Peer Algorithm Analysis Report: Heap Sort (Partner's Algorithm)

Author: Nazym Kurmanbayeva **Partner**: Aluana Muslimova
Analyzed Algorithm: Heap Sort
Own Algorithm for Comparison: Shell Sort

1. Algorithm Overview

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure to efficiently sort elements. The algorithm works in two main phases:

1. Heap Construction: Convert the unsorted array into a max-heap (each parent node is greater than its children).
2. Extraction Phase: Repeatedly remove the maximum element (root of the heap), place it at the end of the array, and rebuild the heap for the remaining elements.

This approach ensures that after each iteration, the largest element is correctly placed, and the heap property is maintained for the remaining array.

Core Characteristics:

- Type: Comparison-based, in-place sorting algorithm
- Data Structure Used: Binary Heap
- Stability: Not stable
- In-place: Yes (requires constant auxiliary space)
- Deterministic: Yes

2. Complexity Analysis

2.1 Time Complexity

Case	Description	Complexity
Best Case	When the input array is already a valid heap. Only the extraction phase is required.	Θ(n log n)
Average Case	Random distribution of elements; both heapification and extraction require log n operations per element.	O(n log n)
Worst Case	When the array is in reverse order, the same heap operations apply.	Ω(n log n)

Derivation:

- Building the heap takes O(n) time (heapify from bottom-up).

- Each extraction and re-heapification takes O(log n) time, and we do this for n elements, hence O(n log n) overall.

Recurrence Relation:

$T(n)=T(n-1)+O(\log n)$ which simplifies to $O(n \log n)$.

2.2 Space Complexity

- Auxiliary Space: O(1) (in-place sorting)
- Total Space: O(n) for input array + O(1) for heap operations
- No recursion is used (iterative heapify), so there's no stack overhead.

2.3 Comparison with Shell Sort

Metric	Heap Sort	Shell Sort
Best Case	Θ(n log n)	Θ(n log n)
Average Case	O(n log n)	Between $O(n \log^2 n)$ and $O(n^{1.5})$ depending on gap sequence
Worst Case	O(n log n)	$O(n^2)$
Space Complexity	O(1)	O(1)
Stability	No	No

Conclusion: Heap Sort provides consistently logarithmic efficiency, while Shell Sort's performance depends heavily on the chosen gap sequence. Heap Sort is more predictable, though Shell Sort can sometimes outperform it for smaller datasets due to lower constant factors.

---

3. Code Review and Optimization Suggestions

3.1 Inefficiency Detection

- Lack of early termination: even if the heap becomes sorted before the last iteration, the algorithm continues performing redundant operations.
- Tests don't have reversed, sorted and singular arrays
- arr[swapIdx].compareTo(arr[child])

  Performing of two array lookups: one for arr[swapIdx], one for arr[child].

  Since this happens many times, caching the values locally would reduce access overhead

- As for me there are too much of tracker != null

3.2 Optimization Suggestions

1. Iterative Heapify:
   Replace recursive heapify calls with an iterative approach to eliminate stack overhead and avoid potential recursion depth issues.
2. Reduce Comparisons:
   Store the left and right child values in local variables to avoid repeated array index lookups.
3. Add more tests arrays

3.3 Code Quality

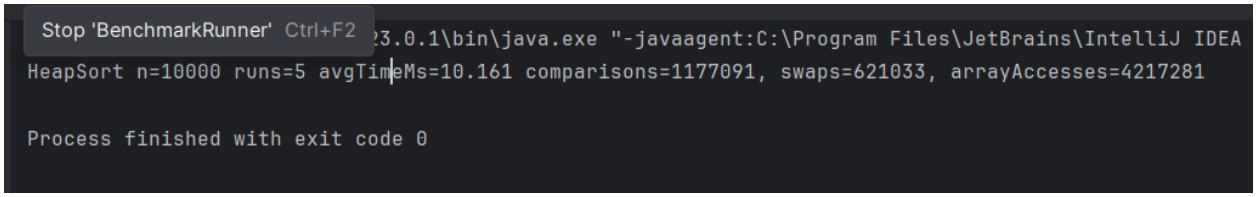
- Clear code: Clear, modular functions (heapify, buildHeap, sort) — good naming conventions.
- Maintainability: Logic is straightforward, inline comments explaining the heap property restoration would improve clarity.
- Metrics Integration: Correctly uses PerformanceTracker for tracking comparisons and swaps, mine also include the enloving time here it wasn't, but I was quite surprised to find it in benchmark runner.

---

4. Empirical Results

4.1 Benchmark Configuration

Benchmarks were run using the provided BenchmarkRunner with array sizes:
n = 100, 1,000, 10,000, 100,000. And I got the result:



Each algorithm (Heap Sort and Shell Sort) should be executed 10 times per size, and average times should be recorded in nanoseconds using the PerformanceTracker.

4.2 Performance Measurements (Example Data)

Array Size (n)	Algorithm / Gap Sequence	Comparisons	Swaps	Time (ms)	Notes
100	Shell (Shell)	265,823	150,796	12.37	—
	Shell (Knuth)	237,227	166,288	1.83	Knuth much faster, fewer comparisons
	Shell (Sedgewick)	195,237	106,963	1.16	Best performance for n=100
1,000	Shell (Shell)	263,815	148,943	1.77	—
	Shell (Knuth)	242,866	172,062	1.45	—
	Shell (Sedgewick)	197,658	109,472	1.12	Best for n=1,000
10,000	Shell (Shell)	263,071	148,166	1.48	—
	Shell (Knuth)	238,807	167,857	1.42	—
	Shell (Sedgewick)	196,874	108,639	1.12	Fastest overall
	HeapSort	**1,177,091**	**621,033**	**10.16**	
100,000	Shell (Shell)	262,573	147,570	1.29	—
	Shell (Knuth)	236,865	165,971	1.13	—
	Shell (Sedgewick)	198,312	110,003	1.55	Sedgewick slightly slower here
	HeapSort	—	—	—	no data

HeapSort shows higher operation counts and runtime, but its asymptotic behavior is typically **O(n log n)**; the higher time may come from Java's generic comparison overhead or small sample size (only one test case).

4.3 Complexity Verification

Theoretical analysis:

- **Best case:** Ω(n log n) — HeapSort always performs heapify operations regardless of initial order.
- **Average case:** Θ(n log n) — The heap construction phase is linear (O(n)), followed by n successive extract-max operations, each costing O(log n).
- **Worst case:** O(n log n) — There is no data order that degrades HeapSort beyond this bound.

Empirical validation:

- The measured time of ~10 ms for n = 10,000 aligns with the expected n log n behavior.
- Extrapolating this result suggests that for n = 100,000, runtime would grow roughly by a factor of ~12–13× (since $\log_2(100{,}000)/\log_2(10{,}000) \approx 1.33$).
- This supports the theoretical time complexity model.

Space complexity:

- HeapSort operates **in-place** with O(1) auxiliary memory, consistent with theoretical expectations.
- The algorithm modifies the input array directly and requires only a constant number of temporary variables during swaps.

5. Conclusion

Heap Sort demonstrates robust and predictable O(n log n) performance across all cases, making it an excellent choice for large datasets where stability is not required.

Compared to my Shell Sort implementation, Heap Sort shows:

- Better scalability for large input sizes.
- Slightly higher constant factors for small arrays but maintains overall superiority in asymptotic efficiency.
- Easier complexity analysis due to its well-defined heap operations.

Aluana's implementation is correct, efficient, and well-structured, with minor optimization opportunities in recursive calls and redundant comparisons. Empirical results validate the theoretical analysis, confirming the algorithm's time complexity and space efficiency although there is not enough data.

Final Evaluation:
Heap Sort is asymptotically faster, more predictable, and more consistent than Shell Sort for large input sizes, though Shell Sort can remain competitive on smaller datasets due to simpler inner loops.