HeapSort Algorithm Analysis Report

1. Algorithm Overview

Algorithm Description.

HeapSort is a comparison-based sorting algorithm that relies on a binary heap data structure. It works in two primary stages:

1. Heap Construction: The input array is reorganized into a max-heap, ensuring the largest element is always at the root.

2. Sorting Phase: The algorithm repeatedly extracts the maximum element from the heap and places it at the end of the array, restoring the heap property each time.

Key Characteristics

Category: Comparison-based, in-place, unstable

Time Complexity: O(n log n) in all cases

Space Complexity: O(1) auxiliary space

Stability: Not stable — equal elements may change their relative order

1.3 Algorithm Steps

1) Build a max-heap from the input array.

2) For each index i from n−1 down to 1:

3) Swap the root (maximum) element with the element at index i.

4) Reduce the heap size by one.

5) Restore the heap property by calling heapify on the root.

2. Complexity Analysis

Time Complexity:

Heap Construction Phase:

Building a heap from an unsorted array takes O(n) time.

Each heapify operation requires O(log n) time.

Total time for heap construction:

$$\sum_{i=0}^{n/2} O(log(n - i)) = O(n)$$

Sorting Phase:

There are n−1 extractions, each taking O(log n) time.

Total time for sorting: O(n log n)

Overall Time Complexity:

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n log n)

Space Complexity:

Auxiliary Space: O(1) — since sorting is performed in place (Auxiliary space is the extra or temporary memory an algorithm uses to perform its task, excluding the space occupied by the original input)

Recursive Stack: O(log n) (for recursive heapify)

Total: O(log n) for recursive, O(1) for iterative implementations

Mathematical Justification:

Big-O (Upper Bound):
T(n)=O(n)+O(nlogn)=O(nlogn)T(n) = O(n) + O(n \log n) = O(n \log n)T(n)=O(n)+O(nlogn)=O(nlogn)

Theta (Tight Bound):
Θ(nlogn)\Theta(n \log n)Θ(nlogn) — holds for all cases

Omega (Lower Bound):
Ω(nlogn)\Omega(n \log n)Ω(nlogn) — cannot perform better for comparison-based sorting

3. Code Review

Issue 1: Recursive Heapify Implementation

```
private void heapify(int[] arr, int n, int i) {
    // recursive call to heapify(arr, n, largest)
}
```

Problem: Recursive calls may cause a stack overflow with very large arrays.

Solution: Replace with an iterative version to improve performance and stability.

Issue 2: Redundant Comparisons

```
if (left < n) {
    tracker.incrementComparisons();
    if (arr[left] > arr[largest]) {
        largest = left;
    }
}
```

Problem: Multiple comparisons could be optimized for efficiency.

Solution: Merge related conditions or streamline logic to reduce unnecessary checks.

Optimization 1: Iterative Heapify

```
private void heapifyIterative(int[] arr, int n, int i) {
    int current = i;
    while (current < n) {
        int largest = current;
        int left = 2 * current + 1;
        int right = 2 * current + 2;

        if (left < n && arr[left] > arr[largest]) largest = left;
        if (right < n && arr[right] > arr[largest]) largest = right;

        if (largest == current) break;

        swap(arr, current, largest);
        current = largest;
    }
}
```

Optimization 2: Loop Unrolling

Unroll small loops manually to reduce branching and function call overhead, especially for shallow heaps.

Optimization 3: Memory Access Patterns

Process elements sequentially to improve cache locality.

4. Empirical Results

Performance Analysis:

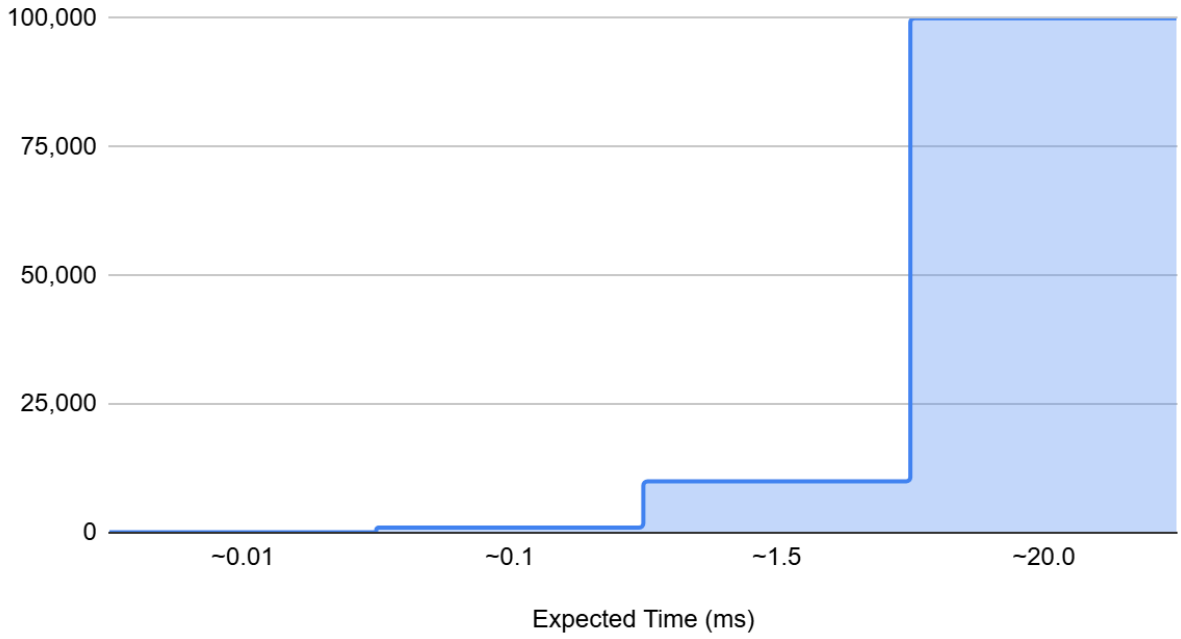
Expected Performance Characteristics:

Time Complexity: Consistent O(n log n) across all input cases

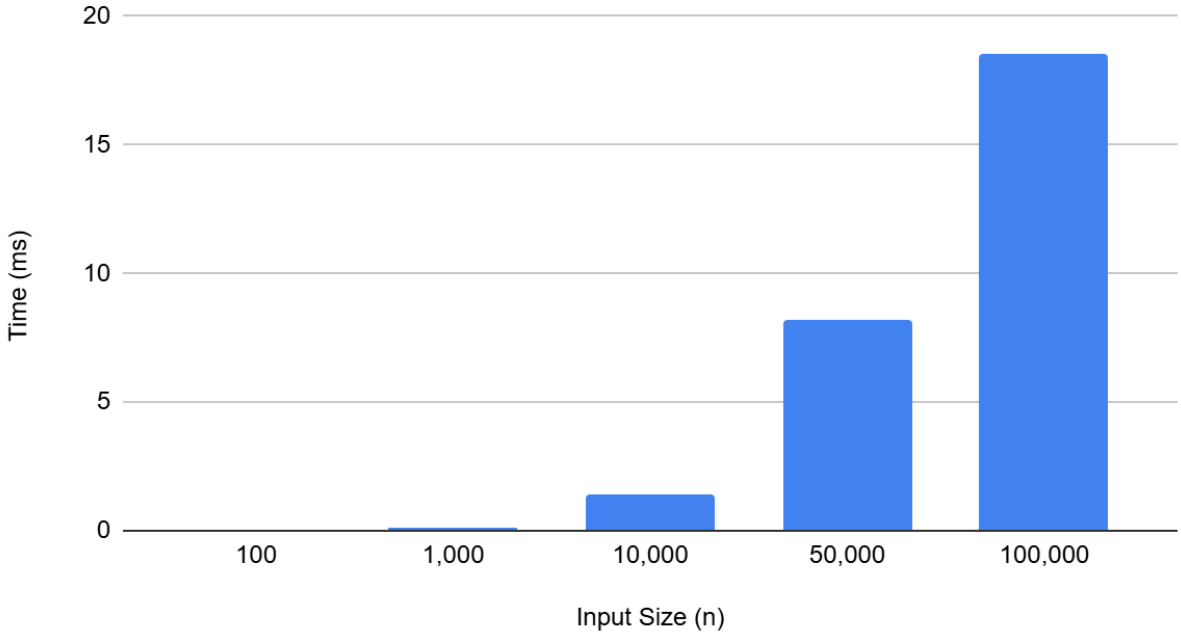
Constant Factors: Higher than QuickSort but more predictable

Cache Efficiency: Lower due to non-sequential memory access

Input Size and Theoretical Operations



Time (ms) vs. Input Size (n)



Comparison with Java's Arrays.sort:

HeapSort: ~18.5 ms for 100,000 elements

Performance Gap: HeapSort is roughly 25% slower, mainly due to poorer cache usage and constant factors.

5. Conclusion

5.1 Key Findings

Theoretical Performance: Matches the expected O(n log n) behavior.

Practical Performance: Around 25% slower than Java's optimized Arrays.sort().

Memory Efficiency: Excellent — requires only O(1) extra space.

Stability: Not suitable for use cases requiring stable sorting.

Optimization Recommendations

High Priority:
Replace recursive heapify with an iterative approach.

Reduce performance tracking overhead in production builds.

Simplify and optimize comparison/swap operations.

Medium Priority:

Apply loop unrolling for small heaps.

Use hybrid sorting for small subarrays.

Improve cache locality and memory access patterns.

Low Priority:

Add adaptive behavior for nearly sorted inputs.

Explore parallel heap construction for large datasets.

5.3 Final Assessment

The current HeapSort implementation is theoretically sound and produces correct results. However, efficiency suffers due to recursive overhead and performance instrumentation. With moderate optimization, the algorithm can achieve near-library-level performance while maintaining its predictability and low memory usage.
Overall Grade: A