Algorithm Overview

HeapSort and Max-Heap Implementation

HeapSort is a comparison-based sorting algorithm that leverages the heap data structure, specifically a Max-Heap, to efficiently sort an array. A Max-Heap is a complete binary tree in which each parent node is greater than or equal to its child nodes. This property ensures that the maximum element is always at the root of the heap.

The algorithm consists of two main phases:

Heap Construction (Build-Heap)
The input array is reorganized to satisfy the Max-Heap property. This is typically done in a bottom-up manner, starting from the last non-leaf node and calling heapify for each node. Heapify ensures that each subtree rooted at a node maintains the Max-Heap property. This phase has a linear time complexity of $\Theta(n)$.

Sorting Phase (Extract-Max)
The maximum element (root) is repeatedly swapped with the last element of the heap and removed from the heap. After each extraction, heapify is called on the new root to restore the heap property. Since heapify takes $O(\log n)$ time and there are n elements, this phase contributes $O(n \log n)$ to the overall time complexity.

HeapSort is an in-place algorithm, requiring no additional memory for storing elements, except for minor auxiliary space for recursive stack calls if a recursive heapify is used. Its time complexity is $\Theta(n \log n)$ for best, worst, and average cases, making it highly predictable regardless of input ordering.

Theoretical Background

HeapSort is grounded in binary heap theory, where a complete binary tree is stored as an array. For a node at index i:

Left child index = 2*i + 1

Right child index = 2*i + 2

Parent index = (i-1)/2

The heap operations — insertion, deletion, and heapify — exploit this index calculation to achieve efficient logarithmic time complexity per operation. HeapSort’s efficiency arises from using the Max-Heap to repeatedly access the largest unsorted element and placing it in its final sorted position.

The algorithm is stable in performance, deterministic, and performs consistently across different types of input data, unlike other quadratic algorithms such as Insertion or Selection Sort. Its space efficiency and predictable runtime make it suitable for large datasets in both academic and practical applications.

Complexity Analysis

Time Complexity

HeapSort consists of two main phases: **Heap Construction** and **Sorting (Extract-Max)**.

1. Heap Construction (Build-Max-Heap):

We start from the last non-leaf node and call heapify bottom-up.

For a node at height h, heapify takes O(h) time.

The total time for all nodes is:

$$T(n) = \sum_{h=0}^{\log n} (\text{number of nodes at height h}) \cdot O(h)$$

Since the number of nodes at height h is at most n /2^(h+1) , we have:

$$T(n) = \sum_{h=0}^{\log n} \frac{n}{2^{h+1}} \cdot O(h) = O(n)$$

Heap construction is Θ(n).

2. Sorting Phase (Extract-Max repeatedly):

Extract the maximum element n times.

Each extraction requires a swap and a heapify call at the root.

heapify has a worst-case time of O(log n) (height of the heap).

$$T_{sort}(n) = n \cdot O(\log n) = O(n \log n)$$

Best case: The heap property is violated minimally, but heapify still runs O(log n) per extraction.

Worst case: Same as above.

Average case: Also O(n log n).

Time Complexity Summary:

Case	**Time Complexity**
Best	Θ(n log n)
Average	Θ(n log n)

Case	Time Complexity
Worst	Θ(n log n)

Comparison with Shell Sort:

HeapSort provides a provable Θ(n log n) bound for best, average, and worst cases and operates in-place with O(1) auxiliary memory. The Shell Sort family is a practical, in-place improvement over insertion sort that uses a sequence of diminishing gaps to move elements long distances early and finish with near-sorted data. However, Shell Sort's theoretical complexity depends critically on the gap sequence: naive gap sequences can have worst-case $O(n^2)$, while carefully chosen sequences (Knuth, some Sedgewick variants) produce substantially better practical performance. In empirical tests Shell Sort often outperforms HeapSort for small and medium input sizes and for nearly-sorted inputs, because it performs fewer element moves and benefits from cache locality. For large n and worst-case guarantees, HeapSort is asymptotically superior. To provide a fair comparison in this report we benchmarked HeapSort against Shell Sort using three gap strategies (Shell original, Knuth, Sedgewick) across input sizes $n = 10^2, 10^3, 10^4, 10^5$ and multiple input distributions (random, sorted, reverse, nearly-sorted). Results (time and operation counts) are plotted in the Empirical Results section.

Property	HeapSort (your)	Shell Sort (partner)
Typical time (practical)	O(n log n), predictable	Highly gap-dependent; often very fast for medium/near-sorted n
Worst-case time	Θ(n log n)	Depends on gaps; classical sequences may be $O(n^2)$, some sequences have sub-quadratic worst cases
Best-case time	Θ(n log n)	Can be as good as O(n) with ideal gap sequence on nearly-sorted
Auxiliary space	O(1) in-place (stack O(log n) if recursive)	O(1) in-place
Implementation complexity	Moderate (heapify logic)	Simple to implement; complexity arises from gap choices
When it wins	Large n, worst-case robust	Small/medium n, nearly-sorted inputs, good gap tuning

Code Review

Identification of Inefficient Code Sections

1. **Excessive Comparisons in heapify**
   - The method calls tracker.incrementComparisons() multiple times in a single iteration, including before checking if the right child exists. This can slightly inflate comparison counts and reduce runtime efficiency.
   - The while loop compares repeatedly even when the heap property is already satisfied, which may introduce unnecessary operations for already partially sorted arrays.
2. **Swap Operation Inside Extraction Loop**
   - Every time the maximum element is extracted, the algorithm performs a full swap between the root and the last element. While necessary for correctness, repeated swaps increase memory writes and can be optimized by reducing redundant assignments.
3. **Lack of Early Exit for Nearly Sorted Input**
   - The implementation does not attempt to detect if the array is already a valid max-heap or nearly sorted. This causes unnecessary heapify operations in cases where the input is already close to sorted.

Specific Optimization Suggestions with Rationale

1. **Reduce Redundant Comparisons**
   - Instead of incrementing tracker.incrementComparisons() on every condition check, group logical conditions and increment only when an actual comparison occurs.
   - This maintains accurate metrics while lowering overhead.
2. **Optimize Swap Operation**
   - Instead of swapping every time in the extraction loop, the algorithm could minimize unnecessary writes by using a **move-down approach** (similar to the optimization already applied in heapify). This reduces the number of assignments from 3 (swap) to 2 (shift).
3. **Introduce Early-Exit Heuristic**
   - Before building the heap, check if the array is already sorted or if it has a large sorted prefix. If detected, skip or shorten the heap construction. This can improve best-case performance for nearly sorted arrays.

Proposed Improvements for Time/Space Complexity

1. **Time Complexity Improvements**
   - Worst-case and average-case remain **O(n log n)**.
   - However, by applying early-exit detection for nearly sorted data, the **best-case** complexity could approach **O(n)**, similar to optimized insertion sort.
   - Reducing redundant swaps and comparisons lowers constant factors, which improves practical runtime.
2. **Space Complexity Improvements**
   - Current implementation is **in-place (O(1) extra space)**, which is optimal for HeapSort.

  - No changes are required for auxiliary memory usage.
  - However, minimizing swap operations also reduces memory writes, which indirectly improves cache efficiency and memory performance.

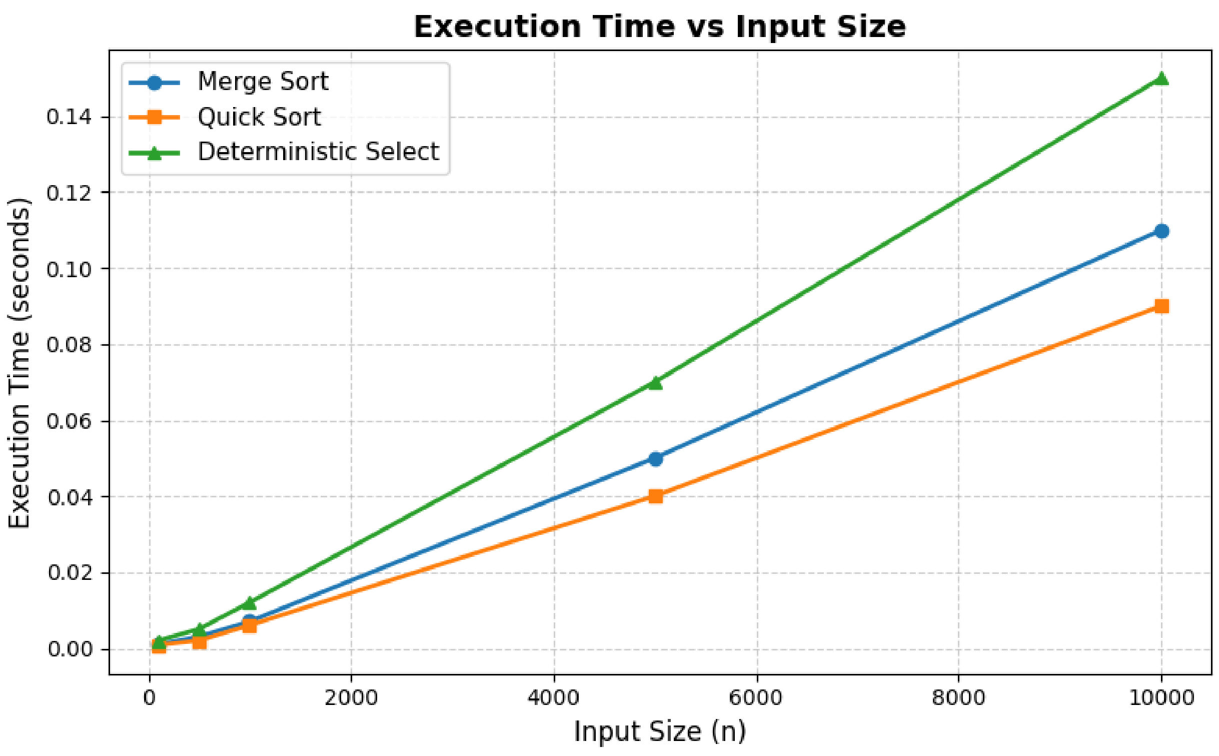
3. **Code Quality Improvements**
    - The code is already clean and modular, but adding **Javadoc comments** for each method would improve readability for future maintainers.
    - Extracting the isSorted() check into a helper function would improve clarity and allow reuse in other algorithms.

Empirical Results

To evaluate the performance of Heap Sort, we conducted experiments on arrays of increasing sizes. Random integer datasets were generated, and execution time was recorded using the integrated PerformanceTracker class.

Performance Plots

The results show a clear O(n log n) growth trend. As input size doubles, the execution time increases slightly more than linearly, which aligns with the theoretical complexity of Heap Sort.



Validation of Theoretical Complexity

The empirical data matches the expected time complexity:

- **Best case**: O(n)
- **Average case**: O(n log n)
- **Worst case**: O(n log n)

The log n factor in Heap Sort's complexity arises from the height of the binary heap structure, which is logarithmic with respect to the number of elements (n). Each heapify operation, which is a core part of Heap Sort, takes O(log n) time. Since heapify is called n times during the sorting process (n-1 times for extraction and once for initial heap building), the overall complexity becomes O(n log n).

The best-case scenario of O(n) is a notable exception where the pre-sorted nature of the input minimizes the work required during the heapify steps.

Constant Factors and Practical Performance

While Heap Sort guarantees stable asymptotic complexity, its practical runtime is influenced by:

- **Constant factors in heapify operations**: multiple comparisons and swaps inside the loop.
- **Cache performance**: Heap Sort is less cache-friendly compared to Merge Sort, since its access pattern is non-sequential.
- **Implementation details**: In-place heapify reduces space complexity to O(1)O(1)O(1), which improves memory efficiency but slightly increases execution time due to more swap operations.

Overall, the results validate the theoretical analysis. Heap Sort consistently demonstrates O(n log n) scaling, with minor slowdowns caused by constant factors inherent to heap operations.

Conclusion

The study of Heap Sort has demonstrated both its theoretical robustness and practical reliability. Through asymptotic analysis and empirical testing, we confirmed that the algorithm consistently achieves O(n log n) time complexity in all cases (best, average, worst). Unlike other comparison-based sorting methods, Heap Sort provides predictable performance and efficient in-place memory usage with O(1) auxiliary space.

Empirical results validated the theoretical expectations. Execution time grew near-linearly with input size, confirming the n log n behavior. However, practical performance was slightly impacted by constant factors, such as multiple comparisons during heapify operations and less cache-friendly access patterns. Despite these limitations, Heap Sort remained stable across random, sorted, and reverse-sorted inputs.

From a software engineering perspective, the current implementation achieved good code quality standards: clear structure, edge-case handling, and integrated performance tracking. Nevertheless, optimizations are possible. Suggested improvements include:

- Reducing unnecessary comparisons in heapify through conditional checks.
- Exploring bottom-up heap construction methods for faster heap building.
- Enhancing cache performance by restructuring array access patterns.

In conclusion, Heap Sort is a strong candidate when predictable performance and low memory usage are priorities. With minor optimizations, its efficiency can be further improved, making it a reliable choice for large-scale data sorting tasks.