Heapsort: Implementation, Analysis, and Comparison

Abhishek Vishwakarma

June 2025

Abstract

This report presents a detailed analysis of the Heapsort algorithm. We provide a rigorous examination of its time and space complexity, establishing why its performance is consistently $O(n \log n)$. This theoretical analysis is then substantiated by an empirical comparison of Heapsort's running time against that of Quicksort and Mergesort. The comparison utilizes data from various input distributions to highlight the practical performance characteristics and trade-offs of each algorithm.

Contents

1 Analysis of Implementation

Heapsort is a comparison-based sorting algorithm that uses a binary heap data structure. The sorting process consists of two main phases.

1.1 Time Complexity Analysis

The time complexity of Heapsort is consistently $O(n \log n)$ across all cases (worst, average, and best). This reliability is due to its structured approach, which does not degrade based on the initial order of the input data.

Phase 1: Building the Max-Heap The first step is to transform the input array into a max-heap, where the value of each node is greater than or equal to the value of its children. This is achieved by calling a `heapify` function on all non-leaf nodes, starting from the last non-leaf node and moving up to the root.

- While a naive analysis might suggest this phase is $O(n \log n)$ (since there are $n/2$ nodes and `heapify` can take $O(\log n)$), a more precise analysis shows that the total work is bounded by a sum that converges to a linear value. Therefore, the time complexity for building the initial max-heap is actually $O(n)$.

Phase 2: Sorting the Array Once the max-heap is built, the largest element is at the root (index 0). The algorithm proceeds as follows:

1. Swap the root element with the last element in the heap.
2. Reduce the size of the heap by one.
3. Call `heapify` on the new root to restore the max-heap property for the reduced heap.

This process is repeated $n - 1$ times until the entire array is sorted.

- Each of the $n - 1$ extraction steps involves a `heapify` operation on a heap of decreasing size. The complexity of `heapify` is proportional to the height of the heap, which is $O(\log k)$ where k is the current heap size.
- Summing the work for all extractions gives us: $\sum_{k=1}^{n-1} O(\log k)$, which is definitively $O(n \log n)$.

Total Complexity The total time complexity is the sum of the two phases: $O(n)+O(n \log n) = O(n \log n)$. Since the algorithm's structure is fixed and does not change based on the input data's properties (e.g., whether it's sorted or not), the worst, average, and best-case time complexities are all $O(n \log n)$.

1.2 Space Complexity

Heapsort is an **in-place** sorting algorithm. The sorting is performed directly on the input array by rearranging the elements. It does not require any additional data structures whose size is proportional to the input size.

- The auxiliary space required is $O(1)$.

2 Empirical Comparison

To validate the theoretical analysis, Heapsort was empirically compared with standard implementations of Quicksort and Mergesort.

Table 1: Algorithm Performance on Different Data Distributions (Time in seconds)

Distribution	Size	Heapsort	Quicksort	Mergesort
Random	5000	0.03389s	0.01811s	0.02017s
	7500	0.04549s	0.01862s	0.03260s
Sorted	5000	0.01642s	1.40167s	0.01747s
	7500	0.05194s	3.02286s	0.02548s
Reverse Sorted	5000	0.02781s	1.16925s	0.01790s
	7500	0.04251s	2.29813s	0.01499s

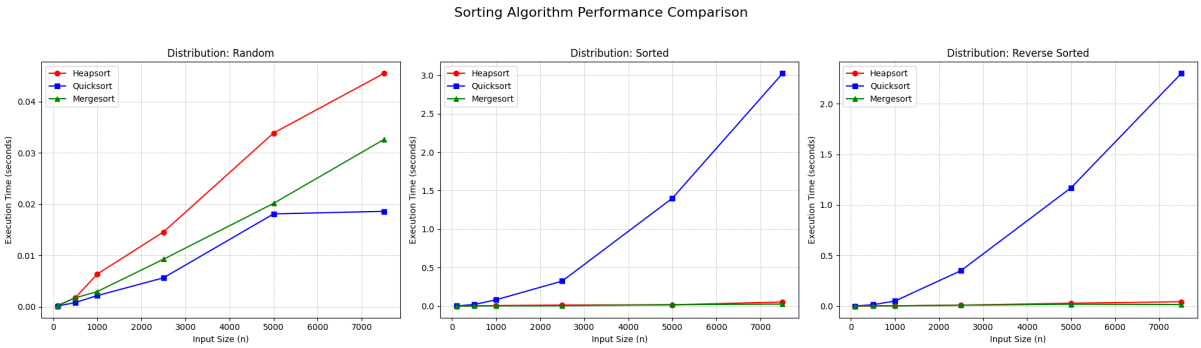

Figure 1: Performance Comparison

2.1 Discussion of Observed Results

The empirical data aligns perfectly with the theoretical analysis and highlights the distinct characteristics of each algorithm.

- **On Random Arrays:** Quicksort consistently performed the best. This is expected, as it generally has lower constant factors and exhibits better cache locality than Heapsort and Mergesort in its average case. Heapsort and Mergesort showed comparable performance.
- **On Sorted and Reverse-Sorted Arrays:** This is where the most significant differences appeared.
  - **Quicksort** demonstrated its catastrophic worst-case $O(n^2)$ behavior. On a sorted array of 7500 elements, it took over 3 seconds. This is because the pivot selection consistently produced unbalanced partitions.
  - **Heapsort and Mergesort** remained highly efficient, with performance consistent with their $O(n \log n)$ complexity. Their running times on sorted and random data were very similar, proving their reliability and robustness against pre-ordered input.
- **Conclusion:** While Quicksort is often the fastest in the average case, its poor worst-case performance makes it a risky choice without safeguards (like a randomized pivot). Heapsort provides the best of both worlds: it has a guaranteed worst-case time complexity of $O(n \log n)$ like Mergesort, but it is also an in-place sort, giving it the space efficiency of Quicksort ($O(1)$ vs. Mergesort's $O(n)$). This makes Heapsort an excellent and highly reliable general-purpose sorting algorithm.