INDIVIDUAL ANALYSIS REPORT

Algorithm overview:

Shell sort.

Brief description:

Shell Sort is improved version of Insertion Sort. In insertion sort, we move elements only one position ahead. Many movements involved. Therefore, shell sort generalizes insertion to improve its efficiency for larger, unsorted arrays.

It works by allowing exchanges of elements that are far apart, to move element closer to their correct position faster.

The algorithm was invented by Donald Shell in 1959.

Shell sort idea:

1. Divide an array into subarrays using a gap value.
2. Performing insertion sort on elements which are gap apart.
3. Reducing the gap until it becomes 1.

Theoretical background:

The performance of shell sort depends on the choice of the gap sequence.

Commonly Shell's original sequence: n/2, n/4…;

Hibbard's sequence: 1, 3, 7,15…;

Knuth's sequence: 3k+1;

Complexity analysis:

Time complexity:

Best time complexity:

O(n logn)

For an optimal sequence, like Hibbard sequence, shell sort can achieve a best-case time complexity of O(n logn). When array is nearly sorted, the operations are minimal.

Worst time complexity:

O(n^2)

For certain interval sequences, like original shell sequence, the algorithm performs poorly, with a time complexity of O(n^2).

Average time complexity:

O(n^(3/2)) to O(nlogn)

Common sequences like Knuth.

Space complexity:

Space complexity of Shell sort is O(1).

Shell sort is in-place algorithm requiring no additional memory.

Space: Θ(1)

Best case: Ω(n)

Worst case: O(n^2)

Partner code:

My partner used three types of gap sequences:

Shell sequence (n/2, n/4, …, 1)

Knuth sequence (1, 4, 13, 40, …)

Sedgewick sequence (1, 5, 19, 41, …)

Shell's original sequence:

Worst Case: O(n^2)
Average Case: Θ(n^1.5)
Best Case**:** Ω(n log n)

Knuth's sequence:

Worst Case: O(n^1.5)
Average Case: Θ(n^1.4)
Best Case**:** Ω(n log n)

Sedgewick sequence:

Worst Case: O(n^1.33)
Average Case: Θ(n^1.25)
Best Case**:** Ω(n log n)

Space complexity: O(1).

Code review:

The given implementation correctly follows the Shell Sort algorithm and supports three different gap sequences — Shell, Knuth, and Sedgewick.
It also uses a PerformanceTracker to measure comparisons, swaps, and array accesses, which is useful for benchmarking and complexity observation.

```
switch (sequence.toLowerCase()) {
    case "knuth":
        gaps = knuthSequence(n);
        break;
    case "sedgewick":
        gaps = sedgewickSequence(n);
        break;
    default:
        gaps = shellSequence(n);
}
```

Each time sort() is called, the algorithm rebuilds the entire gap array**.**
For large arrays or repeated sorting (as in benchmarking), this causes log n-level recomputation overhead that could be avoided by precomputing or caching.

```
 list.add(0, h);
```

This operation shifts all existing elements each time it runs (O(n) per insert).

For large n, this makes the sequence construction $O(n^2)$ instead of $O(n)$.

Fix: Use list.add(h) then reverse once, prevents $O(n^2)$ behavior in sequence creation.

Instead of hardcoding three types, dynamically pick the sequence based on n:

```
if (n < 5000) gaps = knuthSequence(n);

else gaps = sedgewickSequence(n);
```

Rationale: For smaller arrays, Knuth’s gaps perform well (less overhead); for large arrays, Sedgewick’s gaps minimize comparisons.

Memory optimization:

Reusing a single buffer to avoid repeated allocations.

```
private static final int[] GAP_BUFFER = new int[32];
```

Summary:

The code is correct, well-structured and instrumented for performance tracking, but can be improved. These changes will be more scalable, better when there will be large lists and so on.

Empirical results:

Performance plots:

Based on the benchmarking setup in BenchmarkRunner.java, the algorithm was tested on input sizes n = {100, 1,000, 10,000, 100,000} across three gap sequences: Shell, Knuth, and Sedgewick. The empirical runtime (in milliseconds) was recorded using System.nanoTime().

```
=== Benchmarking Shell Sort (shell sequence) ===
n=100 | time=0,424 ms | comps=856 | swaps=400 | accesses=1406
n=1000 | time=0,903 ms | comps=14736 | swaps=7225 | accesses=23237
n=10000 | time=5,200 ms | comps=262768 | swaps=147870 | accesses=387880
n=100000 | time=13,417 ms | comps=4113987 | swaps=2664662 | accesses=5664674

=== Benchmarking Shell Sort (knuth sequence) ===
n=100 | time=1,635 ms | comps=788 | swaps=491 | accesses=1175
n=1000 | time=0,164 ms | comps=13583 | swaps=8577 | accesses=19491
n=10000 | time=1,640 ms | comps=234602 | swaps=163785 | accesses=314271
n=100000 | time=11,050 ms | comps=3527091 | swaps=2601343 | accesses=4535635

=== Benchmarking Shell Sort (sedgewick sequence) ===
n=100 | time=0,319 ms | comps=764 | swaps=479 | accesses=1147
n=1000 | time=0,084 ms | comps=13349 | swaps=7689 | accesses=20053
n=10000 | time=0,871 ms | comps=195105 | swaps=106956 | accesses=293330
n=100000 | time=10,548 ms | comps=2422352 | swaps=1207819 | accesses=3740075
```

All three sequences show an approximately linear increase in runtime with n^1.3–n^1.5 growth behavior.

The Sedgewick sequence maintains the lowest curve, followed by Knuth, while Shell’s original sequence is the slowest.

The empirical growth patterns **confirm the theoretical models**.

Analysis of Constant Factors and Practical Behavior

Comparisons & Swaps:
The total number of comparisons and swaps grows at roughly the same rate as runtime. Sedgewick's gaps yield the fewest data movements — around 40% fewer swaps than the base Shell sequence at n = 100,000.

Memory Accesses:
Memory access counts also align with gap efficiency — fewer passes through the array for Sedgewick, leading to better cache locality and reduced CPU stalls**.**

Instrumentation Overhead:
The PerformanceTracker class introduces consistent overhead, but since all runs use it, relative performance differences remain accurate.

Summary:

Sedgewick's sequence delivers fastest runtime and fewest operations.

Measured results validate theory: both mathematically and practically justified.

Conclusion

The analysis of the implemented Shell sort algorithm which is tested with three gap sequences reveals several key findings about its efficiency, performance and scalability.

The algorithm overall follows the shell sort principles: sorting elements that are far apart and gradually reducing the gap until full ordering. The sorting algorithm implemented correctly with good handling of comparisons, swaps and data movements.

The results align closely with theoretical predictions.

Optimization recommendations:

Dynamically choose best gap sequence: Sedgewick for large, Knuth for moderate, Shell for small.

Memory optimization: caching, cpu.

Shell sort remains strong, space efficient and easy algorithm for medium-sized datasets.