ALGORITHM OVERVIEW

INTRODUCTION

A Max-Heap is a fundamental data structure widely used in algorithms that require efficient retrieval of the maximum element, such as **priority queues** and **heap sort**. It is based on a complete binary tree representation, where each parent node is greater than or equal to its children. This property ensures that the maximum element is always located at the root of the heap.

THEORETICAL BACKGROUND

The Max-Heap is implemented using an array for memory efficiency, where the parent and children indices can be computed arithmetically:

- Parent index: (i - 1) / 2
- Left child: 2 * i + 1
- Right child: 2 * i + 2

Key operations include:

- **Insert** – adds a new element at the end of the heap and “bubbles it up” until the heap property is restored.
- **Extract-Max** – removes the root (maximum element) and “heapifies down” from the new root.
- **Increase-Key** – increases the value of an element and moves it upward if necessary.
- **Build-Heap** – constructs a heap from an unsorted array in linear time using Floyd’s bottom-up method.

Algorithm steps:

1. To **insert**, place the new element at the end and repeatedly swap with its parent until the heap property is satisfied.
2. To **extractMax**, replace the root with the last element, reduce the heap size, and restore the heap property with heapifyDown.
3. To **increaseKey**, update the element’s value and move it upward until the property holds.
4. To **buildHeap**, call heapifyDown for all non-leaf nodes starting from the middle of the array.

Use Cases and Practical Relevance

Max-Heaps are essential in implementing **priority queues**, **job scheduling**, **graph algorithms** (e.g., Dijkstra, Prim’s MST), and **heap sort**.

COMPLEXITY ANALYSIS

TIME COMPLEXITY

1. **Insert**
    a. Worst-case: O(log n) → the new element may bubble up from the bottom to the root.
    b. Best-case: Ω(1) → if the inserted element is smaller than its parent, no swaps are needed.
    c. Average-case: Θ(log n).
2. **Extract-Max**
    a. Worst-case: O(log n) → the new root may trickle down all the way to the last level.
    b. Best-case: Ω(1) → if the heap has only one element, extraction is constant time.
    c. Average-case: Θ(log n).
3. **Increase-Key**
    a. Worst-case: O(log n) → the updated key may bubble up to the root.
    b. Best-case: Ω(1) → if the key does not violate the heap property.
    c. Average-case: Θ(log n).
4. **Build-Heap**
    a. Using Floyd's method, the time complexity is O(n).
    b. Justification: most nodes are near the bottom and require fewer comparisons/swaps.

SPACE COMPLEXITY

- **Heap storage**: O(n) (array-based implementation).
- **Auxiliary space**: O(1) → only a few local variables are used.
- **Tracker overhead**: additional counters (comparisons, swaps, accesses) add negligible O(1) space.

Comparison with Min-Heap

- **Asymptotic complexity** is identical: both structures achieve O(log n) for insert and extract, and O(n) for buildHeap.
- **Practical differences**:
    - Min-Heap implementation uses a more optimized swap and array access accounting, leading to fewer recorded operations.
    - Max-Heap increments the tracker more aggressively, which increases overhead and affects empirical results.

SUMMARY TABLE

Operation	Best Case	Average Case	Worst Case
Insert	$\Omega(1)$	$\Theta(\log n)$	$O(\log n)$
Extract-Max	$\Omega(1)$	$\Theta(\log n)$	$O(\log n)$
Increase-Key	$\Omega(1)$	$\Theta(\log n)$	$O(\log n)$
Build-Heap	$\Omega(n)$	$\Theta(n)$	$O(n)$

The theoretical performance of Max-Heap and Min-Heap is equivalent, but implementation choices can create measurable practical differences.

Recurrence Relation (Worst Case):

Each element may be compared with all previous elements, leading to the following recurrence:

- $T(1) = 0$
- $T(n) = T(n - 1) + (n - 1)$
- $T(n) = 0 + 1 + 2 + \ldots + (n - 1)$
- $T(n) = n(n - 1) / 2$
- $\Rightarrow T(n) = O(n^2)$

CODE REVIEW

INEFFICIENT CODE SECTIONS

· **Redundant Array Access Tracking:**

Similar to the MinHeap, the current MaxHeap increments array access counters multiple times per operation. This inflates memory access metrics and may misrepresent the actual algorithmic performance.

· **Inefficient Error Handling:**

Exceptions for empty heap operations (extractMax, peek) are generic and do not provide informative messages. This reduces debugging clarity and maintainability.

· **No Memory Shrinking Mechanism:**

The heap array grows dynamically when capacity is exceeded but does not shrink when elements are removed. This can cause inefficient memory usage for workloads with fluctuating sizes.

OPTIMIZATION SUGGESTIONS

1. **Improve Error Handling:** Provide detailed exception messages (e.g., "Heap is empty: cannot extractMax()") to improve clarity during debugging and testing.
2. **Optional Metrics Collection:** Use a flag or strategy pattern to enable or disable performance tracking, avoiding unnecessary overhead in production runs.
3. **Memory Management Enhancements:** Introduce array shrinking when the heap size drops below a threshold (e.g., 25% usage). This ensures more efficient memory usage under variable workloads.

CODE QUALITY

Strengths:

· Clear and correct implementation of all standard heap operations (insert, extractMax, peek, heapify).

· Good separation of concerns between algorithm logic and benchmarking/metrics tracking.

· Iterative approach in heapify methods avoids stack overhead from recursion.

Weaknesses:

· Lack of descriptive exception handling.

· Tight coupling between metrics and algorithm code reduces flexibility.

· No dedicated tests for boundary cases (e.g., empty heap, duplicate elements, max-heap with single element).

PROPOSED IMPROVEMENTS FOR TIME AND SPACE COMPLEXITY

Time Complexity:

· The heap operations are already optimal:

- buildHeap: $\Theta(n)$
- insert: $O(\log n)$
- extractMax: $O(\log n)$
- peek: $O(1)$

· No asymptotic improvements are possible, but constant factors can be reduced by decoupling metrics and streamlining error checks.

Space Complexity:

· Current: Θ(n), since the heap is array-based.

· Optimizations: Implementing array shrinking would reduce wasted memory when the heap size decreases significantly.

CONCLUSION

The MaxHeap implementation is correct and efficient for its intended purpose. It achieves the expected time complexities for all operations and is suitable for benchmarking and educational use. The main improvement opportunities lie in **better error handling**, **optional metrics collection**, **memory shrinking**. With these refinements, the MaxHeap can be both more robust and more versatile for practical applications.

EMPIRICAL RESULTS

1. **Performance Plots (Time vs Input Size)**
2. Based on the CLI benchmark (benchmark_all.csv), execution time scales with input size across four dataset types: random, sorted, reverse, and nearly sorted.

Observations (CLI benchmark):

Input Size	Random	Sorted	Reverse	Nearly Sorted
100	0 ms	0 ms	0 ms	0 ms
1,000	1 ms	1 ms	0 ms	0 ms
10,000	2–3 ms	1–2 ms	1–2 ms	1–2 ms
100,000	19 ms	8–9 ms	9 ms	8 ms

- For **small inputs (≤1,000)**, execution completes almost instantly (<1 ms).
- At **100,000 elements**, runtime grows into tens of milliseconds, consistent with expected O(n log n) scaling.
- **Sorted and nearly sorted arrays** are slightly faster, while **random inputs** show the highest time due to increased heapify operations.

Validation of Theoretical Complexity

Theoretical expectations for MaxHeap:

- **BuildHeap:** O(n)

- **Insert/ExtractMax:** O(log n) per operation
- **Overall sort:** O(n log n)

Empirical evidence (n = 100,000, random input):

- Comparisons ≈ 1.75M
- Swaps ≈ 824k
- Array accesses ≈ 6.2M
- Execution time ≈ 19 ms

Since $\log_2$ (100,000) ≈ 17, the expected number of operations is ~1.7M, which aligns closely with measured comparisons and swaps. This validates the theoretical O(n log n) complexity.

Analysis of Constant Factors and Practical Performance

Key insights:

HeapifyDown dominates execution time during extract operations, while HeapifyUp is triggered during insertions.

Input order affects constants, not complexity:

a. Sorted/nearly sorted arrays reduce swaps and comparisons.
b. Random data increases heapify work, leading to higher counts.

Memory usage remains linear (Θ(n)):

c. Peak memory allocations matched input size (up to ~10 MB at n = 100,000).
d. No excessive overhead observed.

Stability across datasets:

e. MaxHeap shows minimal variance across input types compared to quadratic algorithms.

4. Practical Takeaway

- MaxHeap consistently demonstrates O(n log n) scaling with small constant factors.
- Performance is nearly identical across random, sorted, and reverse inputs, showing robustness to input order.
- For large inputs (≥100,000), MaxHeap remains efficient and significantly outperforms quadratic algorithms such as Insertion Sort or Selection Sort.
- Minor optimizations (reducing redundant array access logging, optimizing heapify loops) could further reduce constant factors.

CONCLUSION

Both MinHeap and MaxHeap provide efficient, scalable priority queue operations and sorting foundations. Their empirical results strongly support the theoretical **O(n log n)** complexity. With minor improvements to metric tracking and heapify optimizations, these implementations can serve as both robust educational tools and practical solutions for large datasets.