# Assignment 4: Heap Data Structures: Implementation, Analysis, and Applications

Ronit Pawar

005040714

University of Cumberlands

Algorithms and Data Structures

Satish Penmatsa

30 September 2025

## Introduction

Heapsort is a comparison-based sorting algorithm that relies on the properties of a binary heap to organize and extract elements in sorted order. Its implementation in this assignment follows the classical two-phase approach: (1) constructing a max heap from the input array and (2) repeatedly extracting the maximum element while restoring the heap property. A rigorous analysis of its computational complexity is presented below.

## Time Complexity

The process of building a max heap involves calling the heapify procedure on all non-leaf nodes. Although a single call to heapify has a worst-case complexity of , not all nodes require the same amount of work. Nodes near the leaves require minimal adjustments, while nodes closer to the root require more. Summing across all nodes, the total cost of building the heap is bounded by rather than (Cormen et al., 2009).

After the heap is constructed, the algorithm repeatedly extracts the maximum element, swaps it with the last element, and reduces the heap size. Each extraction requires a single heapify operation, which takes time because the height of the heap is . Since this step is performed times, the cost of the extraction phase is . The total runtime is the sum of the heap construction and extraction phases:

Importantly, this bound holds in the best, average, and worst cases. Unlike Quicksort, which can degrade to with unfavorable pivot choices, Heapsort consistently executes extractions and maintains the heap property in logarithmic time per extraction. Thus, its time complexity is invariant to input order. The uniformity of Heapsort’s complexity arises from the deterministic structure of the heap. Each heapify operation, regardless of the data distribution, traverses at most the height of the heap, which is bounded by . Since these operations are repeated times, the algorithm’s performance does not fluctuate across different input arrangements, ensuring a predictable runtime in all scenarios.

## Space Complexity

One of the strengths of Heapsort is its low memory overhead. Because the heap is maintained in the same array that stores the input, no additional data structures are required beyond a constant number of temporary variables for swapping. Therefore, the algorithm operates with auxiliary space, making it more space-efficient than algorithms such as Merge Sort, which requires additional space. The only potential overhead arises from recursion in the heapify procedure, which has a maximum depth of . This can be mitigated by employing an iterative implementation of heapify, further reinforcing Heapsort’s efficiency in space utilization.

## Results and Analysis

#### Heapsort Demonstration

The initial test of the Heapsort algorithm on a small array confirmed correct functionality.  
Original array: [24, 20, 9, 54, 33, 9, 20, 58, 51, 79]. Sorted array (ascending order):  
[9, 9, 20, 20, 24, 33, 51, 54, 58, 79]. This validates that the implementation correctly builds a max-heap and repeatedly extracts the maximum element to produce a sorted sequence.

#### Priority Queue Scheduler Simulation

Ten tasks with random priorities were inserted into a max-heap priority queue. The scheduler extracted them in descending priority order, demonstrating correct behavior of the insert and extract\_max operations.

* **Generated priorities:** T1=90, T2=61, T3=9, T4=94, T5=31, T6=82, T7=69, T8=38, T9=58, T10=87
* **Execution order:** T4 (94) → T1 (90) → T10 (87) → T6 (82) → T7 (69) → T2 (61) → T9 (58) → T8 (38) → T5 (31) → T3 (9)

This confirms that higher-priority tasks are always executed first, which is consistent with the intended max-heap design.

#### Sorting Algorithm Performance Comparison

Execution times (in seconds) for sorting arrays of increasing size:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input Size | Heap Sort | Quick Sort | Merge Sort | Tim Sort |
| 1000 | 0.00124s | 0.00068s | 0.00101s | 0.00007s |
| 5000 | 0.00838s | 0.00397s | 0.00618s | 0.00041s |
| 10000 | 0.01900s | 0.00841s | 0.01414s | 0.00097s |

### Analysis

Heapsort consistently performed in O(n log n) time, but it was slower than Quicksort and Merge Sort in practice due to more comparisons and swaps per operation. Quicksort was the fastest among the manual implementations. Although its worst-case complexity is O(n²), with random pivot selection it typically outperforms Heapsort. Merge Sort also showed O(n log n) behavior but with a slightly higher overhead because of recursion and merging steps. Timsort (Python’s built-in sort) was by far the fastest, leveraging a hybrid algorithm that exploits real-world data patterns and optimizations.

### Key Takeaways

Heapsort is predictable (O(n log n) worst/average/best case) and uses constant extra space, making it valuable in constrained environments. Quicksort is usually faster in practice due to cache efficiency and fewer data movements but has poor worst-case behavior without optimizations. Merge Sort guarantees O(n log n) and is stable, making it useful when stability is required. Timsort outperforms all others in Python because it is highly optimized for real-world data patterns, combining merge sort and insertion sort. Priority queues with heaps efficiently handle scheduling tasks by always retrieving the highest-priority job in O(log n) time.

# References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms, third edition*. http://portal.acm.org/citation.cfm?id=1614191