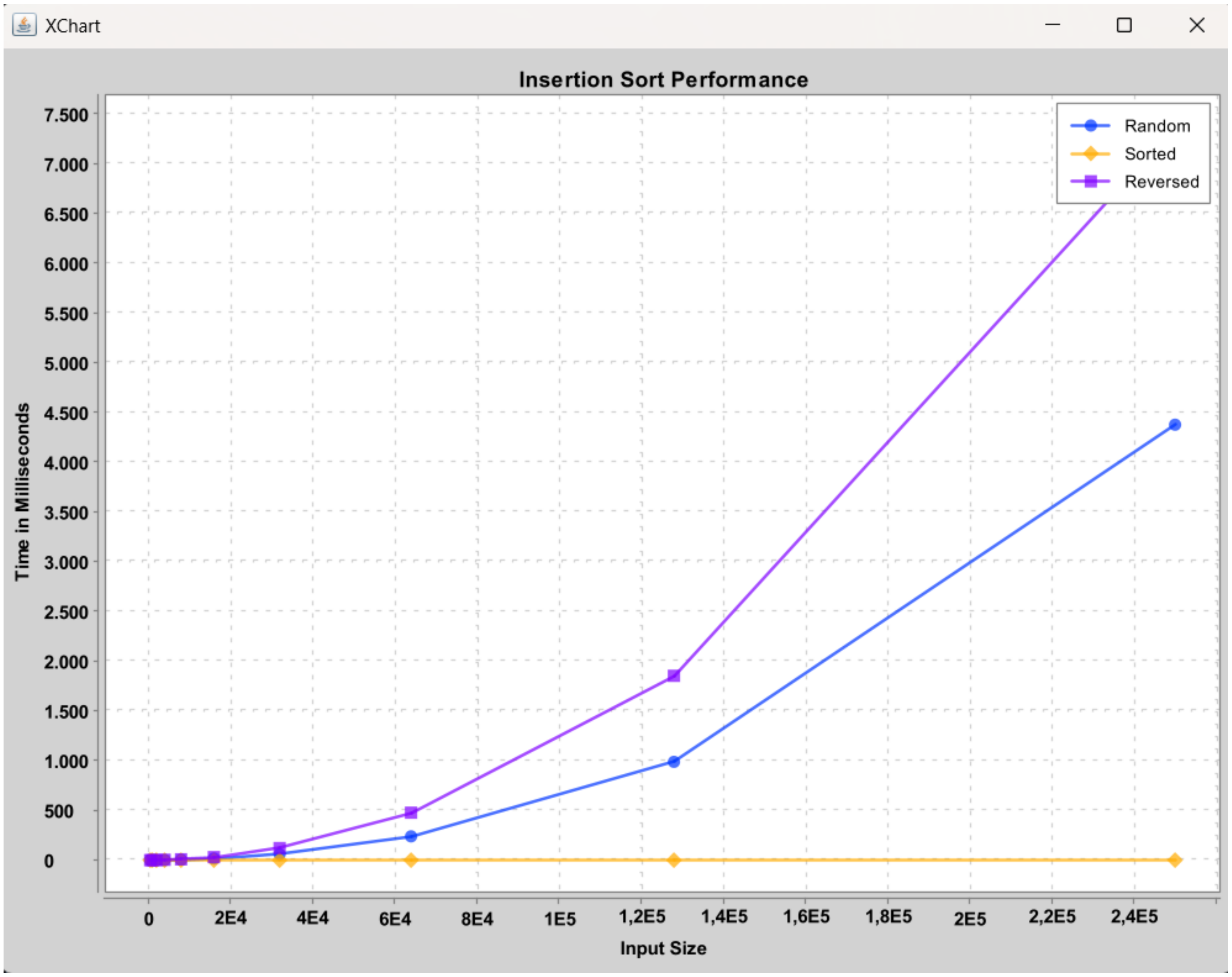

Figure 5: A smaller plot of the functions.

3.1.6 Shaker Sort Analysis

Sorted data (orange line) has an almost negligible execution time because Shaker Sort detects the sorted order and runs in approximately O(n) time. Random data (blue line) follows a typical O(n^2) growth, as the algorithm performs both forward and backward passes to sort the elements. Reversed data (purple line) initially behaves similarly to random data but performs slightly better at larger input sizes. This is because the bidirectional nature of Shaker Sort helps correct the reversed order more efficiently than a standard Bubble Sort. In conclusion, Shaker Sort is highly efficient for already sorted data but still suffers from O(n^2) complexity for random and reversed data.

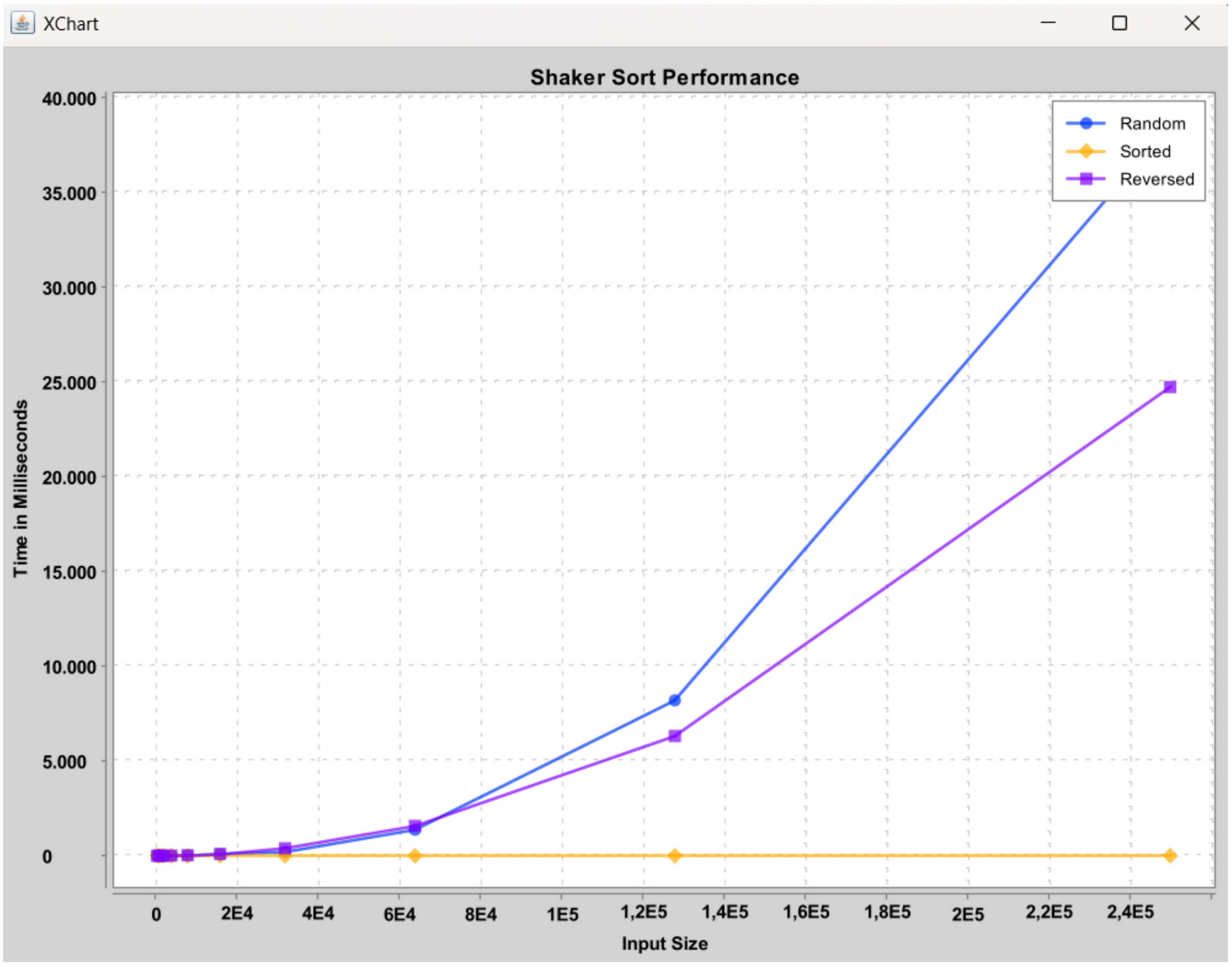

Figure 6: A smaller plot of the functions.

3.1.7 Shell Sort Analysis

Sorted data (orange line) performs the best, showing minimal execution time. Since Shell Sort is an improved version of Insertion Sort, it efficiently detects sorted input and runs in nearly O(n) time. Reversed data (purple line) performs worse than sorted but better than random data. The algorithm benefits from reducing large gaps early in the process, leading to faster corrections. Random data (blue line) has the highest execution time, following a near O(n log n) complexity.

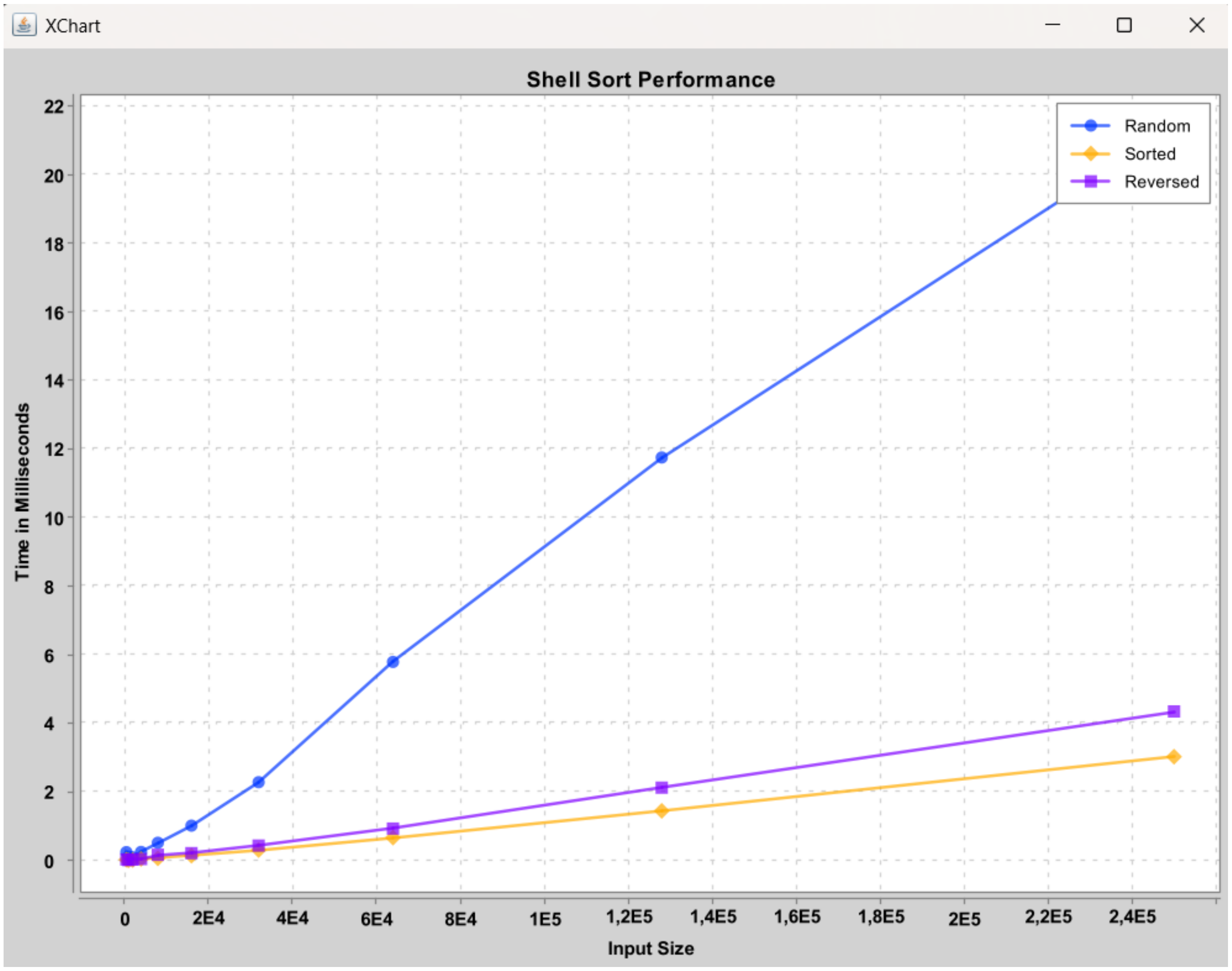

Figure 7: A smaller plot of the functions.

3.1.8 Radix Sort Analysis

Radix Sort performs consistently well regardless of whether the input is random, sorted, or reversed. The algorithm's linear time complexity (O(nk)) makes it very efficient for sorting large numbers. Unlike Shell Sort or Insertion Sort, Radix Sort does not benefit from already sorted data, as seen in the nearly identical performance across all cases.

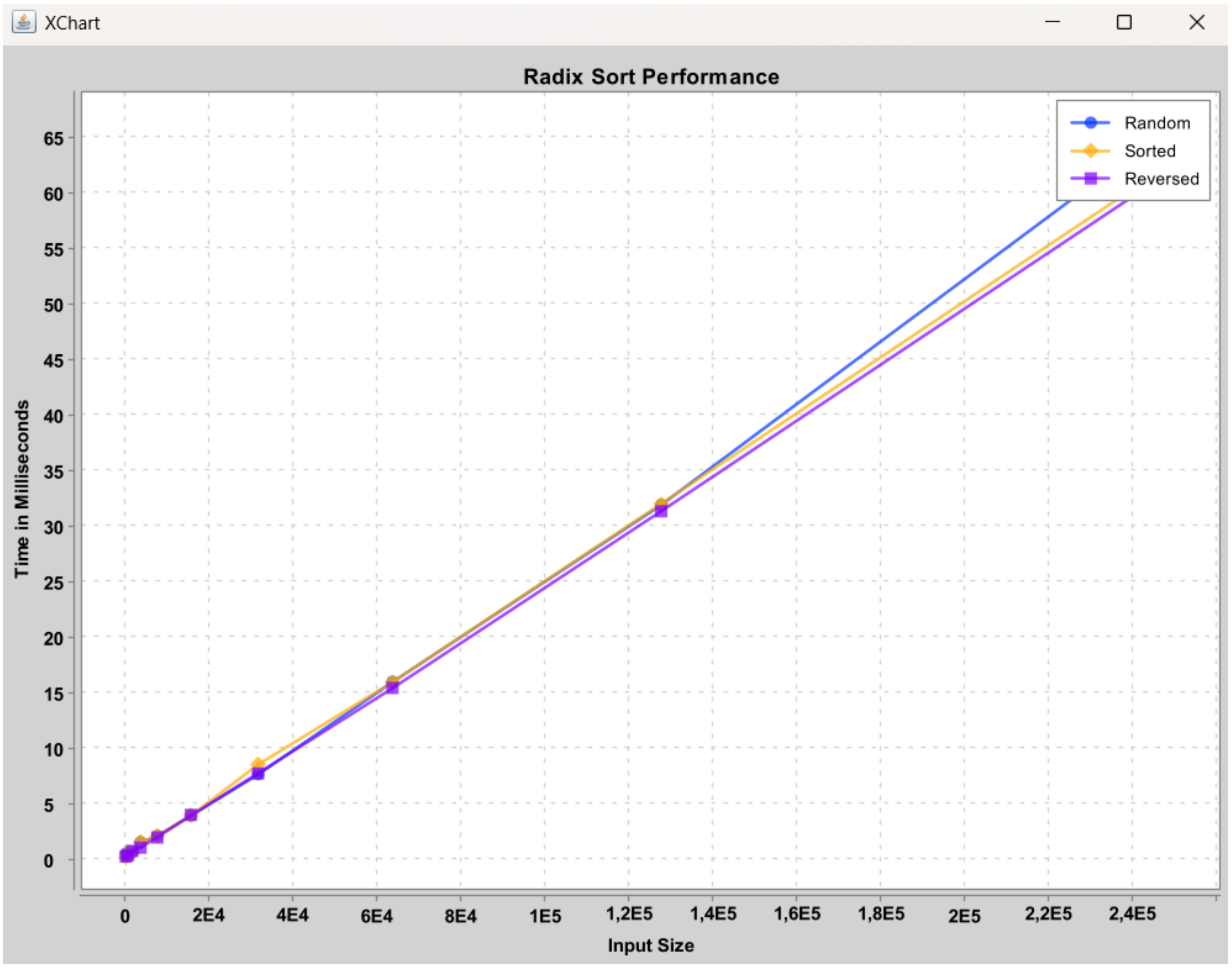

Figure 8: A smaller plot of the functions.

4 Notes

In some sorting algorithms, randomly ordered data is slower, while in others, reversely sorted data takes longer to process. In this assignment, we compared these cases to reach our conclusions.

References

- https://www.geeksforgeeks.org/radix-sort/
- https://en.wikipedia.org/wiki/Sorting_algorithm
- https://www.geeksforgeeks.org/reading-csv-file-java-using-opencsv/