Algorithms Assignment — Heap Sort

Group 4: Zarina & Nursultan
Group: SE-2429

All screenshots are in the folder named "results" (diagrams and total result)

Abstract

This report examines the correctness, complexity, and empirical performance of in-place Heap Sort in Java. We validate the implementation with unit tests and evaluate performance across four input distributions. Results confirm Θ(n log n) scaling with O(1) extra space.

1. Overview

Heap Sort is a comparison-based sorting algorithm that builds a binary max-heap and then repeatedly extracts the maximum element to produce a sorted array. Unlike Quick Sort, Heap Sort guarantees O(n log n) time in the worst case and uses O(1) auxiliary memory. Typical use cases include scenarios where worst-case guarantees and constant extra space are important.

2. Algorithm & Implementation

We implement bottom- up heap construction by calling sift- down from $\lfloor n/2 \rfloor - 1$ down to 0. During the sorting phase, we repeatedly swap the root with the last element of the heap and sift the new root down until the heap property is restored. The implementation is in Java 17 with JUnit 5 tests.

Instrumentation: A PerformanceTracker counts comparisons, swaps, and array accesses. This enables both time-based and operation-based evaluation in the empirical section.

3. Complexity Analysis

• Time (Best/Average/Worst): Θ(n log n). Heapify runs in O(n) and the n extract- max operations cost O(n log n) overall.
• Space: O(1) auxiliary; sorting is performed in place.
• Stability: Not stable.
• Notes: Constants depend on branch behavior, cache locality, and implementation details.

4. Empirical Results

Setup. We benchmark for n ∈{10^2, 10^3, 10^4, 10^5} across four input distributions: random, sorted, reverse, and nearly- sorted, averaging multiple trials. We report both wall- clock time and average operation counts.

How to reproduce:

- Run unit tests: mvn clean test
- Generate CSV: mvn exec:java "-Dexec.mainClass=cli.BenchmarkRunner" "-Dexec.args=--algo heap --n 100,1000,10000,100000 --trials 5 --dist random,sorted,reverse,nearly-sorted"
- Build charts from docs/performance-plots/heap_bench.csv

Insert Figures

Observation. The time curve follows Θ(n log n), consistent with theory. Differences among input distributions are modest because per- extraction work depends primarily on heap height (≈ log n). Operation counts also scale ~ n log n.

5. Peer Review

If the assignment requires peer analysis, review a partner's algorithm (e.g., Shell Sort). Discuss gap sequences, expected complexity, and practical behavior. Identify bottlenecks and suggest improvements. Include brief code snippets or pseudocode to justify claims.

6. Conclusion

Heap Sort offers predictable performance (Θ(n log n)) and constant extra space, making it a strong choice when worst- case guarantees matter or memory is constrained. In practice, Quick Sort may be faster on random data due to better cache behavior, but Heap Sort remains robust across inputs.

Appendix: Project & Reproducibility

- Java 17, Maven 3.9+
- Source: src/main/java/algorithms/HeapSort.java, metrics/PerformanceTracker.java, cli/BenchmarkRunner.java
- Tests: src/test/java/algorithms/HeapSortTest.java
- Data & Plots: docs/performance-plots/heap_bench.csv, time_vs_n.png, array_accesses_vs_n.png
- Run: mvn clean test; then mvn exec:java with the arguments given above.