Individual Analysis Report: HeapSort Implementation

Course: Algorithmic Analysis
Author: Akhmet Alisher SE-2425
Date: 05.10.2025

---

1. Algorithm Overview

HeapSort is a **comparison-based sorting algorithm** that utilizes a **binary heap data structure** to organize elements efficiently. It follows the "heap property," ensuring that the parent node is always greater (or smaller) than its children, depending on whether a max-heap or min-heap is used.

The algorithm operates in two main phases:

1. **Heap Construction:**
   Convert the input array into a max-heap, ensuring the largest element is at the root.
2. **Extraction Phase:**
   Repeatedly swap the root element (maximum) with the last element in the heap and reduce the heap size by one, restoring the heap property each time.

Key Characteristics:

- Deterministic and in-place (requires only O(1) extra memory).
- Not stable (equal elements may change relative order).
- Performs consistently well regardless of input distribution.

---

2. Complexity Analysis

2.1 Time Complexity

- **Best Case:** Θ(n log n)
  HeapSort maintains the same time complexity across all cases since every element

is involved in logarithmic reheapification steps.

- **Average Case:** Θ(n log n)
  Regardless of input distribution, HeapSort repeatedly maintains the heap structure, resulting in predictable and consistent performance.

- **Worst Case:** Θ(n log n)
  Even in adversarial or reverse-sorted input, the heap property ensures performance stability — no degradation to quadratic behavior.

2.2 Space Complexity

- **In-place algorithm:** O(1) auxiliary space.
  The heap structure is implemented directly in the input array, requiring no additional memory allocations.

2.3 Comparison with ShellSort

- HeapSort provides guaranteed O(n log n) performance in all scenarios.

- ShellSort may vary between $O(n \log^2 n)$ and $O(n^2)$ depending on the gap sequence.

- For large datasets, HeapSort offers superior predictability and reliability.

---

3. Code Review & Optimization

Observations

- The provided repository implements HeapSort in a modular and well-structured manner.

- The algorithm is located under `algorithms/HeapSort.java`, and is fully instrumented using a **PerformanceTracker** class to collect detailed runtime metrics.

- A dedicated CLI utility (`BenchmarkRunner`) runs experiments and logs performance data to CSV files.

- JUnit tests (`HeapSortTest`) validate correctness across various input scenarios.

Identified Strengths

- **Modularity:** Clear separation of algorithm, metrics, and benchmarking code.
- **Instrumentation:** Tracks comparisons, swaps, and array accesses.
- **Testing Coverage:** Includes edge cases — empty array, single element, duplicates, and sorted/reverse inputs.
- **Reproducibility:** BenchmarkRunner automates data generation and CSV output.

Potential Improvements

- Include **visualization scripts** (e.g., Python or JavaFX) to plot CSV results.
- Enhance comments within the codebase for educational clarity.
- Implement **heapify optimizations**, such as bottom-up heap construction.
- Extend benchmarking to compare multiple algorithms (e.g., MergeSort, QuickSort).

---

4. Empirical Results

The current implementation generates CSV logs for array sizes `n = 100, 1,000, 10,000, 100,000`.
Each benchmark records:

- Execution time (nanoseconds)
- Number of comparisons
- Number of swaps
- Array accesses

Expected Performance Trends:

- Execution time grows near-linearly with n·log(n).
- Comparisons and swaps scale predictably due to the heap's logarithmic depth.
- For smaller arrays, HeapSort may appear slower than InsertionSort or ShellSort due to heap overhead.

- For large-scale datasets, HeapSort exhibits stable and efficient performance across all input types (random, sorted, reverse).

---

5. Conclusion

HeapSort is a **robust and theoretically optimal** sorting algorithm suitable for consistent large-scale performance.
The current Java implementation demonstrates solid modularity, correctness, and empirical tracking features.

However, further enhancements can elevate its analytical and educational value:

- Integrate **graphical performance visualization**.
- Compare results with other algorithms (e.g., QuickSort, ShellSort).
- Add documentation explaining the **heap property** and **sift-down** process for beginners.

Overall, this version of HeapSort successfully combines theoretical reliability with practical empirical analysis, making it an excellent reference implementation for algorithm performance studies.