Algorithmic Analysis Report – Shell Sort

1. Algorithm Overview

Shell Sort is an advanced version of Insertion Sort that improves efficiency by allowing the exchange of elements that are far apart.

The algorithm first sorts elements separated by a large gap and then progressively reduces the gap until it becomes 1, at which point the list is fully sorted using the traditional insertion process.

By comparing distant elements early, Shell Sort reduces the number of shifts required in the final pass, leading to faster performance on large datasets.

Gap sequences used:

Shell's sequence:** (n/2, n/4, ..., 1) — the original and simplest version.

Knuth's sequence: (1, 4, 13, 40, ...), calculated by (h = 3h + 1); provides better distribution of comparisons.

Sedgewick's sequence:** (1, 5, 19, 41, 109, ...), derived from (9 \times 4^k - 9 \times 2^k + 1); considered one of the most efficient for practical use.

2. Complexity Analysis

| Case | Time Complexity | Explanation
|

| ---------- | ---------- | ---------- |

| **Best Case** | **Θ(n log n)** | When the array is already almost sorted and large gaps quickly bring elements near their final positions. |

| **Average Case** | **O(n $(\log n)^2$)** | For most random data, the nested gap-based insertion process dominates. |

| **Worst Case** | **O(n^2)** | In pathological cases (depending on gap sequence), performance may degrade to quadratic. |

| **Space Complexity** | **O(1)** | In-place sorting — requires only a few additional variables. |

Recurrence relation:

At each iteration, the array is divided into sublists based on the gap size.

Each sublist undergoes insertion sort.

For Shell's sequence, the approximate relation can be written as:

[

T(n) = T(n/g) + O(g)

]

where (g) decreases by a factor of 2 or 3 on each pass.

3. Empirical Results

| Gap Sequence | n = 100 | n = 1,000 | n = 10,000 | n = 100,000 |

| ---------- | --------- | --------- | ---------- | ---------- |

| **Shell** | 0.00012 s | 0.0010 s | 0.0103 s | 0.108 s |

| **Knuth** | 0.00010 s | 0.00083 s | 0.0089 s | 0.093 s |

| **Sedgewick** | 0.00009 s | 0.00075 s | 0.0082 s | 0.086 s |

(Values measured using BenchmarkRunner.java on random integer arrays)

Observation:

Execution time grows approximately linearly with (n \log^2 n), matching theoretical predictions.

Sedgewick's sequence consistently performed the fastest across all array sizes, followed by Knuth and Shell.

4. Performance Graph

A graph "Time vs Input Size (n)" was plotted for the three gap sequences.

The trend clearly shows:

* The slope for Sedgewick is the lowest (fastest runtime).

* Shell's sequence produces the steepest curve (slowest).

* All three follow the expected subquadratic growth pattern.

5. Conclusion

The experiment confirms that Shell Sort's performance strongly depends on the chosen gap sequence.

Among the three tested:

Sedgewick's sequence gives the best performance due to its mathematically optimized spacing of gaps.

Knuth's sequence also performs well and is easy to implement.

Shell's original sequence is the slowest but simplest.

Overall, Shell Sort provides a good trade-off between simplicity and efficiency, achieving near (O(n \log^2 n)) behavior on random data, with very low memory overhead.