# Individual Analysis Report: Shell Sort (Partner's Algorithm)

*Student B:* Balgatay Shynar (Group 2)

*Partner (Student A):* Zhandos Gilazhev

*Algorithm Under Review:* Shell Sort (Shell, Knuth, Sedgwick sequences)

*Comparison Algorithm:* Heap Sort (Bottom-Up Heapify)

*Date*: 05 October 2025

## 1. Algorithm Overview

Shell Sort is an in-place comparison-based sorting algorithm that generalizes insertion sort by allowing exchanges of elements that are far apart, using a sequence of gaps *(or increments)* to progressively sort the array. Introduced by Donald Shell in *1959*, it starts with large gaps and reduces them until the gap is 1, at which point it becomes a standard insertion sort. The key advantage is that it moves elements closer to their final positions early on, reducing the work for smaller gaps.

The performance of Shell Sort heavily depends on the choice of gap sequence. The provided implementations include three variants:

**Shell's Sequence:** Starts with gap = n/2 and halves it each time (gap /= 2). This is the original sequence, simple but can lead to inefficient performance in worst cases. Mathematically, the sequence is defined as $h_k = \lfloor n/2^k \rfloor$ for $k = 1, 2, \dots$ until $h_k = 1$.

**Knuth's Sequence:** Uses gaps of the form 3*gap + 1, starting from 1 and building up to the largest gap less than n/3, then divides by 3. Proposed by Donald Knuth, it aims for better distribution. The sequence is generated by $h_{k+1} = 3h_k + 1,$ with $h_1 = 1,$ up to $h_m < n/3.$

**Sedgwick's Sequence:** Generates gaps using the formula (4^k + 3*2^{max(k-1,0)} + 1), starting from k=0 (gap=5), and adds 1 if missing. This is a variant inspired by Robert Sedgewick's work to optimize average-case performance. The sequence in the code is approximately 1, 5, 8, 23, 77, 281, ..., which aligns closely with Sedgewick's 1982 proposal (1, 8, 23, 77, ...) but includes an extra 5 for small k.

Theoretically, better gap sequences reduce the number of comparisons and swaps by minimizing redundant operations across passes. All variants have **O(1)** space complexity since they are in-place, but time complexity varies. The benchmarks provided test these on random, sorted, reversed, and nearly sorted arrays of sizes 100 to 100,000.

## 2. Complexity Analysis

***Time Complexity Derivation***

Shell Sort's time complexity is analyzed by considering the number of operations across all gap passes. Each pass is similar to insertion sort on subarrays spaced by the gap, with cost proportional to the number of increments needed. The total time is the sum over all gaps $h_k$ of the cost for that pass, where each pass costs $O(n/h_k)$ subarrays, each taking $O(h_k)$ on average for shifts, but up to $O((n/h_k)^2)$ in worst case per group.

**Shell's Sequence (gaps: n/2, n/4, ..., 1):**

The number of gaps is **O(log n).** For each gap *h*, t$O(n^2/h).$ion sort pass costs $O(n)$ in the best case, but in the worst case (when elements require many shifts), it can be if subarrays are unsorted. Aggregating the worst-case is $O(n^2),$ as $\sum_{k=1}^{\log n} O(n^2/2^k) = O(n^2).$ Average case is empirically $O(n^{3/2})$ or $O(n \log^2 n),$ with lower bound $\Omega(n \log n).$

**Knuth's Sequence (gaps: 1, 4, 13, 40, ..., up to < n/3):**

Gaps are of the form $h_k = (3^k - 1)/2.$ The number of gaps is O(logn). Worst-case time is $O(n^{3/2}),$ derived from $T(n) = \sum O(n^{3/2}/h_k^{1/2}) \approx O(n^{3/2}).$ Average case O(n log n) but not tight.

**Sedgwick's Sequence (gaps: 1, 5, 8, 23, 77, ...):**

Gaps grow as $h_k \approx 4^k/3.$ Worst-case $O(n^{4/3}),$ from $T(n) = O(n) \sum_k (n/h_k)^{1/3} \approx O(n^{4/3}).$ Average $O(n^{7/6}).$

In all cases, best-case time is $\Omega(n \log n)$ (from information theory) when the array is nearly sorted, as fewer shifts occur. For sorted input, it's Θ(nlogn) due to comparisons in each pass: $\sum_k O(n/h_k) \cdot O(h_k) = O(n \log n).$

## 2.1 Space Complexity

All implementations are in-place, using **O(1)** auxiliary space (ignoring the input array). However, the Sedgwick variant uses an ArrayList to store gaps, which requires **O(log n)** space for the list itself. This is negligible but technically **O(log n)** extra space. No dynamic allocations occur during sorting (except ArrayList resizes), so practical space is constant.

**Comparison with Partner's Algorithm Complexity**

Assuming the *partner's algorithm* refers to the Sedgwick variant (as it differs most in implementation and has unique gap computation), we compare it to the others in the table below. Zhandos's implementation introduces a unique gap sequence with an initial gap of 5, differing from the standard Sedgewick sequence, which impacts its empirical performance.

| Variant | Best-Case | Average-Case | Worst-Case | Space |
|---|---|---|---|---|
| Shell | Θ(n log n) | O(n^{3/2}) | O(n^2) | O(1) |
| Knuth | Θ(n log n) | O(n log n) | O(n^{3/2}) | O(1) |
| Sedgwick | Θ(n log n) | O(n^{7/6}) | O(n^{4/3}) | O(logn) |

**Vs. Shell's:** Sedgwick has better theoretical worst-case **(O(n^{4/3}) vs. O(n^2))** and average **(O(n^{7/6}) vs. O(n^{3/2})),** due to gaps that are more coprime and grow faster, reducing overlapping subarray issues.

**Vs. Knuth's:** Sedgwick is superior in worst-case **(O(n^{4/3}) < O(n^{3/2})),** as its exponential base-4 growth covers more diverse spacings. Knuth's is **O(n^{3/2}),** but empirically similar for average cases. Sedgwick's bounds are tighter for large n, per Sedgewick's analysis.

**Overall,** no variant achieves **O(n log n)** worst-case like Merge/Heap Sort, but Sedgwick is closest to optimal among simple in-place sorts.

## 3. Code Review

*Identification of Inefficient Code Sections*

The codes are generally clean, but inefficiencies and bugs exist, particularly in metric tracking consistency:

**ShellSortShell.java:** Efficient core loop. Gap halving is simple (O(1) per update). However, the inner while loop breaks early on failed comparisons, which is good, but accesses and comparisons are incremented accurately.

**ShellSortKnuth.java:** Similar structure, with gap computation in a while loop (O(log n) time, fine). The loop is efficient, but starts gap=1 and builds up, then divides by 3—correct for Knuth's sequence.

**ShellSortSedgwick.java:**

- Gap generation uses ArrayList, which involves dynamic resizes (amortized O(1) per add, but O(log n) total). This is inefficient compared to computing gaps on-the-fly without storage, as the number of gaps is small (O(log n)).

- The formula uses Math.pow, which is floating-point and cast to int—potential precision issues for large k (though n<10^5 here). Better to use integer arithmetic (e.g., bit shifts for powers of 4).

- Inner loop: Comparisons are incremented only inside the loop (after condition true), missing the final failing comparison. This undercounts comparisons by ~1 per insertion, leading to inaccurate metrics (e.g., 0 comparisons for sorted arrays in CSV).

- Swaps: Extra `if (j != i) incSwaps()` after loop, which overcounts swaps (each shift already increments, and final assignment isn't a swap). Standard insertion counts shifts as swaps.

- Always adds 1 to gaps if missing, but sequence starts with 5, making it non-standard (standard omits 5).

*Specific Optimization Suggestions with Rationale*

**For All:** Standardize metric counting. Move incComparisons() before the if in inner loops, as in Shell/Knuth, to count all comparisons (including failures). Remove extra swap inc in Sedgwick—rationale: accuracy matches theoretical analysis, where comparisons = inversions + insertions.

**Sedgwick-Specific:**

- Compute gaps without ArrayList: Use a loop to find max k, then iterate downward in integer vars. Rationale: Reduces space to O(1), avoids allocations/resizes, faster for large n.
- Replace Math.pow with shifts/multiplies: e.g., $4^k = 1 \ll (2k); 2^{k-1} = 1 \ll (k-1)$ for k>0. Rationale: Avoids FP overhead, precision safe.
- Remove `if (j != i) incSwaps()`: It's redundant; shifts already counted. Rationale: Prevents overcounting, aligns with other impls (e.g., CSV shows inflated swaps for Sedgwick).

**General:** Add early exit if array sorted after a pass (check zero swaps), but rare benefit. Use sentinels for inner loop? Complicates code without big gains.

*Proposed Improvements for Time/Space Complexity*

- Time: For Sedgwick, on-the-fly gap computation keeps O(n^{4/3}) but reduces constants (no list overhead). Overall, no change to asymptotic time, but practical speedup ~5-10% for gap gen.
- Space: Eliminate ArrayList in Sedgwick → true O(1) extra space.
- To improve further: Adopt a better sequence like Ciura's empirical (1,4,10,23,57,132,...), which can approach O(n log n) average, but requires research.

## 4. Empirical Results

*Performance Plots (Time vs Input Size)*

The provided plots (log-log scale) show time (ms) vs array size n (10^2 to 10^5) for four input types:

- **Random:** Sedgwick (green) scales worst, reaching ~10^3 ms at n=10^5, while Shell (blue) and Knuth (orange) are ~10^1-10^2 ms, with Knuth slightly better. Slope suggests Sedgwick ~O(n^{1.8}), higher than theory—likely due to extra gaps/overhead.
- **Reversed:** Similar, Sedgwick explodes to ~10^4 ms, Shell/Knuth lower (~10^2), validating Shell's O(n^2) vulnerability but Sedgwick worse empirically (buggy gaps?).
- **Sorted:** All low (<1 ms), but Shell highest (~10^{-1}), Knuth middle, Sedgwick near 0—undercounting hides true cost, but actual time minimal as no swaps.
- **Nearly Sorted:** Sedgwick again highest, but closer; Knuth best.

*Validation of Theoretical Complexity*

From CSV: For n=100,000 random, Sedgwick has ~2.5B comparisons/swaps (explains slow time), vs. Shell's 4M, Knuth's 3.7M. Accesses similar scale. Below is a summary table of time (ms) for n=100,000 across types:

| Type | Shell (ms) | Knuth (ms) | Sedgwick (ms) |
|---|---|---|---|
| Random | 17 | 14 | 5179 |
| Sorted | 6 | 4 | 4 |
| Reversed | 9 | 6 | 9750 |
| Nearly Sorted | 10 | 9 | 590 |

- **Shell:** Random/reversed show ~n^{1.5-1.8} slope, matching average O(n^{3/2}), worst O(n^2) for reversed.
- **Knuth:** Flatter slope ~n^{1.3}, close to O(n^{3/2})=n^{1.5}, better than Shell.
- **Sedgwick:** Empirical ~n^{1.8-2}, worse than theory (O(n^{4/3})~n^{1.33})—discrepancy due to metric bugs (undercounting comparisons but high actual ops from extra gaps like 5) and small n. For larger n, theory predicts overtake.

Constant factors: Sedgwick has high constants from list/FP ops; Knuth lowest.

Practical: Knuth best for tested n, Sedgwick worst—contrary to theory, suggesting implementation flaws.

*Analysis of Constant Factors and Practical Performance*

Constants dominate for n<10^5: Knuth's simple gaps minimize overhead. Sedgwick's list and pow calls add ~10-20% time. In CSV, time correlates with comparisons/swaps, not accesses (similar). For nearly sorted, all efficient, but reversed punishes poor gaps. Overall, empirical validates theory partially, but bugs inflate Sedgwick's perceived cost.

## 5. Conclusion

The analysis reveals Shell Sort variants with varying efficiencies: Knuth performs best empirically, while Sedgwick's theoretical superiority is undermined by implementation issues like inaccurate metrics, unnecessary allocations, and a slightly non-standard sequence. Optimizations—standardizing counters, removing ArrayList, integer gap computation—would align Sedgwick closer to O(n^{4/3}). Recommend adopting Knuth for practical use or fixing Sedgwick for large n. Further testing on n>10^6 could validate theory.