**Cross-Review Summary Report**

**1. Overview of Algorithms**

**Heap Sort (implemented by Aiya Zhakupova):**

* **Purpose:** Sorts an array by repeatedly extracting the maximum element from a binary heap.
* **Approach:** Builds a max-heap (bottom-up heapify), then performs n extract-max operations.
* **Properties:** Deterministic, in-place algorithm with predictable runtime.
* **Applications:** Large-scale data sorting when stability is not required and memory is limited.

**Shell Sort with Knuth Sequence (implemented by Nurdan Zhanabayev):**

* **Purpose:** Sorts an array using diminishing gap insertion sorts.
* **Approach:** Starts with a large gap (Knuth sequence), progressively reducing until gap = 1 (regular insertion sort).
* **Properties:** In-place, empirically faster than quadratic sorts, but worst-case performance is weaker than Heap Sort.
* **Applications:** Teaching algorithm, useful for moderately sized arrays and partially sorted data.

**2. Complexity Analysis**

**Heap Sort:**

* Worst Case: **O(n log n)** (each of n extractions costs log n).
* Best Case: **Ω(n log n)** (even for nearly sorted arrays).
* Average Case: **Θ(n log n)**.
* Space: **O(1)** auxiliary, but **O(log n)** call stack with recursive heapify.

**Shell Sort (Knuth):**

* Worst Case: **O(n²)** (gap sequence may degrade to quadratic).
* Best Case: **Ω(n log n)** (partially sorted or favorable inputs).
* Average Case: **Θ(n^1.5)** (empirical and theoretical results).
* Space: **O(1)** auxiliary variables.

**Comparison:**

* Heap Sort has **theoretical superiority** with guaranteed O(n log n) in all cases.
* Shell Sort is less predictable: faster on some inputs (due to cache locality) but weaker asymptotically.
* Both are in-place, but Heap Sort sacrifices cache performance for guaranteed bounds, while Shell Sort benefits from subarray locality.

**3. Code Review**

**Heap Sort Issues:**

* Recursive heapify introduces unnecessary stack frames.
* Redundant array access tracking inflates metrics.
* Comparisons miscounted during swaps.
* **Optimizations:** Iterative heapify, refined metrics tracking, and careful instrumentation would improve performance clarity.

**Shell Sort Issues:**

* Redundant swaps increase cost.
* Array access double-counting in metrics.
* Fixed Knuth gaps limit adaptability.
* Coarse timing (ms resolution) obscures performance on small arrays.
* **Optimizations:** Use adaptive gap sequences (Tokuda, Sedgewick), optimize swaps, and refine benchmark timing to microseconds.

**4. Empirical Results**

**Heap Sort Benchmarks (n = 10,000):**

* ~349,000 comparisons, ~124,000 swaps, ~853,000 accesses.
* Times ranged 2–5 ms due to JVM and OS noise.
* Performance scales as n log n, independent of input distribution.

**Shell Sort Benchmarks (n = 100):**

* ~700 comparisons, ~400 swaps, ~2,200 accesses.
* Runtime often 0–1 ms (small inputs + coarse timing).
* Larger input sizes (not shown) scale closer to n^1.5.

**Comparison:**

* Heap Sort: predictable, stable runtime across distributions.
* Shell Sort: more variance; faster on small/moderate sizes but grows slower at large scales.
* Cache locality favors Shell Sort for moderate inputs, but Heap Sort dominates asymptotically.

**5. Conclusion**

* **Heap Sort:** Robust, predictable, asymptotically optimal (Θ(n log n)), minimal memory use. Main drawback is recursive overhead and less cache efficiency.
* **Shell Sort:** More practical on small/medium datasets with favorable cache performance but weaker theoretical guarantees (Θ(n^1.5) average, O(n²) worst).
* **Overall:** Heap Sort is preferred for large-scale, performance-critical tasks; Shell Sort is educational and useful for moderate or partially sorted datasets.

**Optimization Recommendations:**

* Replace recursive heapify with iterative implementation.
* Improve metrics tracking for accuracy.
* Enhance Shell Sort with adaptive gap sequences and finer benchmark timing.
* Use empirical validation (plots of time vs input size) to confirm theoretical bounds.