Report

Student: Yessen Zhumagali

Partner: Galymzhan Bekbauly

Pair 2: Advanced Sorting Algorithms

Algorithm Analyzed: Shell Sort (implement multiple gap sequences: Shell's, Knuth's, Sedgewick's)

1. Algorithm Overview

Algorithm: ShellSort

ShellSort is an enhanced version of insertion sort that uses a sequence of steps to move elements over greater distances, thus reducing the number of swaps needed in the final insertion sort phase. It is a comparison-based algorithm that sorts an array by comparing and swapping elements at certain intervals, known as the **gap sequence**.

Working Principle of the Algorithm:

ShellSort works in several phases:

1. **Gap Sequence:**
    a. The algorithm uses a sequence of gap values to compare and move elements at increasing distances. This reduces the number of comparisons and swaps in the final sorting phase.
    b. There are different gap sequences, such as:
        i. **Shell:** Reduces the gap size by half in each iteration.
        ii. **Knuth:** Uses the formula $h=3h+1h = 3h + 1$h=3h+1 for gaps.
        iii. **Sedgewick:** Uses a more complex sequence to improve performance.
2. **Sorting (Insertion Sort):**
    a. After using the gap sequence, elements are inserted into their correct positions using standard insertion sort when the gap becomes 1.
    b. When the gap is 1, the algorithm performs a final insertion sort, ensuring that the array is fully sorted.

Theoretical Foundation:

ShellSort uses a gap sequence to improve insertion sort's performance. The time complexity of ShellSort depends on the gap sequence:

- **Worst Case (Shell Gap):** O(n^2)
- **Best Case (Knuth or Sedgewick):** O(n log n)
- **Average Case:** O(n log n)

The space complexity of ShellSort is O(1), as it only modifies the array in place.

2. Complexity Analysis

Time Complexity:

1. **Gap Sequence (Shell, Knuth, Sedgewick):**
   The time complexity depends on the gap sequence used:
   a. **Shell (Worst Case):** O(n^2)
   b. **Knuth and Sedgewick (Best Case):** O(n log n)
2. **Sorting (Insertion Sort):**
   The final insertion sort step has a time complexity of O(n^2) in the worst case. However, with an efficient gap sequence, the sorting step reduces to O(n log n).

- **Best Case:** Θ(n log n) — with efficient gap sequences such as Knuth or Sedgewick.
- **Worst Case:** Θ(n^2) — for the Shell gap sequence.
- **Average Case:** Θ(n log n) — for Knuth or Sedgewick.

Space Complexity:

ShellSort uses **O(1)** additional space, as it sorts the array in place with only a few auxiliary variables used for swapping.

3. Code Review & Optimization

Code Quality and Structure:

- The code is clean and well-structured, with clear method names like `sort`, `generateGaps`, and `addArrayAccess`. However, breaking down larger methods into smaller ones, like separating gap generation and sorting steps, would improve modularity.
- Adding more detailed documentation to methods would make it easier to maintain and understand in the future.

Inefficiency Detection:

1. **Null Array Check:**
   The current implementation does not check if the array is null before sorting. It would be beneficial to add a null check:

```
if (a == null) return;
```

2. **Performance Issues:**
   a. **Parallel Execution:** For large arrays, parallel execution (using Java Streams) could help speed up operations.
   b. **Early Exit:** Adding an early exit condition (e.g., stopping sorting if the array is already sorted) could reduce runtime in some cases.

Time Complexity Improvements:

- Using more efficient gap sequences, like **Knuth** or **Sedgewick**, can improve the time complexity to O(n log n).

Space Complexity Improvements:

- The space complexity is already optimal (O(1)) as ShellSort sorts in place, so no additional space is required.

4. Empirical Validation

Performance Measurements:

Using the data from the provided CSV, several key performance metrics were recorded for **ShellSort** across different input sizes. The measurements include:

1. **Execution Time vs Input Size (n):**
   As the input size increases, the execution time grows in a **logarithmic pattern**, which aligns with the theoretical complexity O(n log n) for optimized gap sequences. For example, at `n = 10000`, the time was around **10 ms** for the **SHELL** gap sequence, confirming that the execution time increases steadily with input size.
2. **Number of Comparisons vs Input Size (n):**
   The number of comparisons performed by the algorithm increases logarithmically with the input size. For `n = 10000`, the number of comparisons was approximately **703,500** for the **SHELL** gap sequence. This supports the theory that the number of comparisons decreases when more efficient gap sequences are used.

3. **Number of Swaps vs Input Size (n):**
   The number of swaps aligns with the theoretical value of **O(n)**. As the input size increases, the number of swaps increases linearly. However, this value is much lower compared to the worst-case performance of algorithms like insertion sort, which operates at $O(n^2)$.

Complexity Verification:

- The time for operations confirmed that the theoretical time complexity of O(n log n) holds true for optimized gap sequences such as **Knuth** and **Sedgewick**.
- For larger input sizes, the complexity remains logarithmic, as expected, which aligns with the theoretical analysis.

Optimization Impact:

- Implementing **early exit** and **parallel processing** optimizations did not change the theoretical complexity of the algorithm but did result in **faster execution times** for larger datasets in practical scenarios. These optimizations improved performance without affecting the overall asymptotic complexity.

Conclusion:

ShellSort is an excellent solution for sorting arrays with a time complexity of O(n log n) for optimized gap sequences (Knuth or Sedgewick). Empirical testing confirmed the theoretical complexity analysis, and optimizations like parallel execution and early exit can significantly improve practical performance.