Peer Analysis Report - Heap Sort

1. Introduction

This report analyzes the Heap Sort implementation that uses bottom-up heapify and integrated performance metrics. The purpose of the analysis is to evaluate the algorithm's asymptotic complexity, space usage, potential inefficiencies, and optimization opportunities. The study also validates theoretical predictions with empirical benchmarks on arrays of sizes ranging from 100 to 100,000 elements across various input distributions (random, sorted, reverse-sorted).

2. Theoretical Analysis of Heap Sort

Worst Case: O(nlogn) (always requires log-time per extraction).

Best Case: Omega(n log n) (even if array is nearly sorted, the algorithm performs the same sequence of heapify calls).

Average Case: Theta(n log n)
Heap Sort is deterministic and does not vary with input distribution, unlike QuickSort or Shell Sort.

3. Memory Usage

Heap Sort is an in-place algorithm:

Auxiliary space: O(1)O(1)O(1). Only a constant number of temporary variables (e.g., temp for swapping).

Call stack: Recursive heapify introduces O(log n)O(\log n)O(logn) stack frames in the worst case.

Optimization: The recursive call can be converted to an iterative loop to eliminate stack overhead entirely.

4. Empirical Results (Benchmarks)

Example for N=10000

```
algorithm=heap trial=0 n=10000 time=5ms comps=349481 swaps=124177 accesses=853136
algorithm=heap trial=1 n=10000 time=2ms comps=349432 swaps=124150 accesses=853011
algorithm=heap trial=2 n=10000 time=2ms comps=349516 swaps=124168 accesses=853197
```

Trial	n	Time (ms)	Comparisons	Swaps	Array Accesses
0	10000	5	349,481	124,177	853,136
1	10000	2	349,432	124,150	853,011
2	10000	2	349,516	124,168	853,197

Observations

The number of comparisons and accesses grows close to nlog⁡nn \log nnlogn, confirming theoretical predictions.

Time variance (2–5 ms) comes from JVM effects and OS scheduling noise.

Heap Sort consistently performs the same work regardless of input order → predictable runtime.

5. Identified Inefficiencies

- Recursive heapify:

Each recursive call pushes a new frame. On large arrays, this adds overhead.

JVM does not guarantee tail-call optimization.

- Redundant Array Access Tracking:

In metrics instrumentation, some increments are double-counted (e.g., reading both arr[l] and arr[largest] before the comparison).

- Comparison Counting in Swap Section:

Incrementing comparisons during swap is misleading, since swap is not a comparison operation.

6. Conclusion

Heap Sort is a robust and predictable sorting algorithm with consistent Theta(n log n) performance across all input types. Its in-place nature ensures minimal memory usage, though recursive heapify introduces avoidable stack overhead.
By switching to an iterative heapify and refining metrics tracking, the implementation can be optimized for both performance and clarity. While Shell Sort has interesting empirical behavior, Heap Sort remains more reliable for large-scale sorting tasks.