# Shell Sort Benchmark Report

Course: Design and Analysis of Algorithms

Student: Tazhibayev Marsel SE-2403

Instructor: Khaimuldin Nursultan

Partner: Aldiyar

Date: 10.10.2025

------------------------------------------------------------

## 1. Objective

The purpose of this assignment was to:  
- Analyze a peer’s implementation of the Shell Sort algorithm.  
- Verify correctness, structure, and metric tracking mechanisms.  
- Detect logical and structural issues in the implementation.  
- Evaluate theoretical and empirical complexity.  
- Document findings and propose optimizations.

## 2. Algorithm Overview

Problem: Sort an array of integers using the Shell Sort algorithm — a generalization of insertion sort that compares elements separated by a “gap” which gradually decreases.  
  
Algorithm Logic:  
1. Choose a sequence of gap sizes.  
2. For each gap, perform a gapped insertion sort.  
3. Gradually reduce the gap until it reaches 1.  
4. The final pass (gap=1) behaves like insertion sort on an almost sorted array.

Gap Schemes Implemented:  
- SHELL: n/2, n/4, n/8, ...  
- KNUTH: 1, 4, 13, 40, ...  
- SEDGEWICK: 1, 5, 19, 41, ...  
  
Variants:  
- Swap-based inner loop  
- Shift-based inner loop

## 3. Theoretical Complexity

|  |  |  |  |
| --- | --- | --- | --- |
| Case | Time Complexity | Space | Explanation |
| Best | Ω(n log n) | O(1) | Depends on gap sequence; for Knuth gaps, behaves close to n·(log n)² |
| Average | Θ(n^(3/2)) | O(1) | Intermediate performance between Insertion and Merge Sort |
| Worst | O(n²) | O(1) | Poor gap selection can degrade to quadratic complexity |
| Stability | Not stable | — | Element order may change for equal keys |

## 4. Implementation Overview

Files:  
- ShellSort.java — core algorithm, three gap schemes, swap/shift toggle.  
- PerformanceTracker.java — metric tracking (time, swaps, comparisons).  
- BenchmarkRunner.java — CLI utility for automated testing and CSV logging.  
- ShellSortTest.java — JUnit 5 test suite.  
  
Metrics Recorded:  
- Runtime (microseconds)  
- Comparisons  
- Swaps  
- Array accesses  
- Memory allocations

## 5. Detected Issues

1. Missing Metric Instrumentation:  
The code never increments counters for comparisons, swaps, or array accesses.  
→ Benchmark results will always show zero values.  
Only elapsed time is correctly measured.  
  
2. Incorrect Sedgewick Gap Sequence:  
Uses min(a,b) instead of standard sequence; can generate invalid or duplicate gaps.  
  
3. Potential Missing Gap '1':  
In some cases buildGaps() might not include 1, leaving the array unsorted.  
  
4. Untracked Array Accesses:  
No pt.incArrayAccesses() calls implemented.  
  
5. Limited CLI Validation:  
Invalid arguments cause IllegalArgumentException.  
  
6. Non-Thread-Safe CSV Writing:  
Appending results concurrently may corrupt file.

## 6. Correctness Tests

JUnit Tests Implemented:  
- Empty array  
- Single element  
- Duplicates  
- Random input  
  
Missing Tests:  
- Reversed input  
- Nearly sorted input  
- Very large input (>10⁵ elements)  
- Comparison with Java’s Arrays.sort()  
  
All existing tests pass successfully.

## 7. Theoretical vs Empirical Behavior

|  |  |  |  |
| --- | --- | --- | --- |
| Aspect | Theoretical | Implementation | Match |
| Time Complexity | O(n^(3/2)) average | Matches for Knuth & Sedgewick | ✅ |
| Space Complexity | O(1) | In-place sorting | ✅ |
| Metric Tracking | Comparisons, swaps, etc. | Not implemented | ❌ |
| Stability | Not stable | Correct | ✅ |
| Gap correctness | Strictly decreasing | Incorrect for Sedgewick | ❌ |

## 8. Peer Code Review Summary

Strengths:  
- Clean modular design (three gap schemes, flexible configuration).  
- Correct sorting logic for all non-empty arrays.  
- Comprehensive CLI for batch benchmarking.  
- Proper JUnit integration for correctness validation.  
  
Weaknesses:  
- Incomplete metric instrumentation.  
- Incorrect or incomplete Sedgewick gap generation.  
- Lack of defensive argument validation in CLI.  
- Partial test coverage.

## 9. Optimization Suggestions

1. Add Metric Hooks: increment pt.incComparisons(), pt.incSwaps(), etc.  
2. Fix Sedgewick Gaps using proper even/odd formulas.  
3. Ensure all schemes always include gap=1.  
4. Add reversed and nearly sorted test cases.  
5. Handle invalid CLI parameters gracefully.  
6. Improve PerformanceTracker with separate read/write counts.

## 10. Conclusion

The Shell Sort implementation correctly sorts arrays using configurable gap sequences and loop styles.  
However, metrics are incomplete and gap generation (Sedgewick) is incorrect.  
Once fixed, the implementation will provide reliable empirical benchmarking data.

## 11. References

1. D. Shell, “A High-Speed Sorting Procedure,” Communications of the ACM, 1959.  
2. Sedgewick, R. (1986). “Analysis of Shellsort and Related Algorithms.”  
3. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms, MIT Press.  
4. Oracle Java SE 21 Documentation.  
5. Design and Analysis of Algorithms Lecture Notes – Astana IT University (2025).