Richa Lahoti
005039141
MSCS-532-M20
Assignment 4

# Part 1: Heapsort Analysis

### Heapsort Algorithm Implementation

The heapsort algorithm implementation is in the github repo linked here: https://github.com/rlahoti-cb/MSCS532_Assignment4. The README.md covers how to run this code to generate the data that is used in the analysis section below. Note, that your numbers may vary slightly due to running the code on a different machine and other variables.

### Analysis

We can begin by analyzing the time complexity of heapsort in the worst, average, and best cases. We can first begin by explicitly stating that the extract operation takes O(log n) in a heap structure. In the best case, the time complexity is O(n log n) because even if the input data is sorted, we still have to build the heap structure and running the extract operation for all n elements results in O(n log n). In the average case, we still run into O(log n) for each extraction across n elements resulting in O(n log n). Furthermore, in the worst case we note the same behavior and time complexity.

In terms of space complexity, my implementation takes O(n) space since we are only modifying the input array with moving elements around. No additional space or memory is required to operate on the heap data structure. This is the value of being able to represent the heap structure as a python list and reference nodes using indices.

### Comparison

In this section, we report the data from our analysis of the heapsort, mergesort, and quicksort algorithms across various data sizes (100K elements, 1M elements) and types of data (random, sorted, reverse sorted).

| Algorithm | Data Type | Data Size | Time Taken (s) |
|---|---|---|---|
| Heapsort | Random | 100000 | 0.491620779 |
| Heapsort | Sorted | 100000 | 0.4430651665 |
| Heapsort | Reversed Sorted | 100000 | 0.4083287716 |
| Heapsort | Random | 1000000 | 7.102103949 |

| | | | |
|---|---|---|---|
| Heapsort | Sorted | 1000000 | 5.439090014 |
| Heapsort | Reversed Sorted | 1000000 | 5.080187798 |
| Mergesort | Random | 100000 | 0.4185872078 |
| Mergesort | Sorted | 100000 | 0.6231200695 |
| Mergesort | Reversed Sorted | 100000 | 0.6346271038 |
| Mergesort | Random | 1000000 | 3.489235878 |
| Mergesort | Sorted | 1000000 | 2.421803951 |
| Mergesort | Reversed Sorted | 1000000 | 3.692567825 |
| Quicksort | Random | 100000 | 0.4240620136 |
| Quicksort | Sorted | 100000 | 0.472935915 |
| Quicksort | Reversed Sorted | 100000 | 0.6076500416 |
| Quicksort | Random | 1000000 | 4.039109945 |
| Quicksort | Sorted | 1000000 | 2.376554012 |
| Quicksort | Reversed Sorted | 1000000 | 2.439034224 |

This data allows us to compare how these algorithms compare when they're executed on different sizes of data and different types of data.

We can note a few trends since there is a lot of data that we can compare:

- Heapsort tends to be insensitive to the type of data being inputted and performs consistently across types of data
- Heapsort tends to perform much worse than quicksort and mergesort when operating on a large amount of data (1M elements)

Theoretically, we expect heapsort to perform similarly to both mergesort and quicksort in terms of the worst case O(n log n), however there are caveats. This heavily depends on the type of data that is being operated on (random, sorted, reverse sorted) and the size of the data input. This is shown in the raw data in the table above and the trends I have described. We note that heapsort tends to perform worse as the input data size increases. For example, we can see that heapsort on 1M random data elements (~7.1s) performs noticeably worse than both mergesort (3.49s) and quicksort (4.04s). Furthermore, we note that heapsort performance is not sensitive to the input data type. We see that both on 100K and 1M elements sorted, heapsort performs relatively the same across random, sorted, and reverse sorted data!

# Part 2: Priority Queue Implementation & Analysis

**Priority Queue Implementation**

The priority queue task simulation implementation is in the github repo linked here: https://github.com/rlahoti-cb/MSCS532_Assignment4. The README.md covers how to run this code to generate the data that is used in the analysis section below. Note, that your numbers may vary slightly due to running the code on a different machine and other variables.

**Design Choices**

There are a few design choices that we can highlight in my task scheduler priority queue implementation.

The first choice was the underlying data structure that we can use to implement a binary heap. In my implementation, I chose to use a python list. One reason why python lists are efficient in this implementation is that we are able to find the parent node index ( (i-1) // 2), left subtree index (2 * i + 1), right subtree index (2 * i + 2) easily. This allows us to quickly traverse the tree structure using simple arithmetic operations instead of having to use pointers as references. Another reason why the python list is efficient is that we are able to use index-based operations on the python list to execute the priority queue's operations (e.g. insert). Again, this allows us to efficiently operate on our priority queue.

One other design choice that I made was that I created a max heap approach to the priority queue. In other words, I implemented the priority queue such that higher priority tasks are higher up in the heap structure. The highest priority task at any given point will be the root node of the heap structure.

**Implementation Details**

In terms of implementation, I chose to first define a Task class which represents the task object. This allows us to store tasks easily without having to implement a more complex approach. I then defined a TaskHandler class which is essentially the priority queue implementation to simulate the handling of tasks with priorities. I chose to use an object-oriented approach since it makes the code structure easier to follow.

**Scheduling Results Analysis**

In this section, we can discuss the results of our time complexity analysis of the priority queue's various operations. These operations include insert time (time taken to insert a task into the heap), extraction time (time taken to extract a task from the heap), and increase key time (time taken to increase the priority of a task to a higher priority).

For this experiment, I ran two iterations. The first iteration was with a heap structure with 100K elements (tasks) and the second iteration was with a heap structure with 1M elements (tasks). In the experiment, each operation was ran on all elements of the heap. For example, I measured the time taken to insert 100K tasks, extract 100K tasks , and increase the priority of 100K tasks.

| Heap Size | Insert Time (s) | Extraction Time(s) | Increase Key Time (s) |
|---|---|---|---|
| 100000 | 0.03942203522 | 0.4888350964 | 0.1286847591 |

| | | | |
|---|---|---|---|
| 1000000 | 0.4066250324 | 5.804657936 | 1.136415958 |

This second table shows how much longer each operation took on 1M elements than 100K elements.

| | Insert | Extract | Increase Key |
|---|---|---|---|
| **Factor of time increase from 100K -> 1M** | 10.31466362 | 11.87447051 | 8.831006609 |

Now that we have described the experiment, we can analyze the results. We can see that the time taken for each operation takes roughly ~10 times longer for the 1M task case than the 100K task case. This aligns with the fact that with 1M elements, we have 10x more data to operate on. These empirical results imply that the time complexity is a function of the input size – so as the data scales in size so does the time taken.

# References


Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). Random House Publishing Services. https://reader2.yuzu.com/books/9780262367509

GeeksforGeeks. (2025, May). Heap sort. https://www.geeksforgeeks.org/dsa/heap-sort/

GeeksforGeeks. (2025, March 12). What is Priority Queue | Introduction to Priority Queue. https://www.geeksforgeeks.org/priority-queue-set-1-introduction/

Programiz. (n.d.). Priority Queue Data Structure. In Data Structures and Algorithms. https://www.programiz.com/dsa/priority-queue