Max-Heap Algorithm Analysis Report

Student: Orazbek Ulan
Partner's Algorithm: Max-Heap Implementation
Assignment: DAA Assignment 2 - Peer Code Review

1. Algorithm Overview

Max-Heap Data Structure

The Max-Heap is a complete binary tree data structure where each parent node contains a value greater than or equal to its children. This property, known as the max-heap property, ensures that the maximum element is always located at the root (index 0).

The partner's implementation provides a classical array-based Max-Heap with the following key operations:

Core Operations:

- `insert(int value)` - Insert element maintaining heap property
- `extractMax()` - Remove and return maximum element
- `peek()` - Return maximum element without removal
- `buildHeap(int[] arr)` - Build heap from array using Floyd's algorithm
- `increaseKey(int index, int newValue)` - Increase value at given index
- `heapSort(int[] arr)` - Sort array using heap sort algorithm

Supporting Operations:

- `heapifyUp(int index)` - Restore heap property upward
- `heapifyDown(int index)` - Restore heap property downward

Theoretical Foundation

Max-Heaps are fundamental in computer science with applications in:

- Priority queues (scheduling systems)
- Heap sort algorithm
- Graph algorithms (Dijkstra's, Prim's)
- Memory management systems

2. Complexity Analysis

Time Complexity Analysis

Insert Operation

Mathematical Derivation:

The `insert()` operation adds element at the end and calls `heapifyUp()`:

```
T_insert(n) = O(1) + T_heapifyUp(n)
```

`heapifyUp()` traverses from leaf to root in worst case:

- Tree height in complete binary tree: $h = \lfloor \log_2(n) \rfloor$
- Maximum comparisons: $h = \lfloor \log_2(n) \rfloor$

Result:

- Best Case: $\Omega(1)$ - element already in correct position
- Average Case: $\Theta(\log n)$ - element bubbles up partially
- Worst Case: $O(\log n)$ - element bubbles to root

ExtractMax Operation

Mathematical Derivation:

The `extractMax()` moves last element to root and calls `heapifyDown()`:

```
T_extractMax(n) = O(1) + T_heapifyDown(n)
```

`heapifyDown()` traverses root to leaf:

- At each level, 2 comparisons (left child, right child)
- Maximum levels: $\lfloor \log_2(n) \rfloor$
- Total comparisons: $2 \times \lfloor \log_2(n) \rfloor$

Result:

- Best Case: $\Omega(1)$ - element already correct
- Average Case: $\Theta(\log n)$ - element sinks partially
- Worst Case: $O(\log n)$ - element sinks to leaf

IncreaseKey Operation

```java
public void increaseKey(int index, int newValue) {
    heap[index] = newValue;      // O(1)
    heapifyUp(index);           // O(log n)
}
```

Result: Θ(log n) in all cases

BuildHeap Operation (Floyd's Algorithm)

Mathematical Proof:

Using bottom-up heapify on ⌊n/2⌋ internal nodes:

```
T_buildHeap(n) = Σ(i=0 to ⌊log₂(n)⌋) nodes_at_level_i ×
work_per_node_i
```

Where:

- Nodes at level i from bottom: ≤ ⌈n/2^(i+1)⌉
- Work per node at level i: O(i)

```
T_buildHeap(n) ≤ Σ(i=0 to ⌊log₂(n)⌋) ⌈n/2^(i+1)⌉ × i
                = n × Σ(i=1 to ∞) i/2^i
                = n × 2 = O(n)
```

Result: Θ(n) - Linear time complexity

HeapSort Operation

```
T_heapSort(n) = T_buildHeap(n) + n × T_extractMax(n)
               = O(n) + n × O(log n)
               = O(n log n)
```

Result: Θ(n log n) in all cases

Space Complexity Analysis

Primary Storage:

- Heap array: `int[] heap` - O(n) space
- No auxiliary data structures for position tracking

Additional Variables:

- `size`, `capacity`, `tracker` - O(1) space

Total Space Complexity: Θ(n)

Comparison with Min-Heap Partner Algorithm

Operation	Max-Heap Time	Min-Heap Time	Max-Heap Space	Min-Heap Space
Insert	Θ(log n)	Θ(log n)	Θ(1)	Θ(1)
Extract	Θ(log n)	Θ(log n)	Θ(1)	Θ(1)
Increase/Decrease-Key	Θ(log n)	Θ(log n)	Θ(1)	Θ(1)
Build-Heap	Θ(n)	Θ(n)	Θ(n)	Θ(n)
Position Tracking	❌ None	✅ HashMap O(n)	Better	Uses more memory

Key Difference: Min-Heap implementation uses HashMap for O(1) key lookup, while Max-Heap uses index-based access.

3. Code Review & Optimization

Critical Issues Identified

Issue 1: Index-Based IncreaseKey (High Priority)

Problem:

```
public void increaseKey(int index, int newValue) {
    // User must know internal heap index!
    heap[index] = newValue;
}
```

Issues:

- Violates encapsulation - exposes internal structure
- User cannot identify element position without additional tracking
- Not user-friendly for practical applications

Solution:

```java
// Add position tracking
private Map<Integer, Integer> positionMap;

public void increaseKey(int oldValue, int newValue) {
    Integer index = positionMap.get(oldValue);
    if (index == null) {
        throw new IllegalArgumentException("Value not found: " +
oldValue);
    }
    heap[index] = newValue;
    positionMap.remove(oldValue);
    positionMap.put(newValue, index);
    heapifyUp(index);
}
```

Issue 2: Fixed Capacity Limitation

Problem:

```java
if (size == capacity) {
    throw new IllegalStateException("heap is full");
}
```

Impact: Limits scalability, requires pre-allocation estimation

Solution:

```java
private void resize() {
    capacity *= 2;
    heap = Arrays.copyOf(heap, capacity);
    tracker.incrementArrayAccesses(size);
}
```

Issue 3: HeapSort Modifies Original Array

Problem:

```java
public int[] heapSort(int[] arr) {
    buildHeap(arr);  // Modifies original array!
    // ... sorting modifies arr further
    return arr;      // Returns same reference
```

```
}
```

Issues:

- Side effects - original array is destroyed
- Violates functional programming principles
- Unexpected behavior for users

Solution:

```
public int[] heapSort(int[] arr) {
    int[] copy = Arrays.copyOf(arr, arr.length);
    buildHeap(copy);
    // ... sort copy
    return copy;
}
```

Issue 4: Missing Duplicate Detection

Current Code:

```
public void insert(int value) {
    heap[size] = value;  // Allows duplicates
}
```

Recommendation:
Add duplicate checking for data integrity:

```
if (contains(value)) {
    throw new IllegalArgumentException("Duplicate value: " + value);
}
```

Issue 5: Inefficient Array Access Tracking

Problem:

```
private void swap(int i, int j) {
    // ... swap logic
    tracker.incrementArrayAccesses(); // Only 1 increment for 4
accesses
```

```
}
```

Current: Under-counts array accesses
Solution: `tracker.incrementArrayAccesses(4);`

Performance Bottlenecks

1. No Position Map: Finding elements requires O(n) linear search
2. Fixed Capacity: Frequent capacity exceptions in dynamic scenarios
3. Redundant Operations: Some operations lack early termination checks

Code Quality Assessment

Strengths:

- ✅ Clean, readable structure
- ✅ Proper encapsulation of helper methods
- ✅ Good use of performance tracking
- ✅ Comprehensive error handling

Weaknesses:

- ❌ Index-based API design
- ❌ Limited scalability
- ❌ Incomplete metric tracking
- ❌ Side effects in heapSort

4. Empirical Results

Benchmark Data Analysis

Test Configuration:

- Input sizes: 100, 1,000, 10,000, 100,000 elements
- Algorithm tested: HeapSort (buildHeap + repeated extractMax)
- Metrics: Execution time, comparisons, swaps, array accesses
- Runs per size: 5 (averaged)

Results Table:

Size (n)	Time (ms)	Comparisons	Swaps	Array Accesses	Comp /n	Time/n log n

100	0.127	1,038	584	1,755	10.38	0.0191
1,000	0.248	16,842	9,065	27,197	16.84	0.0251
10,000	1.366	235,387	124,201	372,604	23.54	0.0104
100,000	9.975	3,019,900	1,575,038	4,725,116	30.20	0.0060

Complexity Verification

Time Complexity Validation

Expected: O(n log n)

Theoretical vs Measured:

- n=100 to n=1,000: 10× size → 1.95× time (expected: ~3.32×)
- n=1,000 to n=10,000: 10× size → 5.51× time (expected: ~3.32×)
- n=10,000 to n=100,000: 10× size → 7.30× time (expected: ~3.32×)

Analysis: Time scaling shows O(n log n) trend but with high constant factors due to:

1. JVM warm-up effects on smaller inputs
2. Cache performance variations
3. System overhead in nanosecond measurements

Operations Complexity Validation

Comparisons per element:

- Theoretical: ~2 × $\log_2(n)$ for heapSort
- Measured: 10.38 (n=100) → 30.20 (n=100,000)
- Trend: Logarithmic increase ✅ confirms O(n log n)

Swaps Analysis:

- Swap/Comparison ratio: ~0.55-0.56 (consistent)
- Interpretation: About half of comparisons lead to swaps, indicating good heap property maintenance

Performance Characteristics

Strengths Observed:

1. Consistent O(n log n) behavior - no worst-case degradation
2. Predictable memory usage - in-place sorting
3. Good constant factors for large inputs ($n \geq 10{,}000$)

Limitations Observed:

1. High overhead on small inputs - JVM/measurement artifacts
2. Only heapSort tested - individual operations not benchmarked
3. Missing increase-key performance data

Comparison with Theoretical Predictions

Metric	Theoretical	Empirical	Match
Time Growth Rate	O(n log n)	~O(n log n)	✅ Good
Space Usage	O(1) auxiliary	O(1) measured	✅ Perfect
Comparisons	~2n log n	~20-30n	⚠️ Higher constants

Conclusion: Implementation correctly follows theoretical complexity with some higher constant factors due to implementation details and measurement overhead.

5. Conclusion

Summary of Findings

The partner's Max-Heap implementation demonstrates solid algorithmic foundations with correct time complexities and efficient space usage. The core heap operations are properly implemented using standard techniques (Floyd's buildHeap, bottom-up/top-down heapify).

Critical Recommendations

High Priority Fixes:

1. Implement value-based increaseKey() with position tracking (HashMap)
2. Add dynamic capacity expansion to remove fixed-size limitation
3. Fix heapSort side effects by working on array copies
4. Add comprehensive benchmarking for individual operations

Medium Priority Improvements:

1. Enhanced error handling with more descriptive exceptions
2. Duplicate detection for data integrity
3. Metric tracking accuracy improvements
4. API documentation with complexity guarantees

Optimization Impact Assessment

Estimated Performance Improvements:

Optimization	Time Impact	Space Impact	Usability Impact
Position Map	+O(1) lookup	+O(n) space	+++High
Dynamic Resize	Negligible	Optimized	+++High
Copy in heapSort	+O(n) once	+O(n) temp	++Medium
Better Metrics	None	None	+Low

Final Assessment

Strengths:

- ✅ Correct core algorithm implementation
- ✅ Proper complexity characteristics
- ✅ Clean, maintainable code structure
- ✅ Good performance tracking integration

Critical Gaps:

- ❌ Index-based API limits practical usability
- ❌ Fixed capacity constrains scalability
- ❌ Incomplete benchmarking of all operations
- ❌ Side effects in sorting method

The implementation successfully demonstrates understanding of heap data structures and achieves the required algorithmic complexity. With the recommended optimizations, this would become a production-ready, user-friendly Max-Heap implementation suitable for real-world applications.