Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2025 Spring

Programming Assignment 1

March 21, 2025

Student name:
Sinan Ermiş

Student Number:
b2220356143

1 Problem Definition

In this report, some of the most common sorting algorithms are tested with various input sizes.

2 Solution Implementation

2.1 Comb Sort

```
    public void combSort(int[] A) {
        int gap = A.length;
        final double SHRINK = 1.3;
        boolean sorted = false;

        while (!sorted) {
            gap = (int) Math.floor(gap / SHRINK);
            if (gap < 1) {
                gap = 1;
            }
            sorted = (gap == 1);

            for (int i = 0; i + gap < A.length; i++) {
                if (A[i] > A[i + gap]) {
                    swap(A, i, i + gap);
                    sorted = false;
                }
            }
        }
    }

    private void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
```

2.2 Insertion Sort

```
    public void insertionSort(int[] A) {
        int n = A.length;
        for (int j = 1; j < n; j++) {
            int key = A[j];
            int i = j - 1;

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

```
                i = i - 1;
            }
            A[i + 1] = key;
        }
    }
```

2.3 Shaker Sort

```
    public void shakerSort(int[] A) {
        boolean swapped = true;
        int start = 0;
        int end = A.length - 1;

        while (swapped) {
            swapped = false;

            // Forward pass
            for (int i = start; i < end; i++) {
                if (A[i] > A[i + 1]) {
                    swap(A, i, i + 1);
                    swapped = true;
                }
            }

            if (!swapped) break;

            swapped = false;
            end--;

            // Backward pass
            for (int i = end; i > start; i--) {
                if (A[i] < A[i - 1]) {
                    swap(A, i, i - 1);
                    swapped = true;
                }
            }

            start++;
        }
    }

    private void swap(int[] A, int i, int j) {
        int temp = A[i];
        A[i] = A[j];
        A[j] = temp;
    }
```

2.4 Shell Sort

```
78      public void shellSort(int[] A) {
79          int n = A.length;
80
81          for (int gap = n / 2; gap > 0; gap /= 2) {
82              for (int i = gap; i < n; i++) {
83                  int temp = A[i];
84                  int j = i;
85
86                  while (j >= gap && A[j - gap] > temp) {
87                      A[j] = A[j - gap];
88                      j -= gap;
89                  }
90
91                  A[j] = temp;
92              }
93          }
94      }
```

2.5 Radix Sort

```
95      public void radixSort(int[] A, int maxDigits) {
96          for (int pos = 0; pos < maxDigits; pos++) {
97              A = countingSort(A, pos);
98          }
99      }
100
101     private int[] countingSort(int[] A, int pos) {
102         int[] count = new int[10]; // Assuming decimal digits (0-9)
103         int[] output = new int[A.length];
104         int size = A.length;
105
106         // Counting
107         for (int i = 0; i < size; i++) {
108             int digit = getDigit(A[i], pos);
109             count[digit]++;
110         }
111
112         // Addition
113         for (int i = 1; i < 10; i++) {
114             count[i] += count[i - 1];
115         }
116
117         // Sorting
118         for (int i = size - 1; i >= 0; i--) {
```

```
            int digit = getDigit(A[i], pos);
            count[digit]--;
            output[count[digit]] = A[i];
        }

        return output;
    }
```

3 Results, Analysis, Discussion

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

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

Algorithm	**Input Size n (ms)**									
	500	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
Random Input Data Timing Results										
Comb sort	0.025	0.066	0.090	0.205	0.481	0.919	2.058	4.420	9.667	19.392
Insertion sort	0.040	0.170	0.667	2.603	10.038	44.059	165.071	695.186	2613.052	10869.951
Shaker sort	0.253	0.295	1.073	3.955	15.405	64.072	342.383	1847.858	8415.386	36555.270
Shell sort	0.022	0.066	0.085	0.204	0.448	0.963	2.141	4.708	10.517	19.377
Radix sort	0.018	0.034	0.066	0.131	0.276	0.532	1.071	2.293	4.230	8.451
Sorted Input Data Timing Results										
Comb sort	0.015	0.041	0.029	0.064	0.145	0.322	0.790	1.516	3.682	6.987
Insertion sort	0.001	0.001	0.001	0.003	0.007	0.016	0.031	0.067	0.122	0.239
Shaker sort	0.001	0.000	0.000	0.002	0.003	0.008	0.013	0.024	0.046	0.096
Shell sort	0.009	0.026	0.010	0.022	0.049	0.110	0.244	0.506	1.096	2.270
Radix sort	0.018	0.035	0.066	0.135	0.278	0.535	1.100	2.118	4.247	8.537
Reversely Sorted Input Data Timing Results										
Comb sort	0.019	0.046	0.039	0.083	0.179	0.391	0.818	1.783	4.054	7.941
Insertion sort	0.082	0.320	1.283	5.030	19.769	77.729	317.531	1415.411	5083.767	19349.034
Shaker sort	0.245	0.232	0.918	3.714	15.625	61.110	245.555	996.731	3928.961	14904.179
Shell sort	0.014	0.042	0.018	0.040	0.085	0.180	0.393	0.784	1.694	3.441
Radix sort	0.018	0.034	0.067	0.133	0.284	0.536	1.122	2.102	4.244	8.311

Table 2: Computational complexity comparison of the given algorithms.

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

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 ($O(1)$): Only uses a few integer variables (gap, SHRINK, sorted, i, j)
- Insertion Sort ($O(1)$): Only uses a few integer variables (key, i, and j)
- Shaker Sort ($O(1)$): Only uses a few integer variables (n, gap, i, j, and temp)
- Shell Sort ($O(1)$): Only uses a few primitive variables (swapped, start, end, temp)
- Radix Sort ($O(n+k)$): k is the range of the input

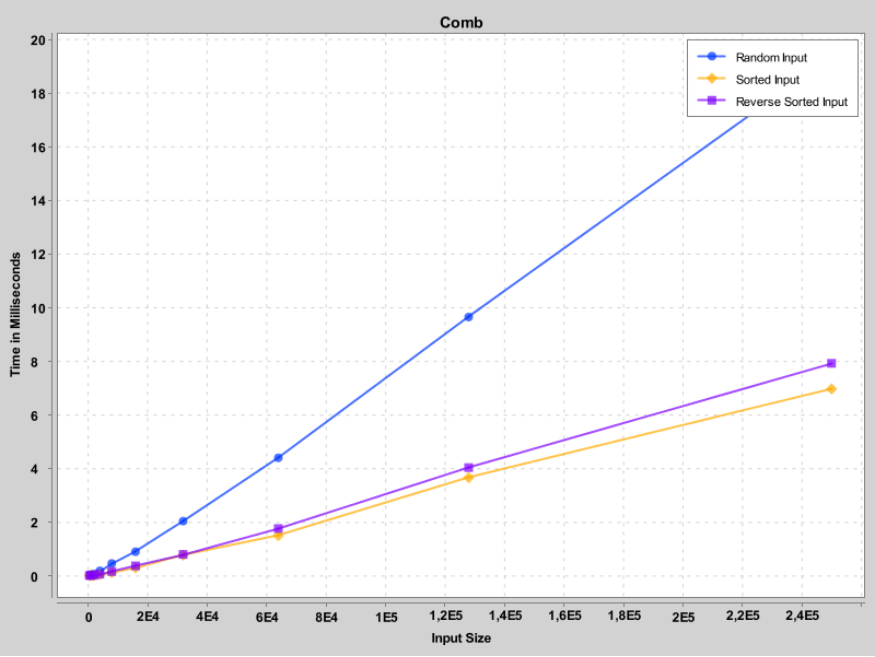

Figure 1: Time to Input Size Graph of Comb Sort

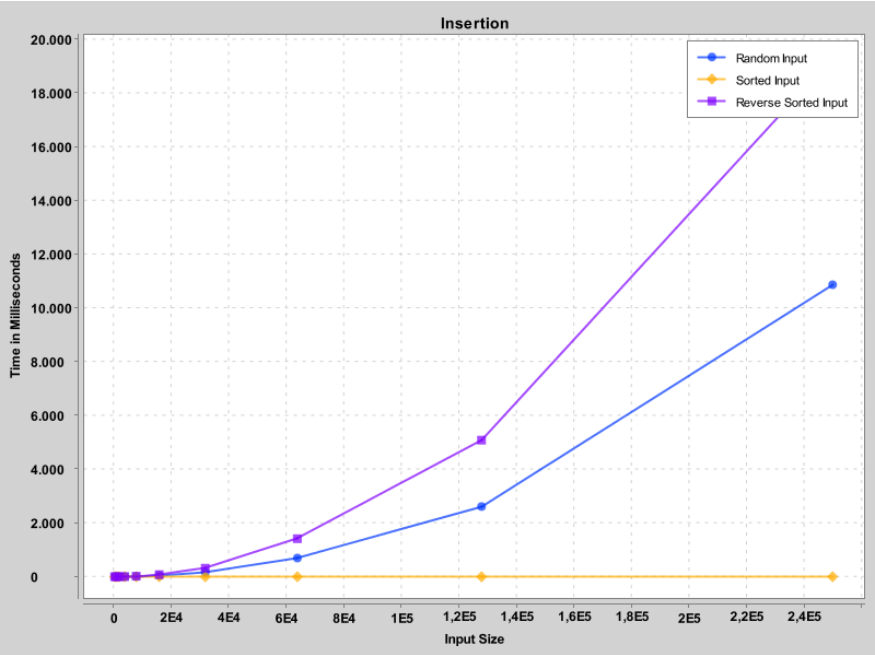

Figure 2: Time to Input Size Graph of Insertion Sort

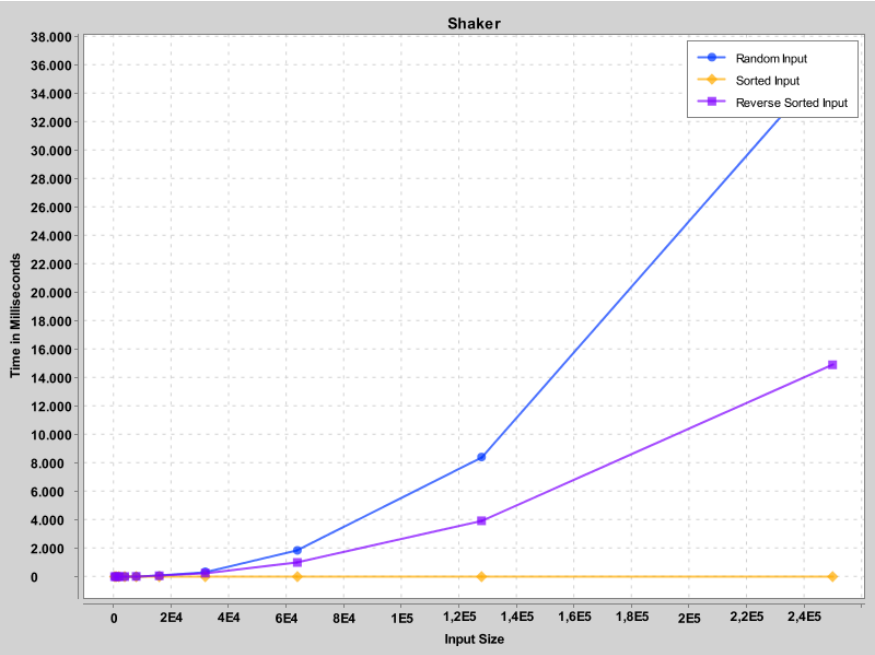

Figure 3: Time to Input Size Graph of Shaker Sort

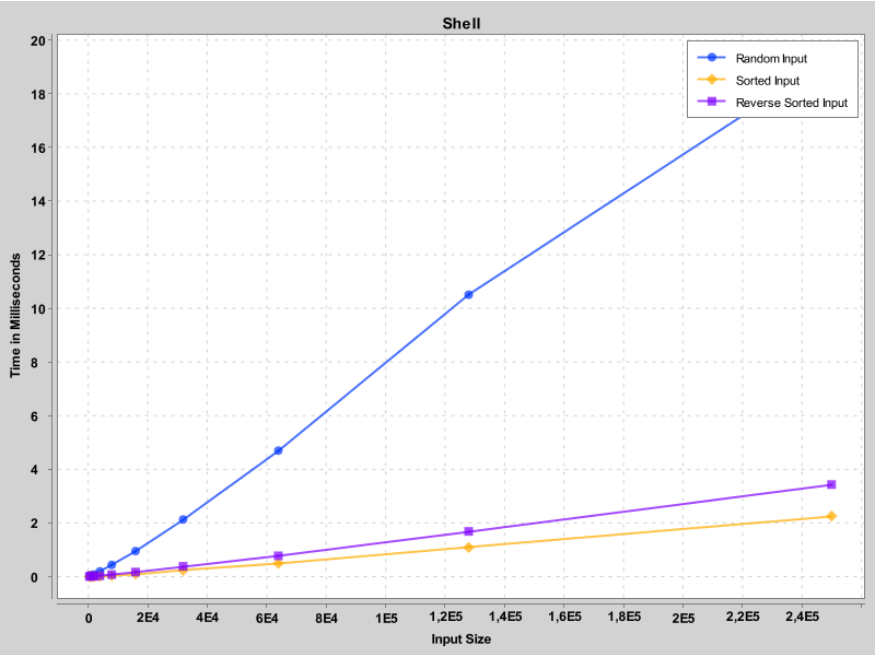

Figure 4: Time to Input Size Graph of Shell Sort

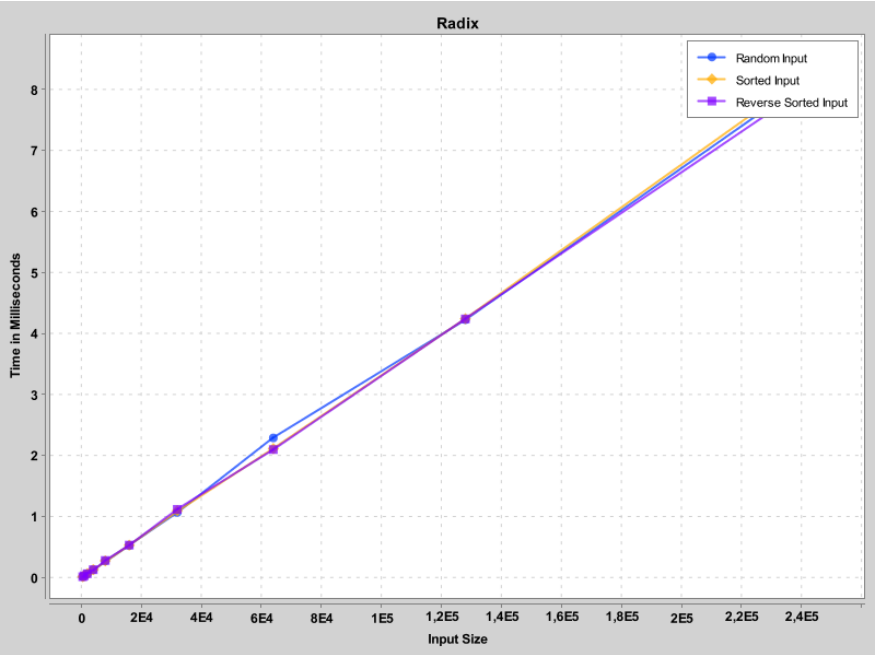

Figure 5: Time to Input Size Graph of Radix Sort

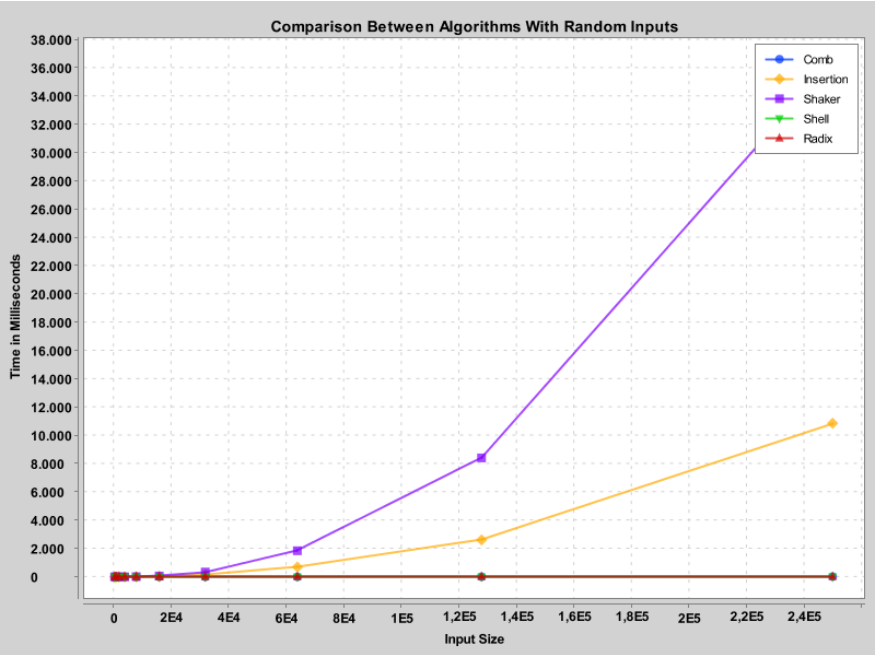

Figure 6: Time to Input Size Graph of Sorting Algorithms With Random Inputs

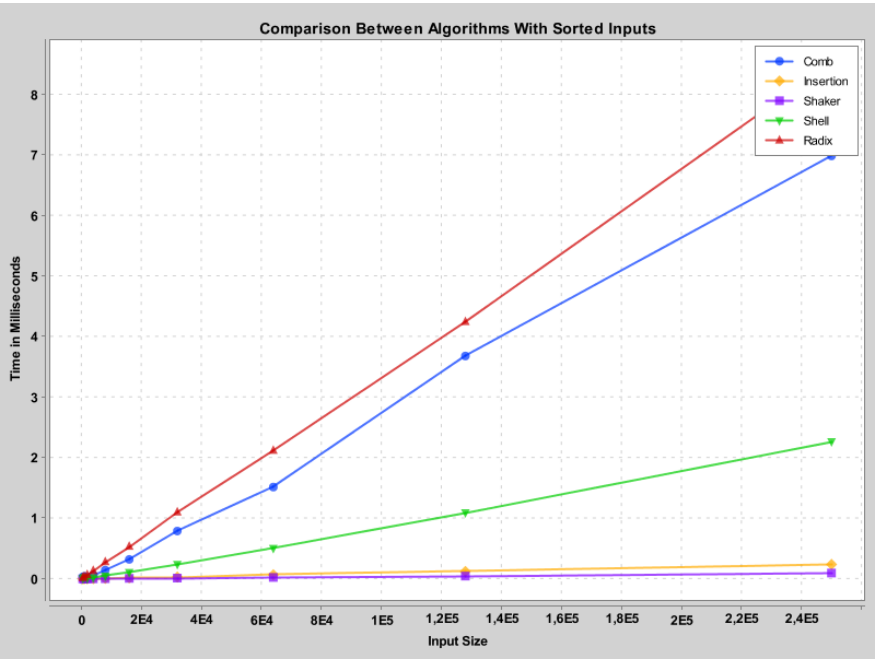

Figure 7: Time to Input Size Graph of Sorting Algorithms With Sorted Inputs

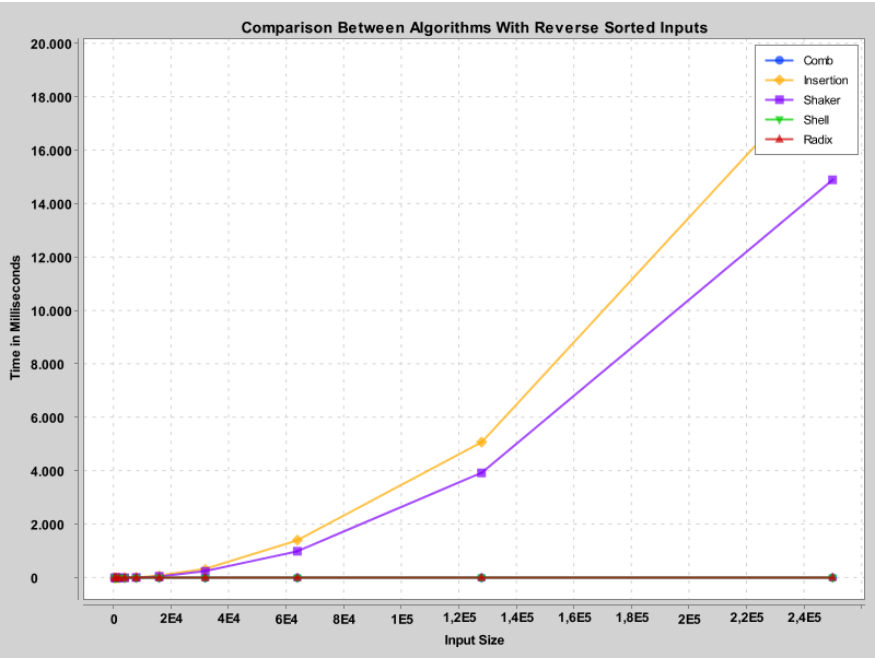

Figure 8: Time to Input Size Graph of Sorting Algorithms With Reverse Sorted Inputs

- **Question:** What are the best, average, and worst cases for the given algorithms in terms of the given input data to be sorted?
  Answer:
  Comb Sort:
  Best Case: Nearly Sorted / Sorted $\Omega(nlog(n))$
  Average Case: Random $\Theta(n^2/2^p)$
  Worst Case: Reverse Sorted $O(n^2)$

  Insertion Sort:
  Best Case: Sorted $\Omega(n)$
  Average Case: Random $\Theta(n^2)$
  Worst Case: Reverse Sorted $O(n^2)$

  Shaker Sort:
  Best Case: Sorted $\Omega(n)$
  Average Case: Random $\Theta(n^2)$
  Worst Case: Reverse Sorted $O(n^2)$

  Shell Sort:
  Best Case: Sorted $\Omega(nlog(n))$
  Average Case: Random, performance depends on the gap sequence $\Theta(nlog(n))$
  Worst Case: Happens when a poor gap sequence is used or when the input is arranged in an order that negates the benefit of early swaps $O(n^2)$

  Radix Sort: However the data is arranged, radix sort needs to process all numbers. Every case depends on input size n and digit count k.

- **Question:** Do the obtained results (running times of your algorithm implementations) match their theoretical asymptotic complexities?

  Answer: Yes, they do. For example, when you look at 2 you can see that with reverse-sorted (worst case) inputs the graph is exponential and when it is sorted (best case) the graph is linear.

- **Question:** Do Shaker Sort and Comb Sort improve performance compared to Bubble Sort? If so, how do their approaches contribute to this improvement?

  Answer: Shaker Sort reduces the number of unnecessary comparisons and swaps. Instead of only moving elements forward like Bubble Sort, Shaker Sort moves elements in both directions (left-to-right, then right-to-left). This helps larger elements move to the right faster and smaller elements move to the left faster. So, it reduces the total number of swaps needed,

especially when the largest or smallest elements are near their correct positions.
In Comb Sort, instead of swapping close elements like Bubble Sort, Comb Sort swaps distant elements by reducing the gap. It does so over time, thereby making it quicker to sort out small values stuck at the end and big values stuck at the beginning.

- **Question:** Is Shell Sort perform better than Insertion Sort for larger datasets? Under what conditions does Insertion Sort still perform well?
  Answer: Shell Sort moves elements in larger increments. If the input is already sorted (best case), Insertion Sort performs better than Shell Sort ($\Omega(n)$ vs $\Omega(nlog(n))$ 7. In other scenarios, for both the average and the worst cases, the Shell Sort performs better 6 8.

- **Question:** Given that Radix Sort is a non-comparison-based sorting algorithm, how does it handle large numerical ranges efficiently?
  Answer: It sorts numbers digit by digit. For example, sorting a range of 1 to 1,000,000 takes only 6 passes (since 1,000,000 has 6 digits).

References

- Quora: What is the best, average, worst case time complexity for Radix Sort?
- Geeks for Geeks: Is Comb Sort better than Bubble Sort?
- Programiz Blog: Exploring Time and Space Complexities of Shell Sort