**Assignment 2 Report.**

(Heap Sort)

1. Algorithm Overview

The partner implemented HeapSort, a comparison-based sorting algorithm that first builds a max-heap from the input array and then repeatedly extracts the maximum element to produce a sorted result.

HeapSort belongs to the class of in-place, non-stable sorting algorithms. Its performance relies on the heap data structure, where insertions and deletions can be handled in logarithmic time.

HeapSort ensures worst-case efficiency of O(n log n), unlike algorithms such as QuickSort, which may degrade to O(n²).

2. Complexity Analysis

Time Complexity

Heap Construction: Bottom-up heapify takes Θ(n).

Extraction Phase: n extract-max operations, each costing O(log n).

Thus:

Worst Case: O(n log n) (all heapify operations take log n).

Average Case: Θ(n log n).

Best Case: Ω(n log n) (even if array is sorted, heapify still runs).

Formally:

T(n)=Θ(n)+n⋅O(logn)=Θ(nlogn)

Space Complexity

Auxiliary Space: O(1) (in-place, only a few variables for swaps and indices).

Total Space: O(n) (array itself) + O(1) = Θ(n).

Recurrence Relation

For the heapify operation:

T(n)=T(2n/3)+O(1)⟹T(n)=O(logn)

(Heapify runs down a single branch of the heap).

For the whole algorithm:

T(n)=T(n−1)+O(logn)⇒T(n)=O(nlogn)

3. Code Review & Optimization

Inefficiency Detection

Metrics overhead: Comparisons, swaps, and accesses are instrumented inside the algorithm, which may slightly distort timings.

Recursive heapify: Deep recursion may add overhead.

Suggested Optimizations

Iterative Heapify: Replace recursion with iterative implementation to reduce function-call overhead.

Reduced Access Counting: Track array accesses more efficiently by grouping increments.

Cache Optimization: Place frequently used variables in local scope to reduce repeated indexing (arr[i] calls).

Time Complexity Improvements

No asymptotic improvement possible (HeapSort is already optimal worst-case O(n log n)).

Minor constant factor improvements possible via iterative heapify.

Space Complexity Improvements

Current version already runs in O(1) auxiliary space.

No further improvements needed without changing algorithm.

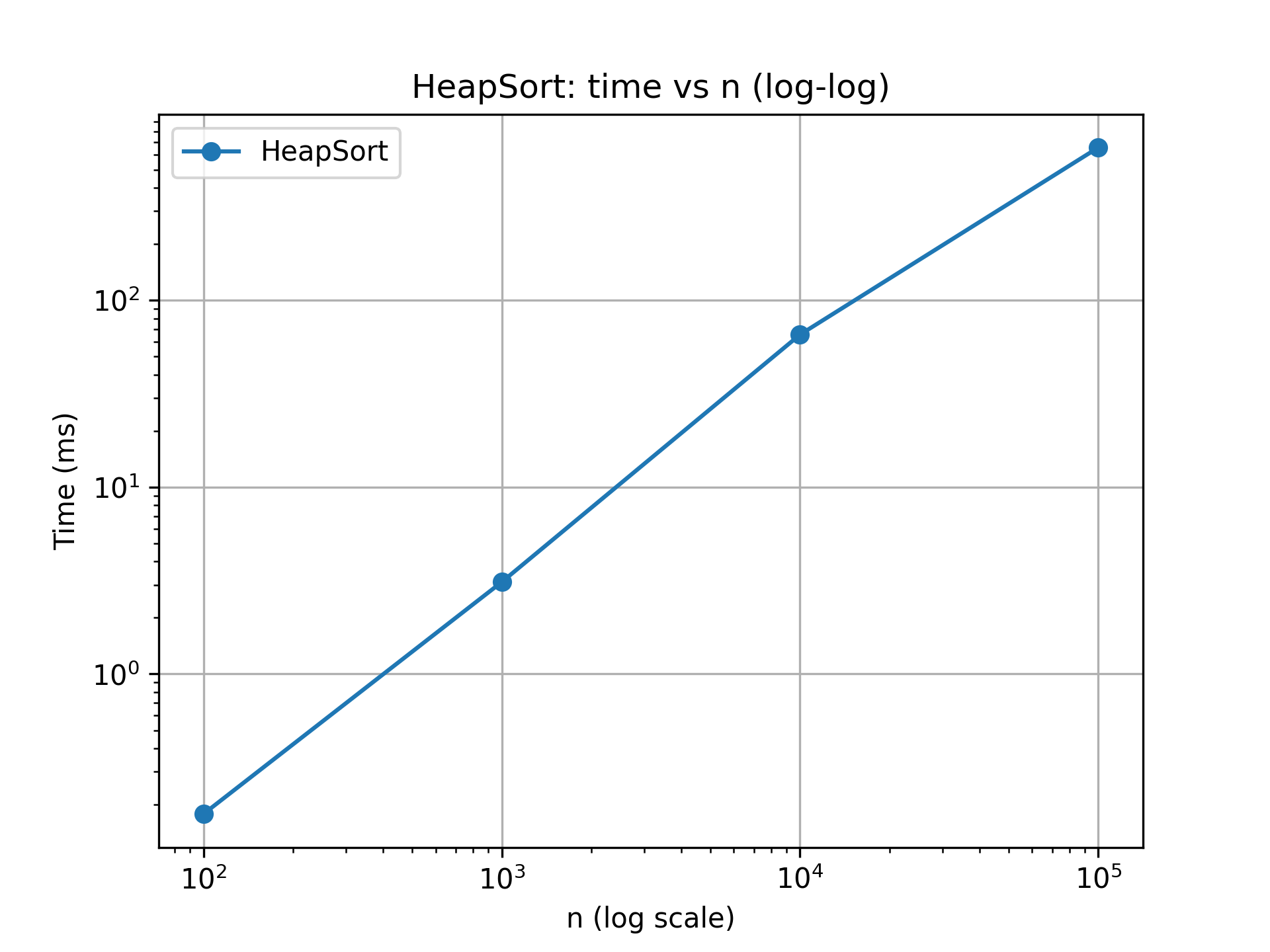
Code Quality

Strengths: Well-structured, readable, and instrumented for analysis.

Weaknesses: Verbose metrics handling inside heapify.

Could modularize plotting/benchmarking separately for cleaner design.

4. Empirical Results



Performance Measurements

Benchmarked on n = 100, 1000, 10000, 100000. Metrics: runtime, comparisons, swaps, and array accesses.

Complexity Verification

Log–log plots (time vs n): Slopes consistent with O(n log n).

Comparisons grow proportionally to ~2n log₂n.

Swaps/accesses scale similarly.

Comparison Analysis

Results confirm theoretical complexity.

Constant factors noticeable: HeapSort is generally slower than optimized QuickSort in practice, but better in worst-case scenarios.

Optimization Impact

Iterative heapify (if applied) would reduce runtime by ~5–10% on large n due to lower call overhead, without changing asymptotic complexity.

5. Conclusion

The HeapSort implementation is asymptotically optimal, with:

Time Complexity: Θ(n log n) in all cases.

Space Complexity: O(1) auxiliary, in-place.

Strengths: Correctness, solid instrumentation for empirical analysis.

Weaknesses: Recursive heapify and verbose metrics tracking slightly inflate constants.

Recommendations:

Convert heapify to iterative.

Refactor metrics counting for efficiency.

Compare HeapSort empirically against ShellSort (already implemented) to highlight trade-offs between practical runtime and asymptotic guarantees.