Implementation and Analysis Report of Heap Sort

I implemented a standard heap sort algorithm using the binary heap data structure. This implementation mainly consists of the following two functions:

1. heapify(arr, n, i): Converts a subtree rooted at index i into a max heap, which means the largest value comes at the top of the triangle.
2. heapsort(arr): First, the initial for loop converts the entire array into a heap structure. The second for loop repeatedly extracts the maximum value to perform sorting.

As described above, the heap sort algorithm consists of two phases: the heap construction phase that converts the input array into a max heap, and the sorting phase that repeatedly extracts the maximum value from the max heap and places it at the end of the array. According to Mohammadagha (2025), the time complexity of heap sort is O(n log n) in worst, average, and best cases, making it a consistent and stable algorithm regardless of the initial distribution of input data.

The time complexity breaks down as follows:

1. Heap construction phase: O(n)

   Intuitively, it seems like O(n log n) complexity because each heapify operation takes O(log n) and is performed on n/2 elements, but since nodes with smaller depth require less processing, it is actually O(n).
2. Sorting phase: O(n log n)

   Performs n - 1 extraction operations, each executing a heapify with O(log n).

Therefore, according to Mohammadagha's (2025) formula, the overall time complexity of heap sort is:

$$THeapsort(n) = Tbuild(n) + Textract(n) = O(n) + O(n\,log\,n) = O(n\,log\,n)$$

As shown, heap sort performs the same number of heap operations regardless of the initial arrangement of input data, resulting in consistent O(n log n) performance in all cases. This means it is an algorithm that does not degrade in performance even in the worst case, unlike other algorithms such as quicksort.

Additionally, Mohammadagha (2025) explains that "in-place algorithms" like heap sort are superior in terms of space efficiency compared to algorithms that require additional memory, such as mergesort. Since it directly modifies the input array in-place without requiring additional arrays during sorting, it provides excellent space complexity of O(1). However, if implemented recursively, it might require O(log n) stack space, which, in the worst case, is equivalent to the height of the tree. To avoid this, an iterative implementation can be used instead of recursion.

Next, I implemented heap sort in my environment and compared its performance with the randomized quicksort that I implemented in Assignment 3. The results are as follows:

Input Size	Heapsort (Random)	Quicksort (Random)	Heapsort (Sorted)	Quicksort (Sorted)	Heapsort (Reverse)	Quicksort (Reverse)
100	0.000144	0.000064	0.000157	0.00012	0.000076	0.000065
500	0.000641	0.000407	0.00067	0.000423	0.000771	0.000365
1000	0.001572	0.001184	0.001594	0.000769	0.001315	0.001078
5000	0.010539	0.006144	0.013552	0.005777	0.009309	0.006274
10000	0.023948	0.012867	0.023566	0.0102	0.021791	0.01187

These results are consistent with Mohammadagha's (2025) findings. In Mohammadagha's study, for an input size of 100,000, heap sort executed in 0.203 seconds while quicksort executed in 0.089 seconds, indicating that quicksort is about 2-3 times faster than heap sort.

In my experiment, I divided the execution time of heap sort by that of quicksort:

Execution Time (seconds) for Random Distribution

Input Size	Heapsort	Quicksort	Ratio (Heap/Quick)
100	0.000144	0.000064	2.25
500	0.000641	0.000407	1.57
1000	0.001572	0.001184	1.33
5000	0.010539	0.006144	1.72
10000	0.023948	0.012867	1.86

Execution Time (seconds) for Sorted Distribution

Input Size	Heapsort	Quicksort	Ratio (Heap/Quick)
100	0.000157	0.00012	1.31
500	0.00067	0.000423	1.58
1000	0.001594	0.000769	2.07
5000	0.013552	0.005777	2.35
10000	0.023566	0.0102	2.31

Execution Time (seconds) for Reverse Sorted Distribution

Input Size	Heapsort	Quicksort	Ratio (Heap/Quick)
100	0.000076	0.000065	1.17
500	0.000771	0.000365	2.11
1000	0.001315	0.001078	1.22
5000	0.009309	0.006274	1.48
10000	0.021791	0.01187	1.84

As shown above, I found that quicksort is on average 1.5 times faster, and at maximum more than 2 times faster than heap sort across all data sizes and array types. Mohammadagha (2025) states that while the theoretical time complexity of both is O(n log n), quicksort often performs faster because it has smaller constant factors. My experimental results also suggest that quicksort was generally faster due to the relationship of constant factors.

However, quicksort has a worst-case time complexity of $O(n^2)$, and although I used randomized quicksort in my experiment, which performed faster than heap sort even for sorted arrays, there is still a possibility of encountering worst-case patterns. Therefore, I believe heap sort is effective when guaranteed worst-case performance is important. Additionally, like quicksort, heap sort can sort in-place, making it usable in applications with memory constraints. I found that it is a very stable and robust algorithm that provides consistent memory usage and performance.

References

Mohammadagha, M. (2025). Hybridization and Optimization Modeling, Analysis, and Comparative Study of Sorting Algorithms: Adaptive Techniques, Parallelization, for Mergesort, Heapsort, Quicksort, Insertion Sort, Selection Sort, and Bubble Sort. Department of Civil Engineering, University of Texas at Arlington.
https://engrxiv.org/preprint/download/4537/7883