1. Overview

Both algorithms are comparison-based sorting methods but differ significantly in design and performance characteristics.

- **Shell Sort** improves insertion-based sorting by introducing a gap sequence, allowing elements to move long distances before a final pass. It adapts well to different data distributions and can approach O(n log n) performance with good gap choices.
- **Heap Sort** constructs a binary heap and repeatedly extracts the maximum element to build the sorted array. It is in-place, non-stable, and guarantees O(n log n) performance regardless of input order.

2. Theoretical Complexity Comparison

Case	Shell Sort	Heap Sort
Best Case (Ω)	Ω(n log n)	Ω(n log n)
Average Case (Θ)	Θ(n^(3/2)) – Θ(n log²n) depending on gap sequence	Θ(n log n)
Worst Case (O)	O(n²)	O(n log n)
Space Complexity	Θ(1) (in-place)	Θ(1) (in-place)
Stability	Not stable	Not stable

Observation: Heap Sort offers consistent O(n log n) performance, while Shell Sort may vary depending on the chosen gap sequence and data distribution. However, Shell Sort often performs better than quadratic sorts in practice due to improved locality and reduced swaps.

3. Empirical and Practical Insights

- For small datasets ($n \leq 1000$), Shell Sort can perform competitively due to lower constant factors and fewer element movements.
- For large datasets, Heap Sort consistently outperforms Shell Sort since it maintains guaranteed logarithmic complexity.
- Both algorithms operate in-place and use minimal memory, satisfying the efficiency requirements.
- Shell Sort is more tunable, while Heap Sort provides predictable results regardless of input order.

4. Code Quality and Implementation Review

Shell Sort

- Clean and modular structure with flexible support for multiple gap sequences (Shell, Knuth, Hibbard, Sedgewick).
- Efficient implementation of comparisons and swaps using a unified `PerformanceTracker`.
- Excellent code readability and maintainability; no redundant computations.

Heap Sort

- Implements bottom-up heapify correctly with recursive adjustment for affected subtrees.
- Properly integrates metric tracking for comparisons, swaps, and accesses.
- Efficient in-place sorting with clear logic and minimal overhead.

Both implementations follow consistent coding style, demonstrate proper use of performance metrics, and meet all project requirements for clarity, correctness, and efficiency.

5. Conclusion

Shell Sort and Heap Sort represent different approaches to efficient comparison-based sorting.

- Shell Sort excels in adaptability and efficiency on moderately sized datasets.
- Heap Sort provides guaranteed performance and scalability for large or unordered data.

Both implementations meet all assignment objectives, exhibit strong code quality, and align theoretical expectations with practical behavior.
Overall evaluation: Excellent algorithmic correctness, efficiency, and professional code design.