Individual Analysis Report: ShellSort Implementation

Course: Algorithmic Analysis

Author: Onalbayev Nursalim SE-2425
Date: 05.10.2025

---

1. Algorithm Overview

Shell Sort is an advanced comparison-based sorting algorithm that extends the idea of insertion sort.
Instead of comparing adjacent elements, Shell Sort compares elements that are a fixed gap apart. The gap sequence decreases over time until it becomes 1, at which point the algorithm performs a standard insertion sort.

- **Main idea:** perform multiple passes of gapped insertion sorts.
- **Gap sequences:** the choice of gap sequence is critical. The simplest is halving (n/2, n/4, …, 1), but more efficient ones exist (Knuth, Sedgewick, Tokuda).
- **Characteristics:**
  - In-place algorithm (O(1) extra memory).
  - Not stable (relative order of equal elements may change).
  - Highly dependent on input distribution and gap sequence.

---

2. Complexity Analysis

2.1 Time Complexity

- **Best Case:**
  - For nearly sorted data, Shell Sort can approach O(n log n).
  - This occurs when the initial gaps quickly reduce the disorder.
- **Average Case:**
  - Depends strongly on the gap sequence.
  - For basic halving gap sequence: typically O(n^(3/2)) ≈ O(n^1.5).
  - For optimized gap sequences: $O(n \log^2 n)$ or better.
- **Worst Case:**
  - With halving gap sequence: $O(n^2)$.
  - No tight general bound is known for all sequences, but practical sequences avoid quadratic growth.

2.2 Space Complexity

- In-place algorithm: O(1) auxiliary space.

2.3 Comparison with HeapSort

- HeapSort guarantees Θ(n log n) in all cases.
- ShellSort's performance fluctuates between O(n log n) and O(n^2), depending on gaps.
- HeapSort is more predictable for large-scale datasets.

---

3. Code Review & Optimization

Observations

- The repository provides a basic implementation of Shell Sort in Java.
- Commit history is minimal (only two commits), suggesting limited development iteration.
- No **unit tests** included.
- No **performance metrics** (comparisons, swaps, array accesses, execution time).
- Gap sequence implemented is the simplest (n/2, …, 1).

Identified Issues

- **Maintainability:** Code is short but lacks modularity and comments.
- **Readability:** No clear package structure (algorithms/, metrics/, cli/).
- **Testing:** Edge cases not covered (empty array, single element, duplicates, sorted/reverse arrays).
- **Performance Tracking:** Missing instrumentation for empirical analysis.

Optimization Suggestions

- Introduce **PerformanceTracker** class to count comparisons, swaps, and execution time.
- Add **unit tests** with JUnit 5 for correctness and edge cases.
- Experiment with alternative **gap sequences** (Knuth: 1, 4, 13, …; Sedgewick; Tokuda).
- Improve code modularity: place algorithm in algorithms/ package, tests in src/test/java/.
- Expand README with usage instructions and theoretical background.

---

4. Empirical Results

The current implementation does not provide benchmarks or CSV data.
A proper evaluation should include:

- Input sizes: n = 100, 1,000, 10,000, 100,000.

- Input distributions: random, sorted, reverse-sorted, nearly sorted.
- Metrics: execution time, comparisons, swaps.
- Graphs: **time vs n**, compared to theoretical $O(n \log^2 n)$ and $O(n^2)$.

Expected Behavior (based on theory)

- For small n, ShellSort is competitive and sometimes faster than HeapSort due to fewer overheads.
- For large n, HeapSort outperforms ShellSort because of guaranteed $O(n \log n)$.
- Gap sequence optimization significantly impacts scaling.

---

5. Conclusion

ShellSort is an elegant improvement over insertion sort that benefits from gap-based comparisons.
However, the current implementation in the repository has several limitations:

- No benchmarks or metrics to validate theoretical performance.
- Uses only a basic gap sequence, leading to quadratic worst-case time.
- Lacks modular structure, testing, and documentation.

Recommendations

- Extend implementation with metrics and empirical analysis.
- Add multiple gap sequences and compare performance.
- Improve repository structure and commit history for better maintainability.
- Compare empirical results against HeapSort to confirm theoretical expectations.

Appendix (Optional for Report)

- **Table:** complexity summary of ShellSort vs HeapSort.
- **Charts:** (to be added after running benchmarks).