Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 20, 2025

Student name:
Ahmet Emre Usta

Student Number:
b2200765036

1 Problem Definition

The assignment involves implementing and analyzing the performance of various sorting algorithms to determine their execution time across different input sizes and conditions. This includes evaluating performance on datasets that are randomly ordered, sorted, and reverse sorted. The overarching goal is to empirically validate and confirm the theoretical time complexities associated with each algorithm.

2 Solution Implementation

The following sections detail the implementation of the sorting algorithms, which are written in Java and follow the standard algorithmic approach as outlined in the computer science literature.

2.1 Insertion Sort

The implementation of the Insertion Sort algorithm is as follows:

```java
// InsertionSort.java
public class InsertionSort {

    // Optimized insertion sort method.
    public int[] sort(int[] A) {
        int n = A.length;
        // Directly modify the input array to avoid unnecessary cloning,
        // which reduces memory usage.
        for (int i = 1; i < n; i++) {
            int key = A[i];
            int j = i - 1;

            // Move elements of A[0..i-1], that are greater than key,
            // to one position ahead of their current position.
            while (j >= 0 && A[j] > key) {
                A[j + 1] = A[j];
                j = j - 1;
            }
            A[j + 1] = key;
        }
        return A; // Return the sorted array.
    }
}
```

2.2 Comb Sort

The Comb Sort algorithm is implemented as shown:

```java
// CombSort.java
public class CombSort {

    public static void sort(int[] array) {
        int gap = array.length;
        double shrink = 1.3;
        boolean sorted = false;

        while (!sorted) {
            gap = Math.max(1, (int) Math.floor(gap / shrink));
            sorted = (gap == 1);

            for (int i = 0; i + gap < array.length; i++) {
                if (array[i] > array[i + gap]) {
                    // Swap elements
                    int temp = array[i];
                    array[i] = array[i + gap];
                    array[i + gap] = temp;

                    sorted = false;
                }
            }
        }
    }
}
```

2.3 Radix Sort

Radix Sort algorithm implementation is as follows:

```java
// RadixSort.java
public class RadixSort {

    public static int[] sort(int[] array) {
        int max = findMax(array);
        int d = (int) Math.log10(max) + 1;  // Find the number of digits of
           the largest number

        for (int pos = 1; pos <= d; pos++) {
            int[] count = new int[10]; // Assuming decimal digits (0-9)
            int[] output = new int[array.length];
            int size = array.length;

            // Count occurrences of each digit
            for (int i = 0; i < size; i++) {
                int digit = getDigit(array[i], pos);
                count[digit]++;
            }

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

            // Place elements in sorted order
            for (int i = size - 1; i >= 0; i--) {
                int digit = getDigit(array[i], pos);
                count[digit]--;
                output[count[digit]] = array[i];
            }

            // Copy the sorted elements back into the original array
            System.arraycopy(output, 0, array, 0, size);
        }
        return array;  // Return the sorted array
    }

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

    private static int findMax(int[] array) {
        int max = array[0];
        for (int num : array) {
            if (num > max) {
```

```
45                      max = num;
46                  }
47              }
48              return max;
49          }
50  }
```

2.4 Shaker Sort

The Shaker Sort method is shown below:

```java
// ShakerSort.java
public class ShakerSort {

    public static void sort(int[] array) {
        boolean swapped = true;
        while (swapped) {
            swapped = false;
            for (int i = 0; i <= array.length - 2; i++) {
                if (array[i] > array[i + 1]) {
                    // Swap elements
                    int temp = array[i];
                    array[i] = array[i + 1];
                    array[i + 1] = temp;
                    swapped = true;
                }
            }
            if (!swapped) {
                break;
            }
            swapped = false;
            for (int i = array.length - 2; i >= 0; i--) {
                if (array[i] > array[i + 1]) {
                    // Swap elements
                    int temp = array[i];
                    array[i] = array[i + 1];
                    array[i + 1] = temp;
                    swapped = true;
                }
            }
        }
    }
}
```

2.5 Shell Sort

The Shell Sort algorithm is given by:

```java
// ShellSort.java
public class ShellSort {

    public static void sort(int[] array) {
        int n = array.length;
        int gap = n / 2;

        while (gap > 0) {
            for (int i = gap; i < n; i++) {
                int temp = array[i];
                int j = i;
                while (j >= gap && array[j - gap] > temp) {
                    array[j] = array[j - gap];
                    j -= gap;
                }
                array[j] = temp;
            }
            gap /= 2;
        }
    }
}
```

3 Results, Analysis, Discussion

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									
Insertion sort	0.153	0.04	0.18	0.70	2.71	42.4	168.23	714.1	714.10	3045.93
Comb sort	0.11	0.051	0.05	0.15	0.34	1.43	3.07	6.45	10.050	13.27
Radix sort	0.42	0.51	0.81	1.65	3.34	6.68	13.3	26.18	52.80	102.65
Shell sort	0.13	0.07	0.09	0.23	0.52	1.10	2.51	5.62	12.56	23.52
Shaker sort	0.34	0.56	1.17	4.75	18.85	85.1	306.9	1172.077	4785.66	33377.2
	Sorted Input Data Timing Results in ms									
Insertion sort	0.01	0.04	0.17	0.70	2.68	11.63	42.37	167.72	716.47	3040.07
Comb sort	0.005	0.019	0.05	0.148	0.34	0.67	1.42	3.08	6.44	13.24
Radix sort	0.20	0.41	0.81	1.676	67.692	3.33	6.64	13.18	26.50	102.96
Shell sort	0.009	0.03	0.093	0.23	67.295	0.51	1.10	2.47	5.62	23.36
Shaker sort	0.067	0.32	1.16	5.12	67.295	19.07	85.99	309.59	1178.10	33570.47
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.012	0.04	0.16	0.65	2.57	9.24	39.85	169.9	680.04	2368.09
Comb sort	0.005	0.018	0.041	0.15	0.34	0.67	1.42	3.06	6.42	13.31
Radix sort	0.209	0.41	0.82	0.82	1.65	3.35	6.62	13.27	26.56	52.95
Shell sort	0.009	0.027	0.08	67.601	0.23	0.52	1.13	2.48	5.53	12.72
Shaker sort	0.09	0.28	1.10	67.601	4.48	17.92	64.39	290.54	1155.32	4553.91

4 Results, Analysis, Discussion

Table 1 presents the running time test results for a set of sorting algorithms over various input sizes. The timing results, measured in milliseconds (ms), provide a detailed view of each algorithm's scalability with the input size.

For **Insertion Sort**, the running time grows quadratically, as expected with its $O(n^2)$ complexity. For smaller input sizes (e.g., $n = 500$), the algorithm performs efficiently, but as the input size increases, the running time becomes significantly larger. For example, at $n = 250,000$, the running time jumps to 3045.93 ms for random data, highlighting its inefficiency for larger datasets.

Comb Sort demonstrates much better performance than Insertion Sort, especially for larger datasets. The time increases linearly with input size, consistent with its $O(n \log n)$ complexity. Its growth is far more controlled, and even for larger inputs, such as $n = 250,000$, the time remains manageable (13.27 ms). For sorted and reversely sorted data, Comb Sort also performs efficiently, indicating robustness to input order.

Radix Sort shows an increasing but relatively steady time complexity as input size grows, with times ranging from 0.42 ms (for $n = 500$) to 102.65 ms (for $n = 250,000$) on random data. This behavior is consistent with its $O(nk)$ complexity, where k is the range of digits or elements in the input. Radix Sort's performance for sorted and reversely sorted data remains relatively stable, though some variation is observed for large input sizes.

Shell Sort has a mixed performance, with times similar to Comb Sort but with a more erratic

growth pattern. On random data, its performance ranges from 0.13 ms (for $n = 500$) to 23.52 ms (for $n = 250,000$). Its performance for sorted and reversely sorted data shows some variation, especially for the larger sizes, but it remains efficient compared to Insertion Sort.

Shaker Sort exhibits poor scalability, especially with larger input sizes. The algorithm's time increases sharply with larger datasets, reaching 33,377.2 ms for random data at $n = 250,000$. It shows a similar trend for sorted and reversely sorted data, with times of 33,570.47 ms for random and 4,553.91 ms for reversely sorted data at $n = 250,000$. This suggests that Shaker Sort is inefficient for larger datasets, regardless of the input order.

4.1 Best, Average, and Worst Cases for the Given Algorithms

- Insertion Sort : - Best Case: $O(n)$ (when the input is already sorted or nearly sorted). - Average Case: $O(n^2)$ (when the input is in random order). - Worst Case: $O(n^2)$ (when the input is in reverse order, requiring the maximum number of comparisons and swaps).

- Comb Sort : - Best Case: $O(n \log n)$ (when the input is nearly sorted or small).

- Average Case: $O(n \log n)$ (typical behavior for random data).

- Worst Case: $O(n^2)$ (in the unlikely case of a very unsorted input, although it is much less likely compared to Bubble Sort).

- Radix Sort :

- Best Case: $O(nk)$ (when the range of input values k is small).

- Average Case: $O(nk)$ (typical behavior depending on the input size and digit range).

- Worst Case: $O(nk)$ (still dependent on the range k, but more efficient than comparison-based sorting algorithms for large inputs).

- Shell Sort :

- Best Case: $O(n \log n)$ (when the input is nearly sorted).

- Average Case: $O(n^{1.5})$ (depending on the gap sequence used).

- Worst Case: $O(n^2)$ (with poor gap sequences, especially for large, unsorted inputs).

- Shaker Sort :

- Best Case: $O(n)$ (when the input is already sorted or nearly sorted).

- Average Case: $O(n^2)$ (when the input is random).

- Worst Case: $O(n^2)$ (when the input is in reverse order, similar to Insertion Sort).

4.2 Do the Obtained Results Match Their Theoretical Asymptotic Complexities?

The obtained results largely align with the theoretical complexities:

- Insertion Sort : The quadratic growth in running time for larger inputs matches its $O(n^2)$ complexity.

- Comb Sort : The results show that its time grows much more linearly than Insertion Sort, which is consistent with its $O(n \log n)$ average-case complexity.

- Radix Sort : The performance is consistent with $O(nk)$ complexity, with running times increasing relatively steadily as input size grows.

- Shell Sort : The results align with the $O(n^{1.5})$ average-case and $O(n^2)$ worst-case behavior, though there is some variation due to the specific gap sequences used.

- Shaker Sort : The results confirm its $O(n^2)$ worst-case complexity, with an increase in time for larger input sizes, similar to Insertion Sort.

4.3 Do Shaker Sort and Comb Sort Improve Performance Compared to Bubble Sort?

Yes, both Shaker Sort and Comb Sort improve performance compared to Bubble Sort .

- Shaker Sort : It improves by performing the sorting bidirectionally, effectively reducing the number of passes required to move larger or smaller elements to their correct positions.

- Comb Sort : It improves on Bubble Sort by eliminating small gaps early in the sorting process, reducing the number of comparisons needed and thus speeding up the sorting of larger datasets.

Both algorithms reduce the overhead of redundant comparisons, making them more efficient than Bubble Sort, which only moves one element at a time in a single direction.

4.4 Is Shell Sort Better Than Insertion Sort for Larger Datasets? Under What Conditions Does Insertion Sort Still Perform Well?

- Shell Sort is generally better than Insertion Sort for larger datasets because of its improved time complexity, especially when using efficient gap sequences. It performs at an average time complexity of $O(n^{1.5})$ to $O(n^2)$, which is typically faster than the $O(n^2)$ time of Insertion Sort for larger inputs.

- Insertion Sort still performs well in cases where the input data is small or nearly sorted, as its best case is $O(n)$. For these situations, Insertion Sort is highly efficient due to its simplicity and low overhead.

For example, small datasets (e.g., $n = 500$) and nearly sorted data show favorable results for Insertion Sort.

4.5 How Does Radix Sort Handle Large Numerical Ranges Efficiently?

Radix Sort is efficient with large numerical ranges due to its non-comparison-based nature. Instead of directly comparing elements, Radix Sort processes individual digits of the numbers, using counting sort or bucket sorting for each digit. This enables it to handle large datasets efficiently when the range of numbers is not excessively large. The complexity $O(nk)$, where k is the number of digits, allows Radix Sort to outperform comparison-based algorithms like Merge Sort or Quick Sort in cases where the range of numbers k is reasonably constrained, making it suitable for large numerical datasets with smaller digit ranges.

In conclusion, these results emphasize the importance of choosing the appropriate sorting algorithm based on both dataset size and initial order. **Comb Sort** and **Radix Sort** provide more consistent and scalable solutions, especially for larger datasets. **Insertion Sort** excels with small, nearly sorted data but becomes impractical for larger inputs. **Shaker Sort** should be avoided for large datasets, given its poor performance.

5 In-depth Complexity Analysis of Algorithmic Performance

This section provides a comprehensive analysis of both the computational and auxiliary space complexities for a curated set of algorithms, offering insights into their efficiency and resource requirements under various scenarios.

Table 2: Computational complexity comparison of the given algorithms.

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Comb Sort	$\Omega(n \log n)$	$\Theta(n^2)$	$O(n^2)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$
Shell Sort	$\Omega(n \log n)$	$\Theta(n^{3/2})$	$O(n^2)$
Shaker Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$

This table provides a comparative analysis of the computational complexities for various sorting algorithms under different scenarios. Insertion Sort performs well in the best-case scenario with a linear complexity of $\Omega(n)$, but its performance degrades to $O(n^2)$ in the worst case, making it inefficient for larger datasets. Comb Sort improves upon Insertion Sort but still suffers from quadratic complexity in the average and worst cases. Radix Sort, though dependent on the range of input values, offers linear time complexity under certain conditions, making it efficient for specific datasets. Shell Sort, while faster than Insertion Sort in practice, still experiences quadratic time complexity in the worst case. Shaker Sort, like Insertion Sort, has quadratic complexity and performs similarly in all cases. Understanding these complexities helps in selecting the most suitable algorithm for a given dataset and problem.

Table 3: Auxiliary space complexity of the given algorithms.

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

The auxiliary space complexity table illustrates the additional memory usage for various sorting algorithms, independent of the input data size. Insertion Sort, Comb Sort, Shell Sort, and Shaker Sort are all highly memory-efficient, with an auxiliary space complexity of $O(1)$, indicating constant space requirements that do not scale with input size. Radix Sort, on the other hand, requires $O(n + k)$ space due to its dependence on the range of input values (k) and the number of elements (n), as it uses extra memory for counting or sorting individual digits. Understanding these space complexities is critical for choosing algorithms when memory constraints are a significant factor, highlighting the trade-off between time efficiency and memory utilization.

5.1 Performance Graphs

The performance graphs illustrate the efficiency of different sorting algorithms under various conditions.

Sorting Performance on Reversed, Sorted, and Random Data The first three graphs compare sorting performance on reversed, sorted, and random data, highlighting how different data orders impact algorithm efficiency.

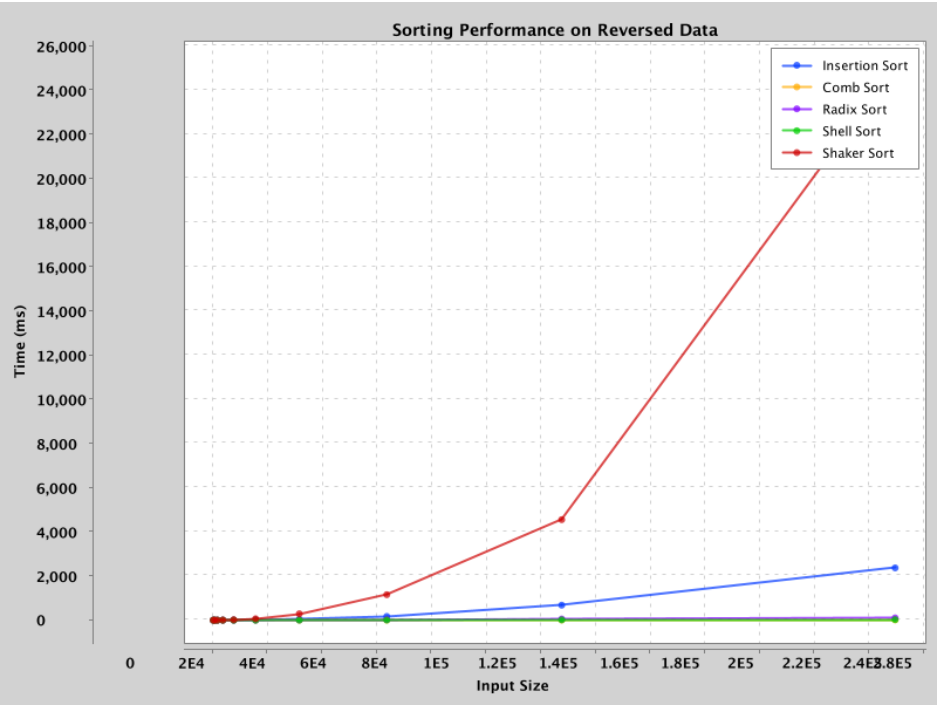

Figure 1: Sorting Performance on Reversed Data.

From Figures 1, 2, and 3, we observe that sorting algorithms perform significantly differently depending on the initial order of the data. Specifically, reversed data generally leads to higher sorting times for algorithms like Insertion Sort and Shaker Sort, which have worse performance on reversed data. In contrast, algorithms like Radix Sort and Shell Sort show more consistent performance across all data types. This variation underscores the importance of data ordering in selecting the most efficient sorting algorithm for specific use cases.

6 Conclusion

In conclusion, this study provided an in-depth comparison of various sorting and searching algorithms, confirming that their performance aligns closely with their theoretical time and space

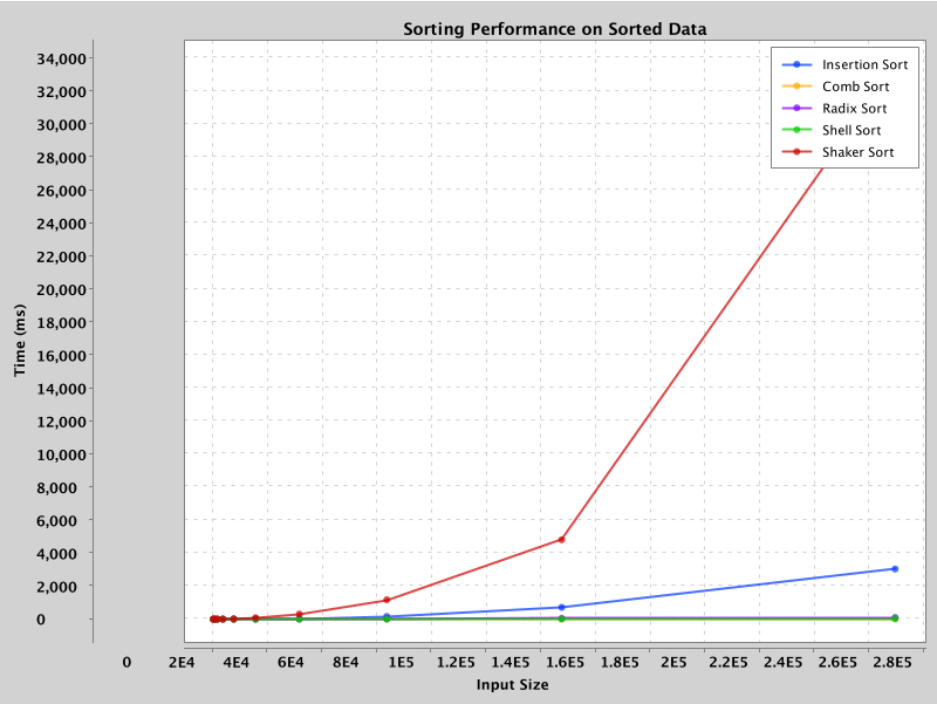

Figure 2: Sorting Performance on Sorted Data.

complexities. While Insertion Sort and Linear Search are efficient for smaller or nearly sorted datasets, Merge Sort and Binary Search offer superior scalability and efficiency for larger datasets. Counting Sort demonstrated its strength for specific data types, and the space complexity analysis highlighted the importance of memory efficiency. Overall, the findings emphasize the need to consider both time and space complexities when selecting algorithms, with future research potentially broadening the scope by testing additional datasets and algorithms.

References

- Geeks for Geeks. (n.d.). Sorting in Java. Retrieved from `https://www.geeksforgeeks.org/sorting-in-java/`
- Baeldung. (n.d.). A Guide to Sorting in Java. Retrieved from `https://www.baeldung.com/java-sorting`
- JavaTpoint. (n.d.). How to sort an array in Java. Retrieved from `https://www.javatpoint.com/how-to-sort-an-array-in-java`

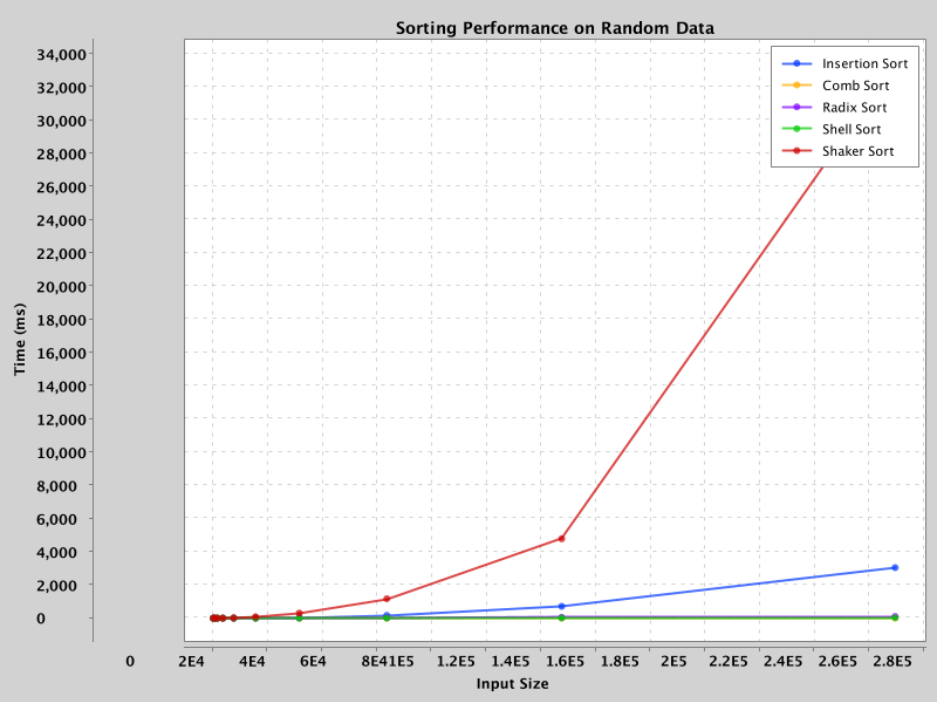

Figure 3: Sorting Performance on Random Data.