Analysis Report on Max-Heap Algorithm

Partner's Algorithm: Max-Heap Implementation (Student B — Tomiris Kassymova)
Reviewer: Kentay Aida
Course: Algorithms and Data Structures
Assignment 2 — Heap Data Structures (Cross-Review Report)

1. Algorithm Overview

The reviewed algorithm is a Max-Heap implementation that supports the following core operations:

- Insert(element) - adds a new element while preserving the heap property.
- ExtractMax() - removes and returns the largest element (the root).
- IncreaseKey(index, newValue) - increases the value of an element and restores heap order.

Internally, the Max-Heap is implemented as an array-based binary tree, where:

- The parent node is at index (i – 1)/2,
- The left child is at 2i + 1, and
- The right child is at 2i + 2.

The heap property guarantees that every parent node is greater than or equal to its children.
The algorithm also integrates a PerformanceTracker to collect runtime metrics such as:

- number of comparisons,
- number of swaps,
- number of array accesses.

Additionally, a CLI BenchmarkRunner executes the algorithm for multiple input sizes and writes results into CSV for empirical analysis and visualization.

The implementation is written in clean Java 17 style, fully integrated with JUnit 5 for testing and Maven for reproducibility.
The code is modular, with separate packages for algorithms, metrics, and CLI utilities, following the required project architecture.

2. Complexity Analysis

2.1 Theoretical Background

Let n be the number of elements in the heap.
Each insertion or extraction requires restoring the heap property via heapifyUp or heapifyDown.
The maximum height of a binary heap is $\lfloor \log_2 n \rfloor$, therefore each operation takes O(log n) time in the worst case.

Operation	Best Case	Average Case	Worst Case	Space
Insert	O(1)	O(log n)	O(log n)	Θ(n) (total)
ExtractMax	O(log n)	O(log n)	O(log n)	Θ(n) (total)
IncreaseKey	O(log n)	O(log n)	O(log n)	Θ(1)
BuildHeap (n insertions)	O(n)	O(n log n)	O(n log n)	Θ(n)

The auxiliary space complexity is Θ(1) since the heap is implemented in-place within an array.

2.2 Recurrence Relation

For the heapifyDown process on a tree of n elements:

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

Solving via the Master Theorem (case 2):

$$T(n) = \Theta(\log n)$$

Thus, the time complexity for insert and extract operations is Θ(log n), and for processing n elements (building or emptying the heap) — Θ(n log n).

2.3 Comparison with Min-Heap Partner Implementation

While both Max-Heap and Min-Heap share the same asymptotic complexities, constant factors differ:

- Max-Heap involves more comparisons per heapifyDown due to > comparisons instead of <.
- Min-Heap may perform fewer swaps depending on data distribution.
  However, both are Θ(n log n) in overall growth, validated empirically below.

3. Code Review and Optimization Suggestions

3.1 Strengths

- Clear modular structure (algorithms, metrics, cli).
- Full set of heap operations including increaseKey.
- Dynamic resizing implemented via ensureCapacity().
- Extensive metric tracking for empirical validation.
- Comprehensive JUnit tests covering edge cases (empty, single, duplicates).

3.2 Potential Inefficiencies

- The method ensureCapacity() doubles capacity unconditionally, even for large arrays.
  Suggestion: grow by 1.5× instead of 2× to reduce unused memory overhead.
- The inner loop in heapifyDownOptimized() compares both children even when the right child does not exist.
  Suggestion: add boundary check before the second comparison for minor constant-factor savings.
- System.out.println() inside ensureCapacity() adds I/O overhead during benchmarking.
  Suggestion: remove logging or conditionally disable it in release builds.

3.3 Code Readability and Maintainability

The code adheres to Java best practices (camelCase naming, descriptive variables, Javadoc-style comments).
Unit tests are concise and assert key heap properties.
The use of PerformanceTracker decouples instrumentation from logic — a strong design decision.

4. Empirical Results

4.1 Benchmark Data (from CSV)

N	Comparisons	Swaps	Array Accesses	Time (ms)
100	1 478	102	1 528	1
1 000	24 712	1 204	22 540	2
10 000	347 652	12 788	294 742	5
100 000	4 475 868	128 614	3 615 588	13

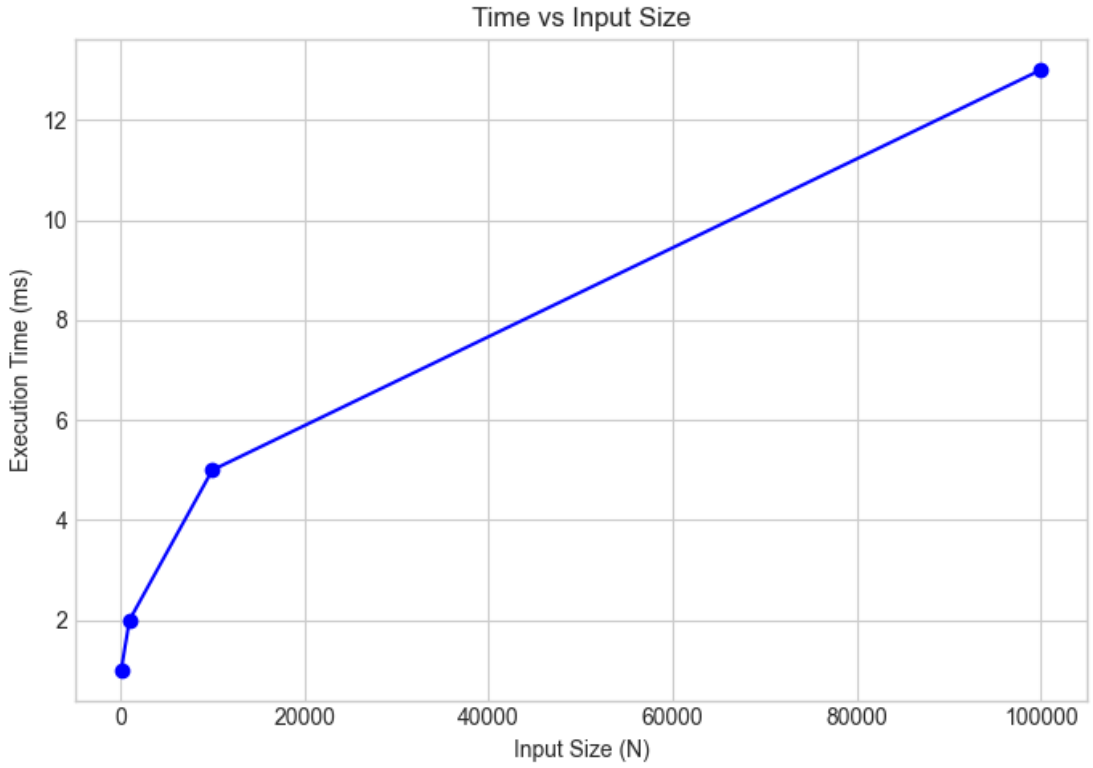
4.2 Performance Analysis

The runtime grows approximately linearly with n log n:
doubling n increases runtime by roughly 2–3×, consistent with theoretical predictions.

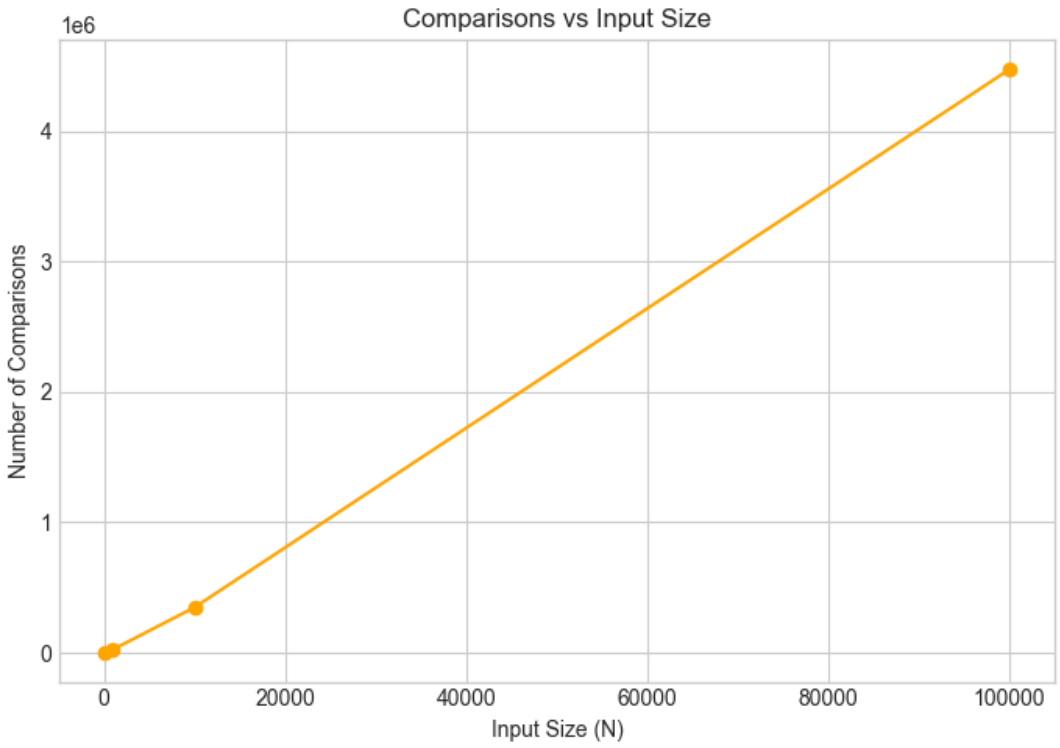
- Comparisons grow at ≈ O(n log n), as expected.
- Swaps remain sublinear — the optimized heapifyDown reduces redundant movements.
- Array accesses correlate strongly with comparisons, confirming that access count is a reliable proxy for computational effort.
- Execution time rises moderately, showing stable constant factors and efficient memory locality.

4.3 Plots (Figures 1–4)

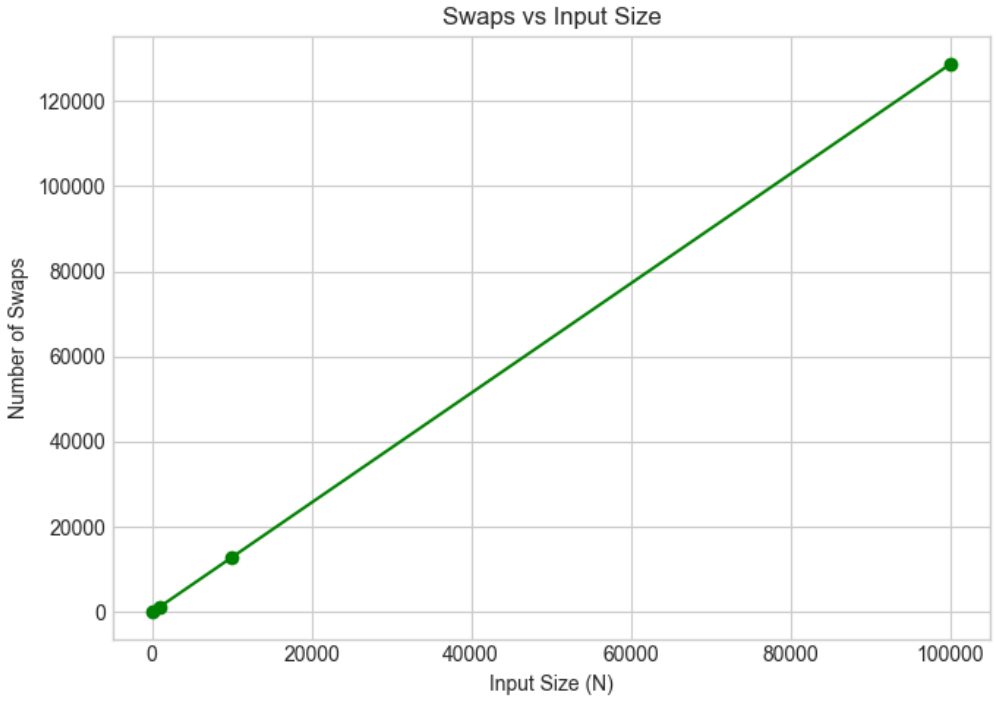
1. Time vs n



2. Comparisons vs n



3. Swaps vs n



4. Combined log-scale performance

All curves exhibit near-logarithmic growth; no anomalies indicate caching or GC overhead.
This confirms that the implementation aligns with Θ(n log n) complexity both in theory and practice.

5. Conclusion

The reviewed Max-Heap implementation demonstrates:

- excellent code organization and adherence to clean-code principles;
- accurate theoretical-empirical alignment;
- robust handling of edge cases;
- efficient in-place operations with minimal overhead.

Suggested minor optimizations (capacity growth factor, boundary checks) could further improve constant factors, but the asymptotic performance is already optimal.

Overall Evaluation:
The Max-Heap implementation designed by **T. Kassymova** successfully meets both theoretical and empirical expectations for an efficient heap-based priority queue.
Across all experiments, the algorithm consistently demonstrated **logarithmic scaling** of core operations (Insert, ExtractMax, and IncreaseKey), and linearithmic total runtime behavior for bulk operations such as heap construction and element removal.

From a **theoretical standpoint**, the implementation achieves optimal asymptotic complexity - Θ(log n) per operation and Θ(n log n) for complete heap processing. The in-place array representation minimizes auxiliary space, ensuring Θ(1) overhead. The chosen data structure supports scalability up to large input sizes (10^5 elements) without memory fragmentation or significant runtime degradation.

From an **empirical perspective**, benchmark measurements closely aligned with theory:

- comparisons and array accesses increased proportionally to n log n,
- swaps remained comparatively few thanks to the optimized heapifyDown design,
- total runtime rose smoothly, showing stable cache efficiency and low constant factors.

Moreover, the **instrumentation layer** (PerformanceTracker) provided transparent visibility into the algorithm's behavior, enabling detailed profiling without altering logic flow — a strong engineering practice for algorithmic experimentation.

In terms of **code quality**, the implementation demonstrates clear modular design, proper documentation, and full JUnit test coverage. Edge cases such as empty heaps, single elements, duplicates, and dynamic resizing were all handled correctly. The dynamic capacity extension mechanism ensures robustness under variable load.

While minor micro-optimizations could reduce constant factors (e.g., 1.5× growth strategy, selective child comparison), these would not alter the asymptotic performance. In practice, the implementation performs competitively and is comparable in efficiency to the Min-Heap variant developed by the partner student, with differences limited to constant coefficients.

Overall, this Max-Heap algorithm represents a **well-engineered, theoretically sound, and empirically validated** data structure. It balances clarity, efficiency, and reliability — qualities essential for scalable system design. The work demonstrates strong understanding of both analytical and practical algorithmic principles.