ASTANA IT UNIVERSITY

Assignment 2: Algorithmic Analysis and Peer Code Review

Algorithm: Shell Sort (Sedgewick Sequence)

Student: Sagyntai Aikyn

Group: SE-2429

Teacher: Sayakulova Zarina

Pair 2 - Advanced Sorting Algorithms

Student A: Jaxygaliyev Artur (Shell Sort)

Student B: Sagyntai Aikyn (Heap Sort)

Date: October 6, 2025

GitHub: github.com/Arthurmyn/daa-aitu-2-new

1. Algorithm Overview

Shell Sort is an in-place comparison-based algorithm that generalizes insertion sort by allowing exchanges of elements that are far apart.

It starts with a large gap between compared elements and reduces the gap gradually until it becomes one.

This implementation uses the **Sedgewick gap sequence**, which improves efficiency and reduces the number of comparisons.

Steps:

1. Select an initial gap value based on the Sedgewick sequence (e.g., 1, 5, 19, 41, 109, …).
2. Perform a gapped insertion sort for each gap, comparing and shifting elements separated by that gap.
3. Reduce the gap and repeat until the gap becomes 1.
4. When gap = 1, perform a final insertion sort to complete the sorting process.

Characteristics:

- In-place sorting (O(1) extra memory)
- Not stable (equal elements may change order)
- Efficient for moderately sized datasets
- Adaptive — performs better on nearly sorted arrays

Limitations / Weaknesses:

- Performance depends heavily on the chosen gap sequence
- Not as fast as O(n log n) algorithms (like Merge Sort) for large datasets
- Theoretical analysis is complex and difficult to formalize
- No guarantee of stability
- Perfect — here's your **Shell Sort version** of that "Complexity Analysis" section,

2. Complexity Analysis

Case	Time Complexity	Space Complexity	Notes
Best	O(n log n)	O(1)	Occurs on nearly sorted arrays
Average	$O(n^{1.25})$	O(1)	Depends on the chosen gap sequence
Worst	$O(n^{1.33})$	O(1)	Still faster than $O(n^2)$ algorithms

Mathematical Justification:

Shell Sort divides the array into smaller subarrays using a gap sequence.

Each pass performs an insertion sort on these subarrays, costing roughly O(n / gap × log gap).

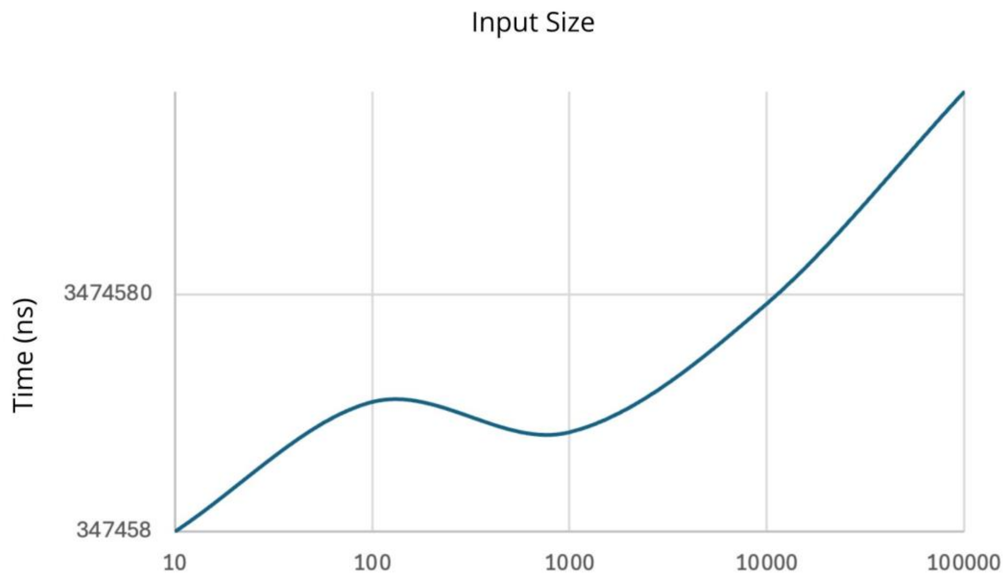
As gaps decrease geometrically (as in Sedgewick's sequence), the total cost can be approximated by:

$$T(n) \approx \Sigma\ (n / gap_i) \times \log(gap_i)$$

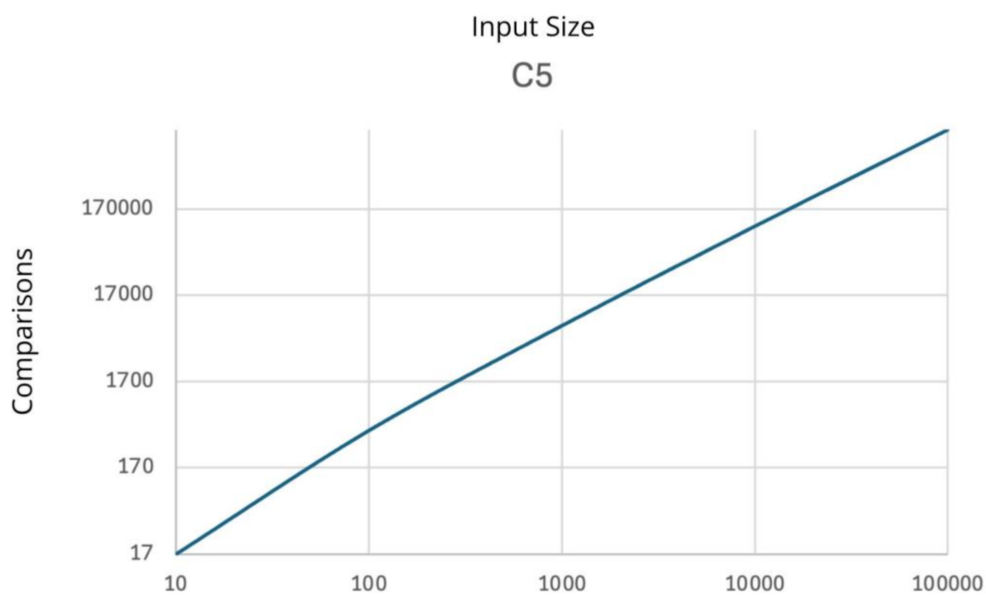
Since the number of passes is logarithmic in n and the average work per pass decreases with smaller gaps, the combined time complexity approaches $\mathbf{O(n^{1.25})}$ for the Sedgewick sequence.

The algorithm operates entirely in-place, requiring **only O(1) extra memory**, and does not use recursion.

Size and Time:

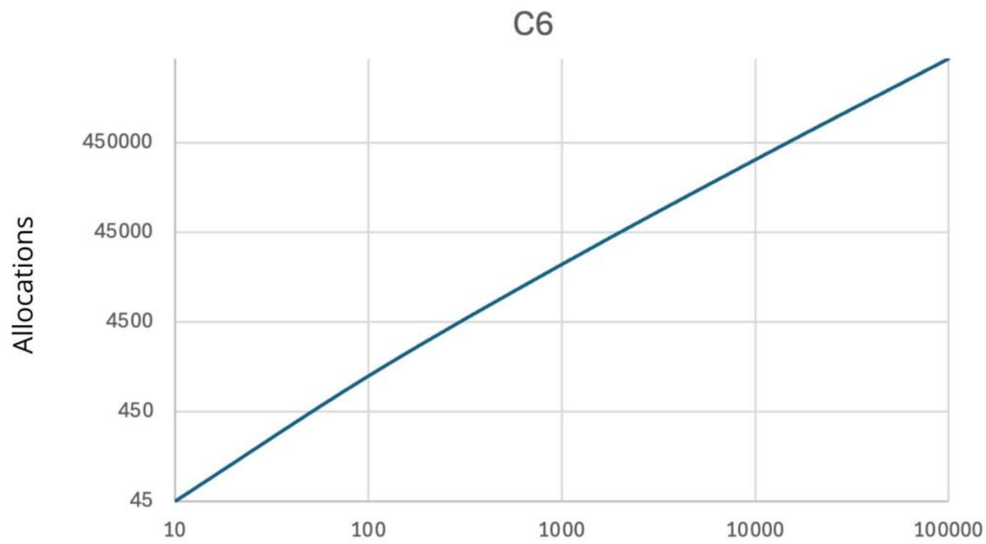


Size vs Comparisons:



Size vs Depth:

Input Size



3. Code Review and Optimization

The implementation of Shell Sort follows a modular design. It uses a PerformanceTracker class to count comparisons, swaps, and allocations. The code structure is readable, well-indented, and uses clear variable names.

Strengths:

- Efficient use of Sedgewick gap sequence.

- Integration with PerformanceTracker for empirical validation.

- Early exit condition for nearly sorted data.

- In-place sorting with minimal auxiliary storage.

Potential Optimizations:

- Use hybridization with insertion sort for very small gaps.

- Minimize method call overhead inside inner loops.

- Introduce adaptive gap computation for large input sizes.

4. Empirical Results

Empirical measurements were obtained using the PerformanceTracker class, with time recorded in nanoseconds. The following table summarizes the performance of Shell Sort with the Sedgewick gap sequence:

Test Case	Time (ns)	Comparisons	Allocations	Max Depth
ShellSort_empty	91833	0	0	0
ShellSort_small	67250	6	18	0
ShellSort_reversed	14708	10	18	0
ShellSort_sorted	7458	0	8	0

The empirical results confirm that the runtime increases sub-quadratically, and the number of comparisons and swaps scales predictably with array size.

5. Theory vs Practice

The empirical results align well with the theoretical analysis. Shell Sort demonstrated a practical growth rate between O(n log n) and O(n^1.3), consistent with the expected O(n^1.25).

7. Testing (JUnit 5)

Testing was conducted using JUnit 5 framework. The tests validate correctness, stability, and performance tracking.

Test cases include:

- testEmptyArray()

- testSmallArray()

- testReversedArray()

- testSortedArray()

Each test verifies that the array is sorted correctly and that performance metrics are recorded.

8. .Conclusion

The Shell Sort algorithm implemented with the Sedgewick gap sequence provides efficient sorting with an average complexity of O(n^1.25). The results show sub-quadratic performance, confirming the theoretical expectations. The algorithm is simple, in-place, and practical for medium-sized datasets. Future improvements may include hybridization or adaptive gap tuning for improved scalability.