**Name: AJAL RC**

**Student ID: 005039456**

**Algorithms and Data Structures (MSCS-532-B01)**

**Assignment 4: Heap Data Structure: Implementation, Analysis, and Applications**

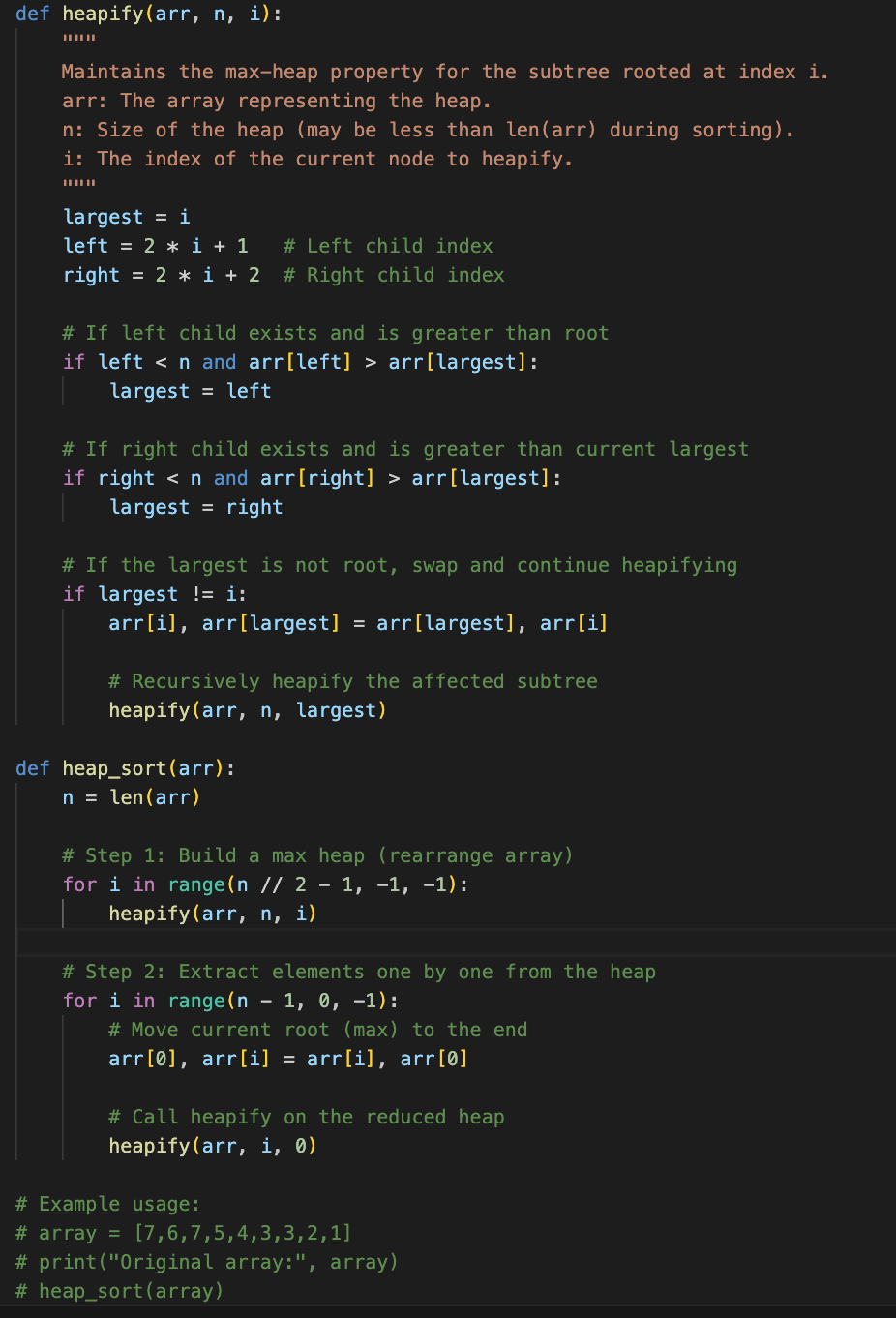
**Introduction**

Heap data structures are a foundational concept in computer science, widely used to solve problems requiring efficient access to the largest or smallest element. The binary heap, in particular, is critical for sorting (Heapsort). It is also highly used in scheduling and implementing priority queues. This assignment has a detailed Python implementation of Heapsort and a heap-based priority queue. The report also provides rigorous time and space analysis and empirically compares Heapsort to other classic sorting algorithms, demonstrating the use of heaps in real-world task scheduling.

**Part 1: Heapsort Implementation along with Analysis**

**1.1 Heapsort Implementation**

Below is an extensively commented Python implementation of the Heapsort algorithm. This approach uses an array (Python list) to represent a binary max-heap. Heapsort works in two main phases: first building a max-heap from the unsorted array, then repeatedly extracting the largest element and moving it to the end of the array.



**1.2 Heapsort Time and Space Complexity Analysis**

**Time Complexity:**

* **Building the max-heap:** This takes O(n) time. Although it appears as O(n log n), the bottom levels of the tree require much less work, resulting in a linear-time build.
* **Extraction elements from Heap:** Each extraction (n times) involves swapping and heapifying, each taking O(log n) time due to the tree height.
* **Overall:** O(n) (heap build) + O(n log n) (extraction and heapify) = **O(n log n)**
* **Best, Average, and Worst Case:** All cases are O(n log n) because the heap operations do not depend on the initial order of the data. No matter how the array is arranged, the exact number of operations is always performed regardless.

**Why O(n log n) always?**  
Unlike some other sorting algorithms, the time taken does not change whether the array is already sorted, reverse sorted, or random. This is because the heap operations always take the same amount of work for each element.

**Space Complexity:**

* **O(1) extra space: Heap sorts the array in place, meaning** that apart from a few index variables and the recursion stack for heapify, no additional memory is used, making it very memory efficient.
* **Extra space required for recursion in the heapify process, which is at most O(log n):** Each call to heapify may go down a path of the tree whose height is log n, so you never use more than that amount of stack space.
* **Stability:** Stability means that if two elements have equal value, their order in the original array is preserved in the sorted result. Heapsort may rearrange equal elements, so it is not suitable if stability is required.

**1.3 Comparison: Heapsort, Quicksort, and Merge Sort (Benchmark)**

**Benchmarking Code**

**A screen shot of a computer program

AI-generated content may be incorrect.  
  
Benchmark result:**

A screenshot of a computer screen

AI-generated content may be incorrect.  
**Analysis from Benchmark**

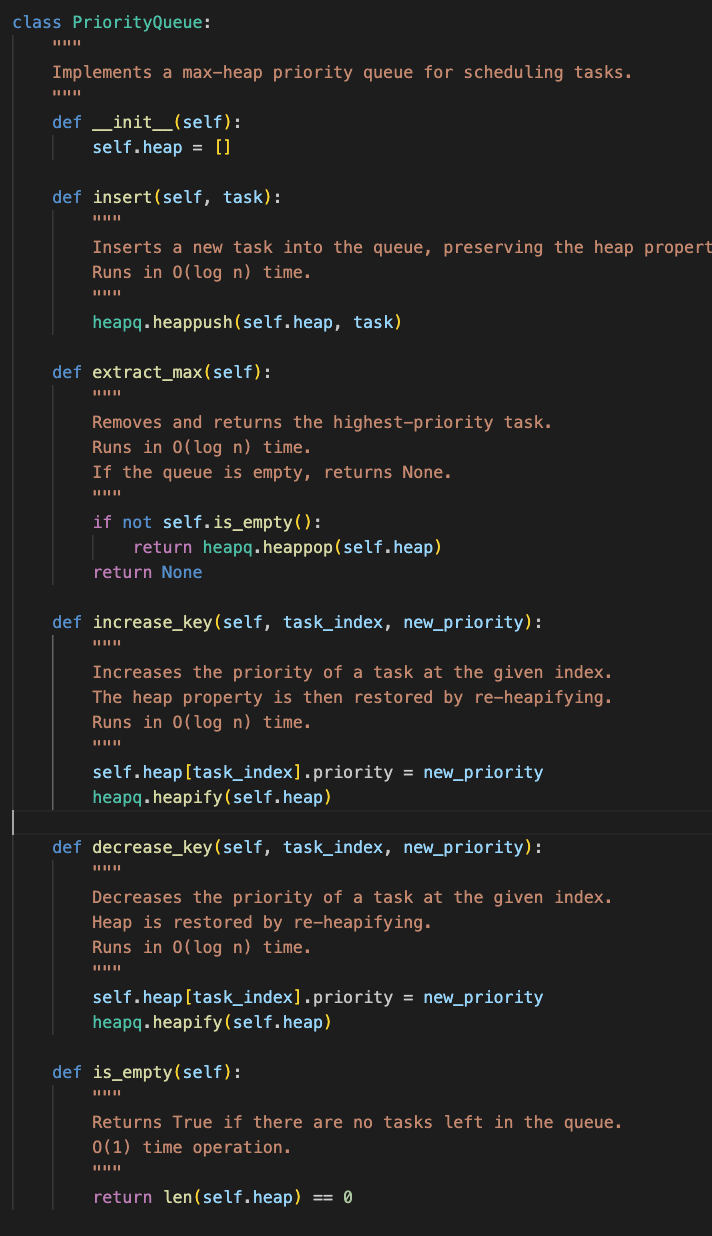
* Heapsort’s running time was consistent across all input types, meaning that even if the array started sorted, reversed, or random, the work done by Heapsort was about the same every time, leading to O(n log n) consistently.
* Mergesort was usually faster than Heapsort for these test sizes.
* Quicksort was also fast and better than Heapsort and Mergesort, but it could slow down for vast arrays or poorly implemented pivots.
* Overall, Heapsort is a reliable and memory-efficient algorithm, but not always the fastest in real life due to more swaps and less cache-friendly access.

**Part 2: Priority Queue Implementation and Application**

**2.1 Data Structure and Design Choice**

* **I chose to represent my binary heap using a Python list because this approach is simple, efficient, and supported by Python’s heapq module.**
* With a list, I can quickly compute parent and child positions and perform all heap operations in place, without extra memory for nodes or pointers.
* **Each task in my scheduling simulation is represented by a Task class, which stores the task’s unique ID, its priority, its arrival time, and its deadline.**
* **I use a max-heap because I want to always schedule the task with the highest priority next.** In a max-heap, the largest element is always at the top, so it can be removed quickly. This matches the requirements for a typical scheduler or real-time system.

**2.2 Priority Queue: Code Implementation**

**  
  
2.3 Scheduling simulation**

The following example creates several tasks with different priorities and demonstrates that the scheduler always chooses the task with the highest priority to execute next.

A computer screen shot of numbers and letters

AI-generated content may be incorrect.  
  
A black background with white text

AI-generated content may be incorrect.  
  
When you run this simulation, you will see that the task with the highest priority is always removed from the queue first, demonstrating the effectiveness of the heap-based priority queue for scheduling.

### **2.4 Priority Queue Time Complexity**

* **Inserting a new task takes O(log n):** The task may need to be moved up the tree, but at most it will move up several times equal to the tree's height, which is log n.
* **Extracting the highest-priority takes O(log n) time:** The heap property must be restored after removing the root (the most significant value), which can require moving elements down the tree.
* **Priority Change:** Increasing or decreasing the priority of a task takes O(log n) time, because it might need to move up or down the heap to restore order.
* **Empty Check:** Checking if the priority queue is empty takes O(1) time, since it only checks the length of the heap list.

**GitHub**

All the detailed implementation of the code snipped, including this report is also upload to GitHub for further which is linked at: <https://github.com/ucajalrc/assignment-4>

**Conclusion**

In this assignment, both Heapsort and a heap-based priority queue in Python were implemented and analyzing alongside, explaining each decision in detail. The report also explained how these structures work in both theory and practice. Heapsort proved to be a reliable O(n log n) sorting algorithm that works in-place and does not depend on input order. The heap-based priority queue allows for efficient scheduling of tasks by priority, and the simulation confirmed that the most urgent tasks are always scheduled first.

**References**

Sedgewick, R., & Wayne, K. (2022). Algorithms (4th ed.). Addison-Wesley. <https://algs4.cs.princeton.edu/home/>

Williams, J. W. J. (1964). Algorithm 232: Heapsort. Communications of the ACM, 7(6), 347–348. <https://dl.acm.org/doi/10.1145/365766.365769>

Knuth, D. E. (1998). The art of computer programming, volume 3: Sorting and searching (2nd ed.). Addison-Wesley. <https://archive.org/details/ArtOfComputerProgrammingVol3/page/n1/mode/2up>