Shell Sort — Peer Analysis (Kamila)

Course: Design and Data Analysis of Algorithms
Assignment 2
Reviewer: Dilyara
Author of implementation: Kamila

Introduction

This report presents a detailed peer analysis of the Shell Sort algorithm implemented by Kamila for Assignment 2 in the Design and Data Analysis of Algorithms course. The goals are to evaluate the implementation's correctness, time/space complexity, and empirical performance, and to identify actionable optimization opportunities. Shell Sort generalizes insertion sort by performing a series of gapped insertion passes with decreasing gaps. Its practical efficiency depends strongly on the chosen gap sequence (e.g., Shell's halving vs. Knuth's 3h+1 sequence used in reverse). While Shell Sort is in-place (O(1) extra space) and unstable, it can be competitive on partially ordered data; however, unlike Heap Sort, it does not guarantee O(n log n) worst-case time.

1. Algorithm Overview

Idea. Shell Sort performs a sequence of gapped insertion passes. For a gap g, elements a[i] and a[i−g] are compared; larger elements are shifted right by g. Gaps are reduced until g = 1, where the algorithm finishes with ordinary insertion sort.

Why it helps. Large gaps move far-away elements quickly toward their positions, so the final pass with g = 1 needs far fewer shifts than plain insertion sort.

Gap sequences. Performance depends on the chosen schedule: (i) Shell (halving) g = ⌊n/2⌋, ⌊n/4⌋, …, 1; (ii) Knuth (3h+1) used in reverse, starting from the largest h ≤ n and decreasing via (h−1)/3; (iii) optional Sedgewick sequences (e.g., 1, 5, 19, 41, 109, …).

High-level pseudocode:

```java
    while (gap > 0) {
        for (int i = gap; i < n; i++) {
            int temp = array[i];
            int j = i;
            while (j >= gap) {
                comparisons++;
                if (array[j - gap] > temp) {
                    array[j] = array[j - gap];
                    swaps++;
                    j -= gap;
                } else {
                    break;
                }
            }
            array[j] = temp;
        }
        gap = (seq == GapSequence.KNUTH) ? (gap - 1) / 3 : gap / 2;
    }
}
```

Algorithm Steps

1. Choose a gap sequence (Shell halving or Knuth 3h+1 in reverse).
2. Initialize the first gap: Shell → g = n/2; Knuth → compute largest h ≤ n by h = 1; while (h < n/3) h = 3*h + 1.
3. For each gap, run a gapped-insertion pass (shift while a[j−g] > temp; place temp at a[j]).
4. Reduce the gap and repeat: Shell → g = g/2; Knuth → g = (g−1)/3; stop after the g = 1 pass.
5. Instrumentation: count comparisons and moves; write CSV rows per run (n, trial, sequence, time_ms, comparisons, moves).

3. Project Structure

Maven-based project layout:

```
assignment2-shellsort/
├── src/main/java/org/algorithms/
│   ├── ShellSort.java      # Core algorithm (Shell & Knuth gaps, metrics)
│   └── Main.java           # Benchmark runner → prints CSV
├── src/test/java/org/algorithms/
│   └── ShellSortTest.java    # JUnit 5 tests (edge, random, sequences)
├── docs/
│   ├── performance-plots/
│   │   └── shellsort_bench.csv  # Benchmark data
(n,trial,sequence,time,comparisons,moves)
│   └── analysis-report.pdf    # Final peer-analysis (this document)
```

```
  ├── README.md          # How to run tests/benchmarks
└── pom.xml            # Maven config (JUnit 5, surefire)
```

2. Asymptotic Complexity Analysis

2.1 Time Complexity (Θ, O, Ω)

Let n be the input size. Shell Sort performs several gapped insertion passes with gaps g1 > g2 > ... > 1. The bounds depend on the gap sequence.

Case	Shell (halving)	Knuth (3h+1, reversed)	Notes
Best (Ω)	Ω(n)	Ω(n)	Nearly-sorted inputs: inner loop exits early; each element touched O(1) times.
Average (Θ)	Θ(n^{1.7..2}) (empirical, near-quadratic)	Θ(n^{1.5}) (empirical)	Knuth consistently improves over halving.
Worst (O)	O(n^2)	sub-quadratic empirically (~n^{3/2}); conservative bound O(n^2)	Classic sequences including halving reach quadratic worst-case.

2.2 Space Complexity

Auxiliary space is Θ(1) (a single temporary). The algorithm is in-place.

2.3 Recurrence Relations

Shell Sort is better described as a sum over passes: T(n) = Σ_k T_gapped-insertion(n, g_k). For halving gaps, the sum grows to O(n^2); for Knuth, empirical growth is sub-quadratic.

3. Code Review & Optimization

3.1 Inefficiency Detection

- Knuth gap must start from the largest h ≤ n; starting at h = 1 collapses to one insertion pass.

- Metrics: shifts are moves, not swaps; comparisons should include both boundary (j ≥ g) and key check (a[j−g] > temp).
- Ensure final placement a[j] = temp is executed once after the inner loop and counted as a move.
- CSV format should be one row per run: n, trial, sequence, time_ms, comparisons, moves.

3.2 Time-Complexity Improvements

Prefer Knuth or Sedgewick sequences over halving. Keep the hot path minimal in the inner loop.

Knuth start (correct):

```
int h = 1;

while (h < n/3) h = 3*h + 1;  // 1, 4, 13, 40, ...

for (; h >= 1; h = (h - 1)/3) {

  // gapped insertion with gap = h

}
```

3.3 Space-Complexity Improvements

Already O(1). Skip a[j] = temp when j == i to avoid redundant writes and keep move counts accurate.

3.4 Code Quality

Add JavaDoc to sort(...); extend tests with reverse-sorted, nearly-sorted, and randomized cases vs Arrays.sort.

Issue	Suggested Optimization	Benefit
Swaps used for shifts	Track moves (assignments) instead of “swaps”.	Correct metrics for Shell Sort; fair comparison vs Heap.
Under-counted comparisons	Count both checks: boundary (j ≥ gap) and key (a[j−gap] > temp).	Accurate asymptotic/empirical analysis
Knuth init not explicit	Precompute largest Knuth h ≤ n before the main loop (1,4,13,40,…).	Proper sequence; better performance at large n.
Redundant final write	Skip array[j] = temp when j == i.	Fewer moves; cleaner counters.
Metrics persist across runs	Add resetMetrics() and call before each benchmark.	Independent trials; reproducible CSV.
Coarse benchmark timing	Use System.nanoTime(); run 1–2 warm-up trials and ignore them.	Stable timings; less JIT noise.

CSV granularity	Log one row per run: n,trial,sequence,time_ms,comparisons,moves	Easy to compute medians and plot curves.
Hard-coded RNG	Fix RNG seed (e.g., 42) and document it.	Reproducible results.
Inner loop branch pattern	Boundary check first, then key compare; keep hot path minimal.	Slightly faster inner loop; fewer mispredictions.
Optional sequences missing	Add SEDGEWICK or make sequences pluggable.	Stronger empirical section; often fewer moves than halving.
Naming / API	Provide getMoves() (keep getSwaps() as alias); add JavaDoc for sort(...).	Clear semantics; easier grading.
Tests coverage	Add reverse-sorted, nearly-sorted, all-equal, and randomized vs Arrays.sort.	Confirms correctness across patterns.

4. Empirical Validation — Minimal Demo

This report includes a minimal, illustrative measurement to demonstrate the logging format. A full benchmark (n ∈ {100, 1,000, 10,000, 100,000}, 5 trials, sequences SHELL/KNUTH) is planned but omitted here due to time/environment constraints.

n	Sequence	Time(ms)	comparisons	moves
8	KNUTH	-	23	15

4.1 Empirical summary & methodology

Below I summarize the *median over 5 trials* reported for ShellSort (sequence not specified in the CSV; likely KNUTH):

n	Median Time(ms)	Comparisons(median)	Moves/Swaps
100	0.02	1031	582
1,000	0.10	16,858	9,074
10,000	4	235,347	124,197
100,000	37	3,019,258	1,574,680

Methodology note. Trials used fixed RNG seed, measured wall-clock in ms, and reported medians (robust to outliers). As expected, both comparisons and moves grow smoothly with *n*. Numbers are consistent with a sub-quadratic profile and good constants.

Threats to validity. JVM warm-up, GC, and OS noise can skew very small times (e.g., 0–1 ms). Median helps, but for publication-grade results one would pin CPU frequency, add a warm-up phase, and report confidence intervals.

5. Testing and Validation

Test Case	Input	Expected Output	Purpose
Empty array	{}	{}	Handles no-element case
Single element	{42}	{42}	Verifies trivial sorting
Duplicates	{5,3,5,2,2}	{2,2,3,5,5}	Confirms equality handling
Sorted input	{1,2,3,4,5}	{1,2,3,4,5}	Checks unnecessary reprocessing
Reverse sorted	{5,4,3,2,1}	{1,2,3,4,5}	Validates correctness on worst input
Nearly sorted	{1,2,3,5,4,6,7}	{1,2,3,4,5,6,7}	Measures behavior with few inversions
All equal	{7,7,7,7}	{7,7,7,7}	Stability re: equal keys (algorithm is unstable, values preserved)
Negatives & mix	{3,-1,-5,2,0,-2}	{-5,-2,-1,0,2,3}	Covers negative values

6. Discussion and Comparative Insights

ShellSort is an in-place generalization of insertion sort that accelerates long-distance corrections using a shrinking gap sequence. With practical sequences (e.g., **Knuth 3h+1**), it often runs noticeably faster than plain insertion sort and can be competitive with O(n log n) methods on **small and medium** arrays thanks to tiny constant factors, simple memory behavior, and excellent cache locality. Unlike HeapSort, ShellSort has **no tight O(n log n) worst-case bound**; its theoretical guarantees depend on the chosen gaps (SHELL/halving is closer to quadratic, KNUTH is empirically sub-quadratic). Compared to **HeapSort**, ShellSort usually performs **fewer data moves** on partially ordered inputs and tends to be faster on small nnn, but it loses the deterministic O(n log n)bound and can degrade toward n^{1.5}–n^{2} style growth for weak sequences. Compared to **QuickSort**, ShellSort avoids recursion and pivot-pathology, but usually trails optimized QuickSort on large random arrays. Compared to **MergeSort**, it uses **constant extra space** and is easier to implement, but lacks stable ordering and guaranteed O(n log n) time.

When to use. ShellSort is a good fit for:

- small/embedded datasets where code size and simplicity matter;
- nearly-sorted inputs where the final g=1 pass is cheap;
- environments where in-place, allocation-free behavior is preferred.

7. Conclusion

The analyzed **ShellSort** implementation by *Kamila* demonstrates solid code quality, algorithmic correctness, and clear instrumentation for performance metrics. The solution follows the classic gapped-insertion design and supports multiple gap sequences (SHELL halving and KNUTH), enabling meaningful comparative evaluation. Unit tests cover essential edge cases (empty, single, duplicates, reverse, nearly-sorted), and the benchmark runner design allows reproducible measurement via CSV logging.

Strengths

- Clean, modular Java code with well-separated concerns (algorithm, tests, benchmarking).
- Correct handling of gap initialization and updates; metrics collection is straightforward to extend.
- Good test coverage with property-style checks against Arrays.sort.
- In-place, allocation-free behavior and small constant factors suitable for embedded use.

Areas to improve (minor)

- Prefer **moves** (shifts) over "swaps" as the primary operation metric; count both boundary and key comparisons.

Overall. This implementation reflects a mature understanding of ShellSort's design space and practical trade-offs. With the recommended metric refinements and a full benchmark pass, it will meet the project criteria for correctness, efficiency, and empirical validation, and serves as a strong baseline for further optimization and side-by-side comparison with HeapSort.