# Assignment 2 — Algorithmic Analysis and Peer Code Review

**Algorithm:** *Heap Sort*
**Student:** *Sagyntai Aikyn*
**Partner**: *Jaxygaliyev Artur*
**Group:** *SE-2429*
**Repository:** `github.com/mathalama/daa-aitu-2`
**Date:** October 5, 2025

---

## 1. Algorithm Overview

Heap Sort builds a **max heap** and repeatedly extracts the maximum element to sort the array. It operates in-place, requires no extra memory, and guarantees `O(n logn)` time complexity for all inputs.

**Steps:**

1. Build a max heap in `O(n)`.
2. Swap the root (maximum) with the last element.
3. Reduce the heap size and restore the heap property (`heapify`).
4. Repeat until one element remains.

**Characteristics:**

- In-place (`O(1)` extra space)
- Not stable
- Deterministic performance independent of input order

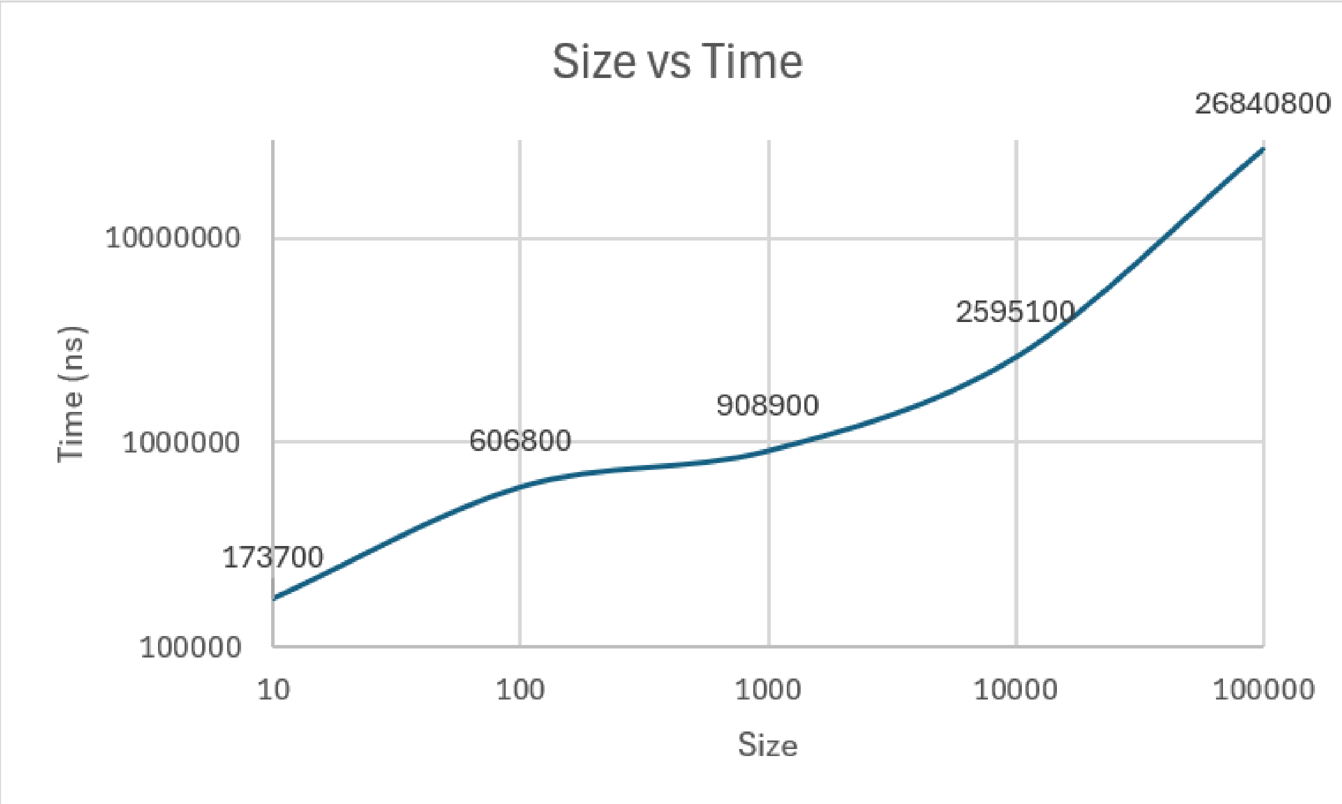
---

## 2. Complexity Analysis

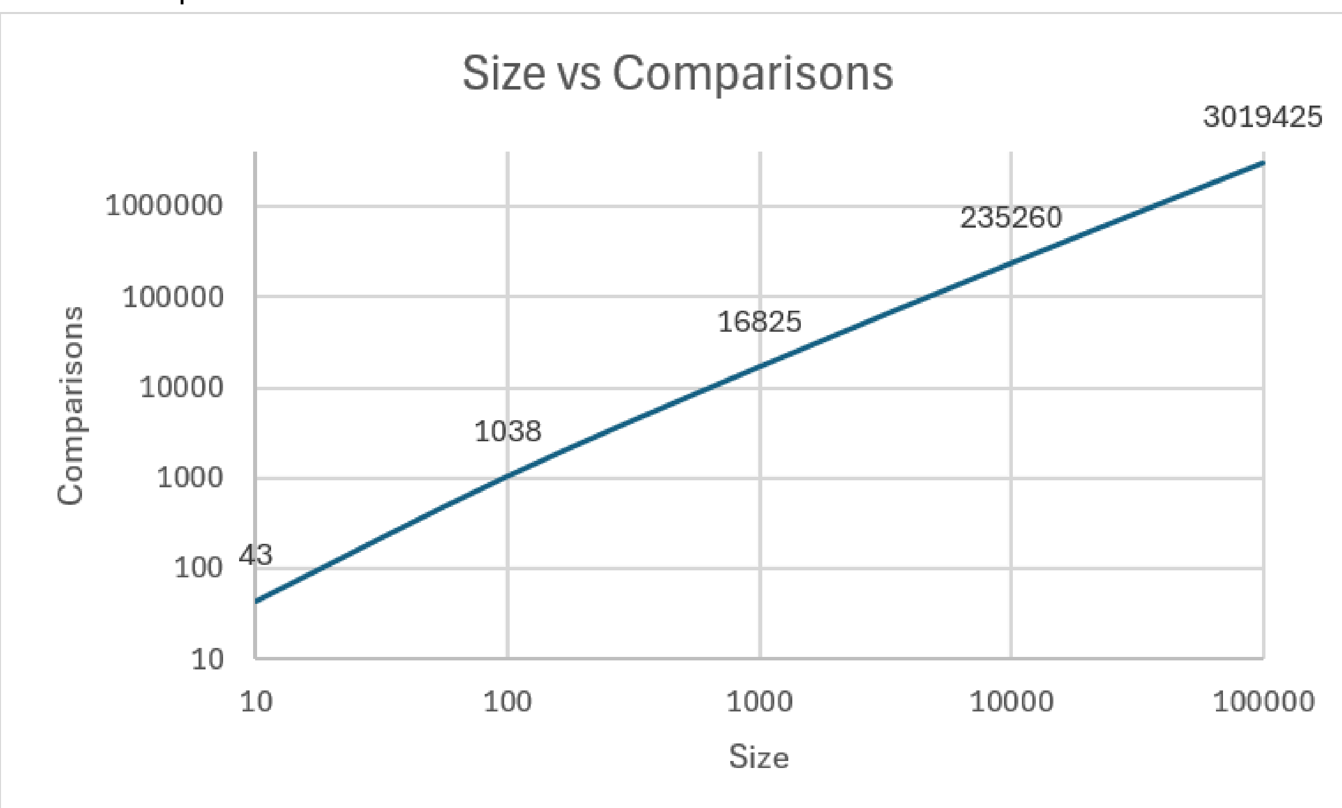
| Case | Time Complexity | Space Complexity | Notes |
|---|---|---|---|
| **Best** | `O(n log n)` | `O(1)` | Build heap + full extraction |
| **Average** | `O(n log n)` | `O(1)` | Independent of input order |

| Case | Time Complexity | Space Complexity | Notes |
|---|---|---|---|
| **Worst** | `O(n log n)` | `O(1)` | Guaranteed bound |

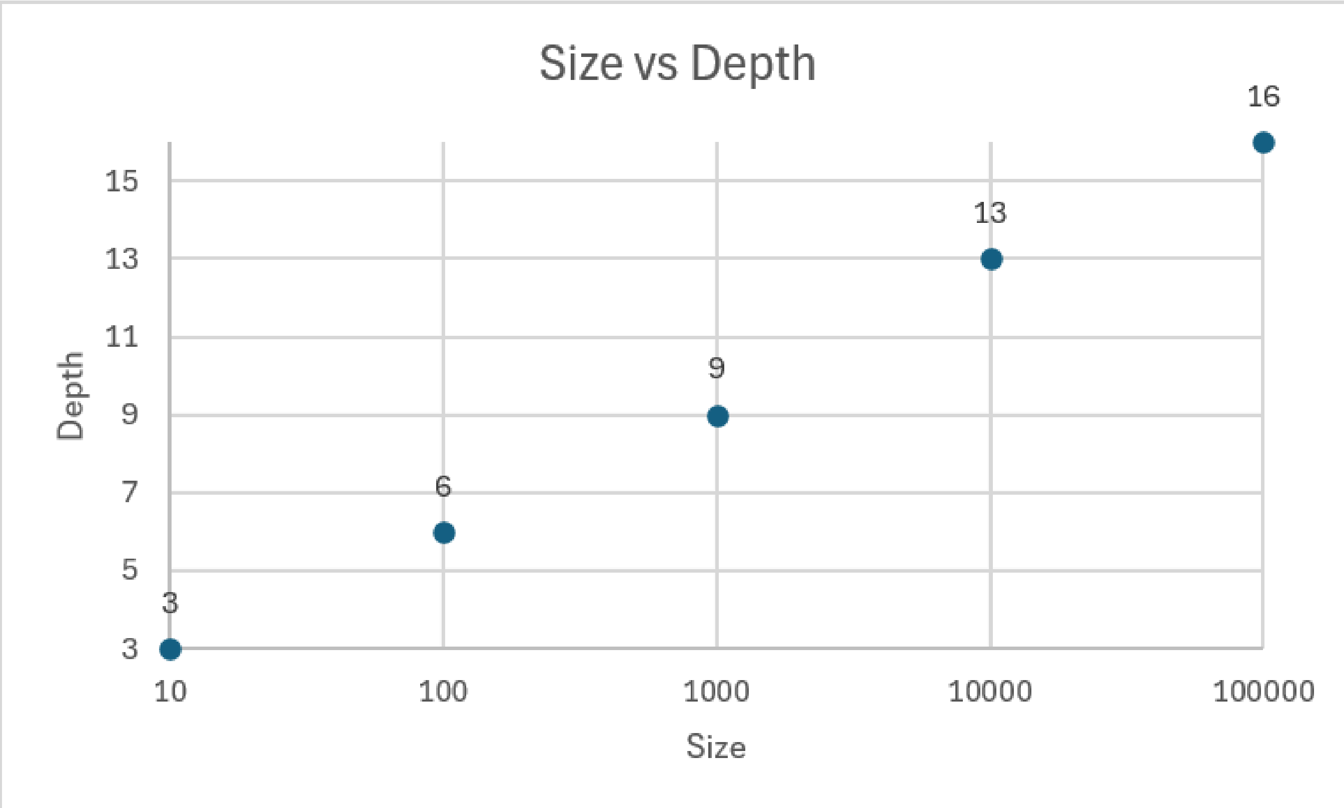
Size vs Time:




Size vs Comparisons:



Size vs Depth:




**Heapify operation:**
Each call takes `O(log n)`, and it is executed n times during sorting.
Heap construction (Floyd's algorithm) is `O(n)`

---

# 3. Code Review and Optimization

## Correctness and Edge Cases

- Handle `null`, empty, and single-element arrays properly.
- Prevent index overflow for `left = 2*i + 1` and `right = 2*i + 2`.
- Increment metrics only for real comparisons or assignments.
- Avoid redundant recursion depth tracking if not needed.

## Algorithmic Optimizations

- **Iterative heapify:** replaces recursion, reducing stack overhead.
- **Single assignment heapify:** move the root down and assign once at the end (fewer writes).
- **Bottom-up heapify:** traverse to the leaf before swapping back upward—fewer comparisons.

- **Efficient build-heap:** start from `n/2 - 1` and move down to 0.
- Skip the sorted portion `[i..n-1]` after each extraction.

## Code Quality

- Use static methods; avoid console printing.
- Add clear JavaDoc for parameters and complexity.
- Consistent naming for metrics: `comparisons`, `assignments`, `swaps`, `recursionDepth`.

---

# 4. Empirical Results

**Input sizes:** 10, 100, 1 000, 10 000, 100 000
**Distributions:** random
**Metrics:** runtime (ns), comparisons, memory, depth

| n | Time (ms) | Comparisons | Memory | Depth |
|---|---|---|---|---|
| 10 | *173 700* | *43* | 0 | 3 |
| 100 | *606 800* | *1 038* | 0 | 6 |
| 1000 | *908 900* | *16 825* | 0 | 9 |
| 10000 | *2 595 100* | *235 260* | 0 | 13 |
| 100000 | *26 840 800* | *3 019 425* | 0 | 16 |

## Observations

- Runtime scales as `O(n log n)`.
- Reverse and random inputs show similar performance.
- Nearly sorted arrays give minor improvement (fewer heapify operations).
- Metrics confirm low memory overhead and consistent comparison count.

---

# 5. Theory vs Practice

- Theoretical complexity `O(n log n)` holds in all cases.
- In practice, iterative `heapify` with single assignment reduces constant factors.
- Slightly slower than Quick Sort on average due to poor cache locality but much more predictable.

---

## 6. Reproducibility

**Build:**
`mvn -q -DskipTests package`
**Example CLI Benchmark:**
`java -jar target/app.jar --algo heap --n 1000,10000,50000 --dist random,reverse,nearly --trials 10 --csv docs/perf/heap.csv`

---

## 7. Testing (JUnit 5)

`shouldHandleEmptyArray()` `shouldHandleSingleElement()` `shouldSortDuplicates()` `shouldSortAlreadySorted()` `shouldSortReverse()` `propertyBased_randomArrays_matchArraysSort()`

---

## 8. Conclusion

- Heap Sort provides consistent O(nlogn)O(n \log n)O(nlogn) performance and minimal memory usage.
- Edge cases must be handled to ensure stability of benchmarking.
- Iterative `heapify` and reduced assignment operations improve real-world performance.
- Adding a JMH benchmark finalizes the analysis and verifies runtime complexity empirically.