# Assignment 4: Heapsort Implementation and Analysis

Ashim Sedhain

University of the Cumberlands

MSCS-532-M20: Algorithms and Data Structures

June 15, 2025

# Heapsort Implementation and Analysis

## Implementation

In designing the Heapsort algorithm, the decision to use a max-heap over a min-heap aligns with the classic Heapsort structure, where the largest element is repeatedly extracted and placed at the end of the array. The heap is implemented using an array-based binary tree, leveraging the parent-child index relationships. This representation allows constant-time access to child and parent nodes without additional memory overhead from using explicit tree nodes or pointers. The implementation itself is broken down into two primary operations:

- Heap Construction: Bottom-up heapification from the last non-leaf node to the root. This is efficient because lower levels require less work, resulting in linear time complexity.
- Heap Sort Phase: Each iteration involves swapping the root (maximum element) with the last element in the unsorted section and reducing the heap size. This maintains in-place sorting.

## Analysis of Heapsort

### Time Complexity

Heapsort consistently operates in $O(n \log n)$ time, regardless of the input distribution. Building the Max-Heap takes $O(n)$ time, not $O(n \log n)$, because the cost of heapifying nodes decreases exponentially down the tree. Specifically, most nodes are at the bottom of the tree and require fewer operations to maintain the heap property.

After the heap is built, we perform $n$ extractions, each of which involves a call to heapify that takes $O(\log n)$ time in the worst case. Therefore, this phase takes $O(n \log n)$ time.

Since the build and extract phases are sequential, the overall time complexity remains

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

Best, Worst, and Average Cases: All are $O(n \log n)$ because the structure of the heap and the extraction process do not depend on the initial ordering of the array.

**Space Complexity**

Heapsort is an in-place sorting algorithm. It does not require any auxiliary arrays. Therefore, its space complexity is $O(1)$. There is no recursion stack overhead as heapify is implemented iteratively, and only a constant amount of extra space is used.

**Empirical Comparison with Quicksort and Merge Sort**

**Experimental Setup**

All sorting algorithms were implemented in Python and benchmarked using time.perf_counter() on input arrays of varying characteristics:

- Random
- Sorted
- Reverse-sorted

**Summary of Experiment**

For fairness, built-in sorted() (which uses Timsort) was avoided. Custom implementations of Merge Sort and Quicksort were used.

| **Input Type** | **Heapsort (ms)** | **Quicksort (ms)** | **Merge Sort (ms)** |
|---|---|---|---|
| Random | 0.919 | ~0.6 | ~0.8 |
| Sorted | 0.977 | ~1.5 (worst case) | ~0.8 |
| Reverse-sorted | 0.795 | ~1.5 (worst case) | ~0.8 |
| Repeated | 0.140 | ~0.7 | ~0.8 |

**Observations**

- Quicksort: Faster than Heapsort in random input scenarios due to lower constant factors. However, without randomized pivot selection, its performance degrades on sorted or reverse-sorted inputs $O(n^2)$. This was observed in the higher runtimes.
- Merge Sort: Consistent $O(n \log n)$ performance across all distributions. It is not in-place (requires $O(n)$ space), which can be a disadvantage for memory-constrained systems.
- Heapsort: Maintains consistent performance across all input types, validating the theoretical guarantee of $O(n \log n)$ in all cases. However, it tends to be slower in practice due to the overhead of repeated swaps and poorer cache performance.