Algorithmic Analysis Report: Max-Heap Implementation

Algorithm Overview
The Max-Heap is a binary tree-based data structure that maintains the heap property, where each parent node is greater than or equal to its children. It is typically represented as an array where, for each element at index i, its children are located at indices 2i + 1 and 2i + 2. The Max-Heap supports efficient insertion, extraction of the maximum element, and heap construction operations. It is a foundational structure used in *Heap Sort* and priority queues.

Complexity Analysis
Theoretical complexity analysis of the Max-Heap considers time and space across its key operations. The notations Θ, O, and Ω are used to represent average, worst, and best-case scenarios respectively.

Operation	Best Case (Ω)	Average Case (Θ)	Worst Case (O)	Explanation
Build Heap	Ω(n)	Θ(n)	O(n)	Bottom-up heap construction runs in linear time.
Insert	Ω(1)	Θ(log n)	O(log n)	Percolation may occur up to the root in the worst case.
Extract-Max	Ω(log n)	Θ(log n)	O(log n)	Removal of root requires down-heapify operation.
Increase-Key	Ω(1)	Θ(log n)	O(log n)	May bubble up depending on key change.
Get-Max	Ω(1)	Θ(1)	O(1)	Root access operation with constant time.

Space Complexity
The Max-Heap implementation uses **O(n)** total space to store *n* elements. Since it operates directly on the array and uses only a few auxiliary variables for swapping and indexing, its **auxiliary space** is **O(1)**.

Code Review & Optimization
Upon reviewing the partner's Max-Heap code, the implementation follows standard heap operations effectively but could be improved in several aspects: **Inefficiency Detection:** The heapify method calls could be optimized by avoiding redundant swaps when subtrees are already valid heaps. **Time Complexity Improvements:** Consider iterative heapify instead of recursive to avoid function call overhead. **Space Complexity Improvements:** Avoid unnecessary temporary arrays during heap construction. **Code Quality:** Use consistent naming (e.g., maxHeapify) and clear method documentation for better readability.

Empirical Results
Performance benchmarks were run on input sizes n = 100, 1,000, 10,000, and 100,000 using the same benchmarking harness used for Min-Heap. Measured execution times were consistent with theoretical predictions: **O(n log n)** scaling for insertions and heapify-based operations. The Max-Heap showed slightly higher constant factors compared to the Min-Heap due to more frequent upward percolations when larger elements were inserted. These results validate the asymptotic complexity and confirm that the Max-Heap is computationally efficient for large-scale data operations.

Comparison with Min-Heap Implementation
When compared to the Min-Heap (student's own algorithm), both data structures exhibit identical asymptotic complexity. However, empirical tests show minor performance variation depending on input characteristics: On random data, execution times are nearly identical. On sorted data, the Min-Heap may perform slightly faster due to fewer percolations when inserting smaller elements. In memory usage, both structures are in-place and differ only by comparison direction. Overall, both

implementations demonstrate O(n log n) operational complexity with strong alignment between theoretical and experimental observations.

Conclusion

The Max-Heap algorithm efficiently supports priority queue operations with logarithmic time complexity and constant auxiliary space. Through both theoretical and empirical validation, the implementation adheres to expected asymptotic bounds and demonstrates practical performance stability. Optimization suggestions include minimizing recursive overhead and improving code readability for maintainability. The comparison with the Min-Heap reveals similar asymptotic efficiency, confirming that both heaps are optimal solutions for ordered data access under different priority directions.