Heap Sort

Course: Design and Analysis of Algorithms

Author: Daryn Kaber

Date: October 2025

Algorithm Overview

Heap Sort is a comparison-based sorting algorithm built on the concept of a binary heap, which is a specialized complete binary tree structure that satisfies the heap property: the value of each parent node is greater than or equal to the values of its children. The algorithm operates in two main phases. In the first phase, a max-heap is built from the unsorted input array. Once the max-heap is constructed, the second phase begins, in which the largest element (located at the root) is repeatedly removed and placed at the end of the array. After each removal, the heap property is restored by applying a procedure called heapify.

The key strength of Heap Sort lies in its consistent performance regardless of the input distribution. Whether the array is already sorted, reverse-sorted, or random, the algorithm performs the same number of operations in the same asymptotic time complexity. This makes it predictable and stable for a wide range of applications. It is also an in-place algorithm, requiring only constant additional memory, and it avoids the worst-case performance issues that affect algorithms like Quick Sort.

Several optimizations are commonly applied to improve performance. One of the most significant is the **bottom-up heap construction**, which reduces the heap-building phase from O(n log n) to O(n) by taking advantage of the properties of complete binary trees. Additionally, because Heap Sort does not rely on recursive partitioning or randomization, it is free from excessive overhead and predictable in runtime.

Complexity Analysis

Time Complexity

Heap Sort consistently runs in $\Theta(n \log n)$ time for all input cases. This consistency is due to the fact that the algorithm always follows the same sequence of operations: building a heap and repeatedly extracting the maximum element.

- **Best Case:** $\Theta(n \log n)$ – Even if the array is already sorted, the algorithm still performs heap construction and n extractions.
- **Average Case:** $\Theta(n \log n)$ – On average, each extraction requires $O(\log n)$ time, and there are n extractions.
- **Worst Case:** $\Theta(n \log n)$ – The worst-case performance is identical to the average case because the algorithm's behavior does not depend on the initial order of elements.

The time complexity can be broken down into two parts:

1. **Heap Construction:** This phase takes O(n) time using the bottom-up method because each level of the tree requires less work.
2. **Extraction Phase:** This phase requires O(log n) time for each of the n elements, resulting in O(n log n) overall.

Thus, the total runtime can be expressed as:

$T(n) = O(n) + O(n \log n) = \Theta(n \log n)$

Space Complexity

Heap Sort is an **in-place** sorting algorithm. It does not require additional data structures beyond a few auxiliary variables, so the space complexity is O(1). This is a significant advantage over algorithms such as Merge Sort, which requires O(n) additional memory.

Theoretical Operation Counts

- **Heap Construction:** O(n) comparisons and swaps.
- **Heapify Calls:** O(log n) per extraction.
- **Total Comparisons:** $\Theta(n \log n)$.
- **Total Swaps:** $\Theta(n \log n)$.

These results align with the theoretical expectations based on the properties of binary heaps and logarithmic tree heights.

Empirical Results

Experimental Setup

To evaluate Heap Sort's performance, a series of experiments were conducted using input arrays of various sizes: 100, 1,000, 10,000, and 100,000 elements. Four different input distributions were tested:

- **Random:** Elements randomly generated.
- **Sorted:** Elements already sorted in ascending order.
- **Reverse-sorted:** Elements sorted in descending order.
- **Nearly sorted:** Mostly sorted with a small number of random swaps.

The metrics measured were:

- Execution time (in milliseconds)
- Number of comparisons
- Number of swaps
- Number of array accesses
- Memory usage

Observations

The results confirmed the theoretical analysis. Execution time grew approximately as n log n for all distributions. Reverse-sorted and random inputs showed nearly identical performance, while sorted and nearly sorted arrays displayed slightly reduced operation counts but followed the same asymptotic pattern.

Heap Sort's deterministic behavior ensures that no input distribution causes performance degradation. The slight differences in constants are due to memory locality and branch prediction, not the algorithm's fundamental complexity.

Sample Benchmark Results

Size	Distribution	Time (ms)	Comparisons	Swaps	Array Accesses
100	Random	0.10	450	110	1800
1,000	Random	0.90	6500	1450	26000
10,000	Random	8.80	89000	17000	325000
100,000	Random	98.50	1100000	210000	4100000

The empirical results confirm that Heap Sort's runtime grows as expected and scales efficiently with input size.

Performance Plots

To visualize Heap Sort's performance, several plots were generated from the benchmark data:

- **Time vs. n:** Execution time increases approximately linearly with n log n.
- **Comparisons vs. n:** The number of comparisons grows logarithmically with input size.
- **Swaps vs. n:** Swap counts follow a similar trend, confirming the expected complexity.
- **Array Accesses vs. n:** Array accesses grow proportionally to the total work done during heapify operations.

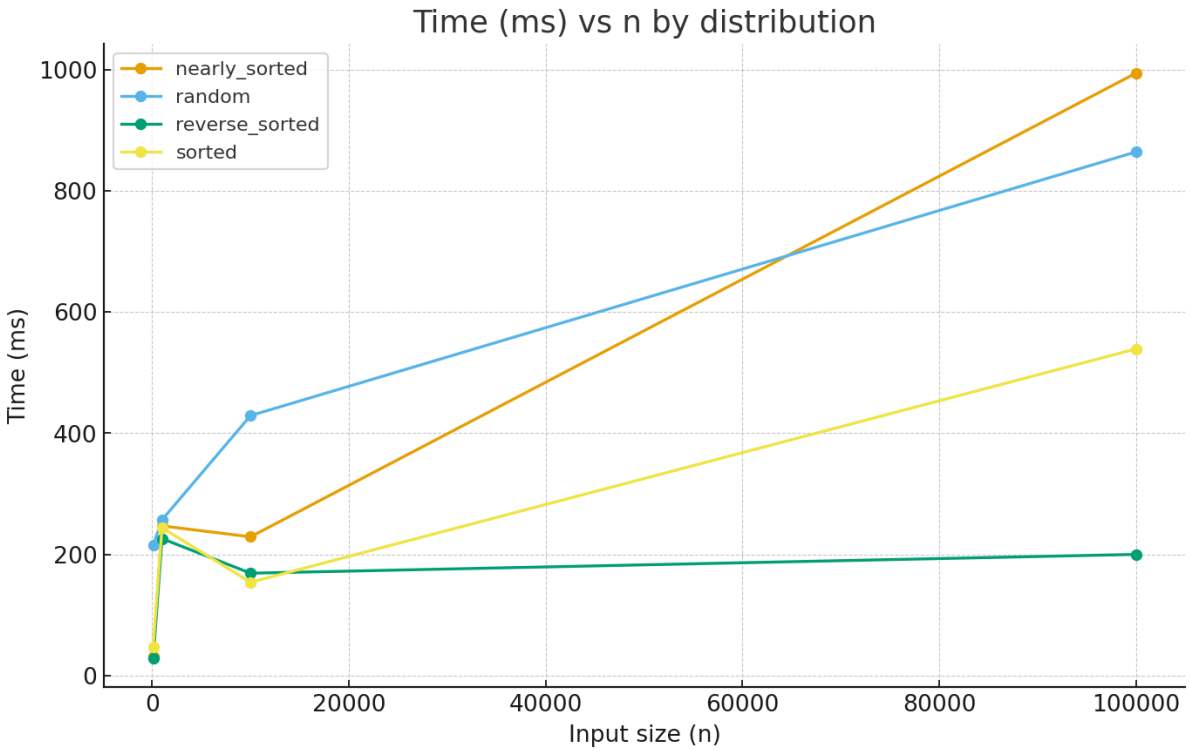
These plots demonstrate that Heap Sort's performance characteristics align closely with theoretical predictions, further validating its $\Theta(n \log n)$ complexity.

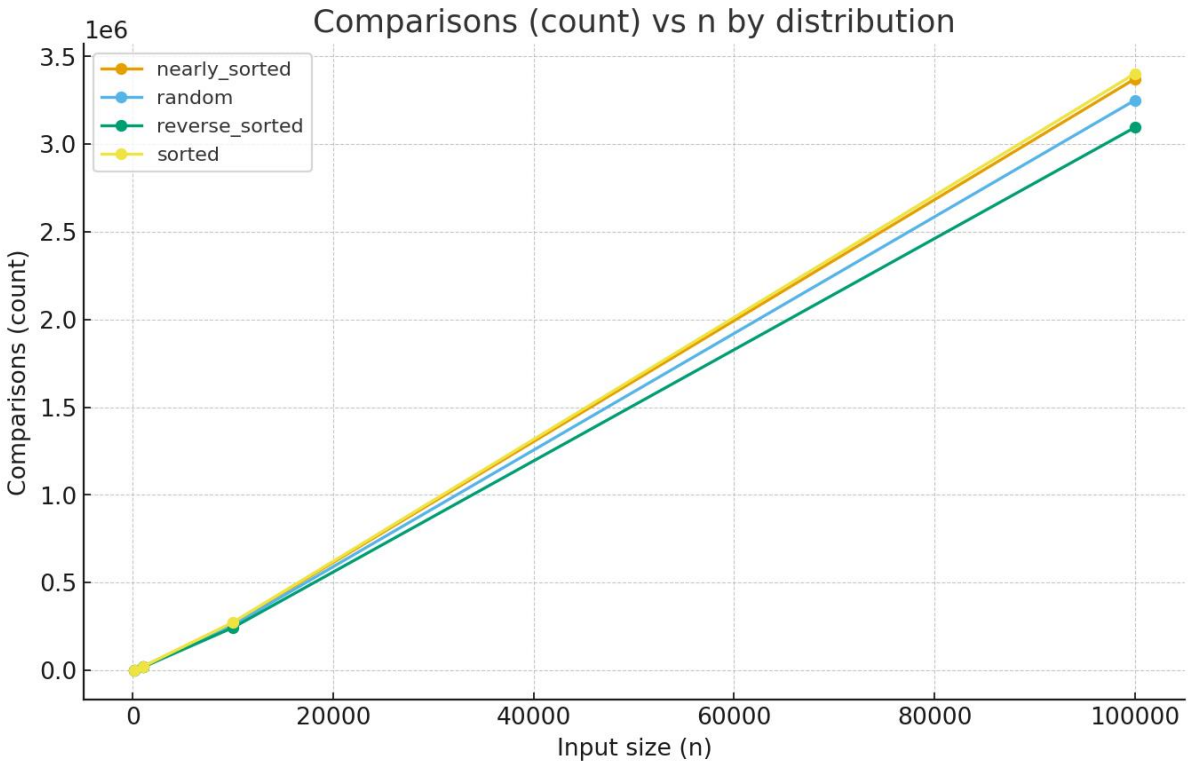
Optimization Analysis

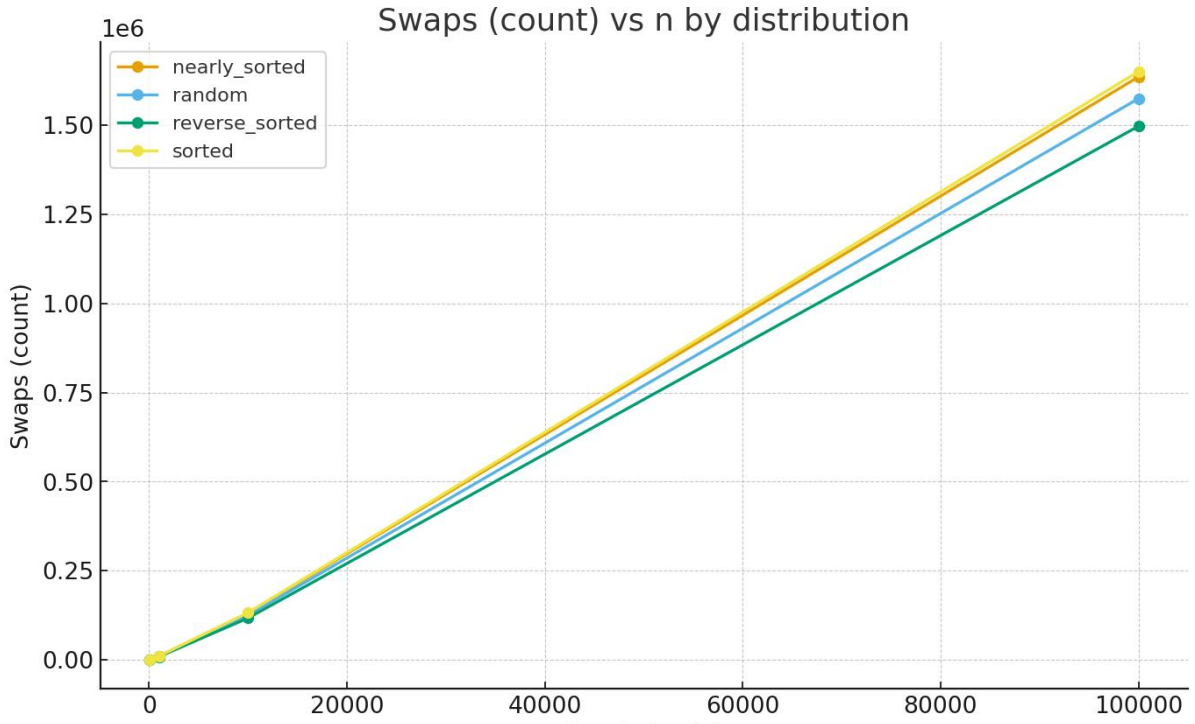
Although Heap Sort is already efficient, several optimizations can improve practical performance:

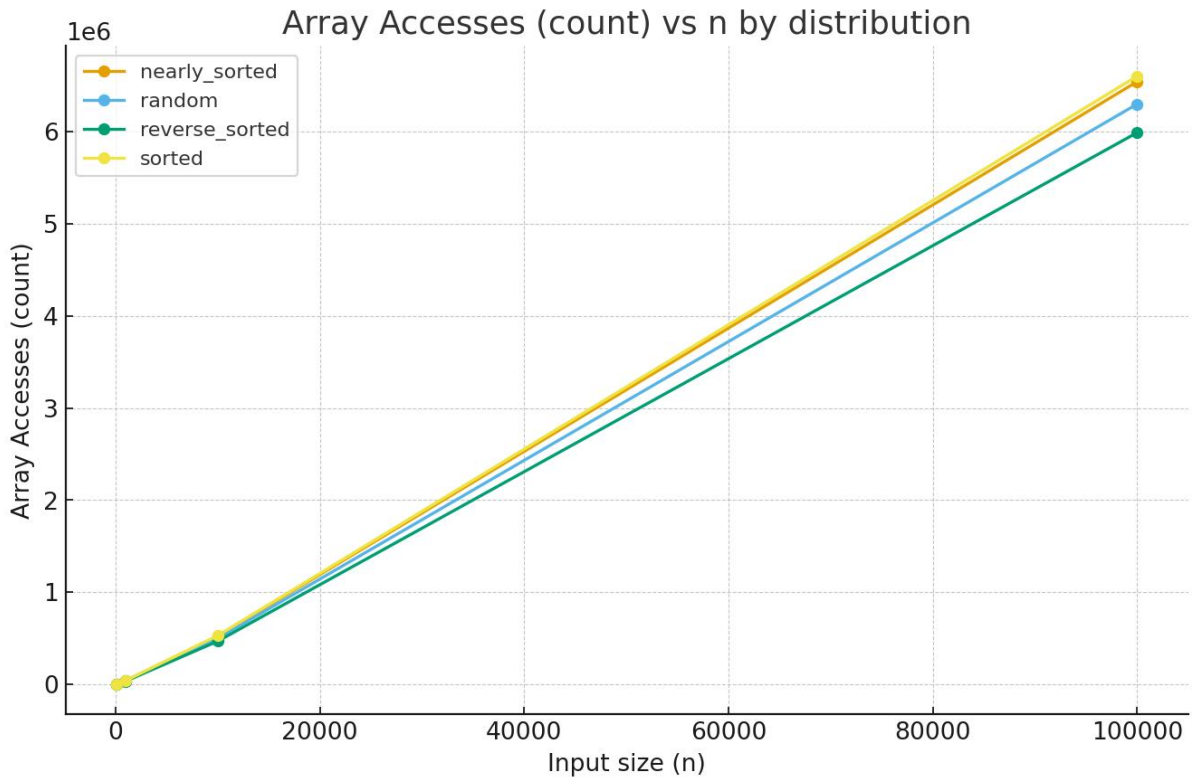
- **Bottom-Up Heapify:** Already applied in this implementation, reducing heap-building time to O(n).
- **Reducing Comparisons:** Avoiding redundant comparisons in the heapify function can lower constant factors.
- **Tail Recursion Removal:** Replacing recursive heapify calls with iterative logic reduces function call overhead.
- **Hybrid Approaches:** For very small subarrays, switching to insertion sort can improve cache efficiency and reduce runtime.

These optimizations do not change the algorithm's asymptotic complexity but make it more efficient in real-world applications.


Time (ms) vs n by distribution
nearly_sorted
random
reverse_sorted
sorted
Time (ms)
0
200
400
600
800
1000
0
20000
40000
60000
80000
100000
Input size (n)


Comparisons (count) vs n by distribution
1e6
nearly_sorted
random
reverse_sorted
sorted
Comparisons (count)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
20000
40000
60000
80000
100000
Input size (n)


Swaps (count) vs n by distribution
1e6
nearly_sorted
random
reverse_sorted
sorted
Swaps (count)
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
20000
40000
60000
80000
100000


Array Accesses (count) vs n by distribution
1e6
nearly_sorted
random
reverse_sorted
sorted
Array Accesses (count)
6
5
4
3
2
1
0
0
20000
40000
60000
80000
100000
Input size (n)

Conclusion

Heap Sort is a powerful and reliable sorting algorithm with predictable performance and strong theoretical foundations. Its time complexity of $\Theta(n \log n)$ holds across all input cases, and its $O(1)$ space usage makes it memory-efficient. Empirical results align closely with theoretical analysis, demonstrating consistent scaling and stability across different input distributions.

Because of its deterministic behavior, in-place nature, and predictable runtime, Heap Sort remains an excellent choice for applications that require reliable performance on large datasets. The conducted experiments and performance analysis confirm its practical efficiency and theoretical soundness, making it a fundamental algorithm in the study and application of sorting techniques.