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

**Overview:**

This assignment explores heap data structures, focusing on their implementation, time complexity analysis, and real-world applications in sorting (Heapsort) and priority queue operations. By implementing Heapsort and a priority queue using a binary heap, this task aims to deepen the understanding of heap-based algorithms and their practical utility.

**Heapsort Implementation and Analysis**

**1. Implementation:**

The Heapsort algorithm has been implemented using a max-heap structure. The algorithm operates in two main phases:

* **Building the Max-Heap**: The input array is transformed into a max-heap by applying the max\_heapify function to all non-leaf nodes, starting from the last node up to the root. This ensures that the largest element is always at the root of the heap.
* **Sorting**: Once the max-heap is built, the maximum element is repeatedly extracted (using the extract\_max function) and placed at the end of the array (Kobbaey et al., 2022). After each extraction, the heap property is restored by reapplying the max\_heapify function to the heap.

**2. Analysis of Implementation:**

* **Time Complexity**:
  + **Building the Heap**: The max\_heapify operation has a time complexity of O(log n) for each node. Since we only need to apply max\_heapify to non-leaf nodes (n/2 nodes), the overall time complexity for building the heap is O(n).
  + **Heapsort Process**: The extract\_max operation, which extracts the root and restores the heap, has a time complexity of O (log n). Since this operation is repeated n times, the overall time complexity of Heapsort is O (n log n) in all cases—whether the input is sorted, reverse-sorted, or random.
* **Space Complexity**: The space complexity of Heapsort is O (1), as the algorithm sorts the array in place without requiring additional storage.

**3. Comparison:**

When comparing Heapsort with other sorting algorithms such as Quicksort and Merge Sort, empirical testing shows that Heapsort is more stable in terms of time complexity. It consistently performs at O (n log n) across various input sizes and distributions. In contrast, Quicksort can have a worst-case time complexity of O(n²), especially when dealing with sorted or reverse-sorted data, while Merge Sort guarantees O(n log n) but requires additional space for temporary storage.

**Priority Queue Implementation and Applications**

**1. Data Structure**:

To represent the priority queue, I opted for an array-based binary heap. This choice is motivated by the simplicity and efficiency of heap operations—both insertion and extraction of the highest-priority task can be performed in O (log n) time.

**Task Class Design:**

A Task class was created to represent individual tasks, with attributes such as:

* task\_id: A unique identifier for the task.
* priority: The priority value, where higher values represent higher priorities.
* arrival\_time: The time when the task was created.
* deadline: The task’s deadline for completion.

For this assignment, a max-heap is used, meaning that tasks with higher priority values are given precedence (Lafore et al., 2022).

**2. Core Operations:**

* **Insert(task)**: The insert function places a new task at the end of the heap array and maintains the heap property by "bubbling up" the new task if necessary. The time complexity of this operation is O(log n).
* **Extract\_max()**: This operation removes and returns the task with the highest priority (root of the heap). The time complexity is O(log n) due to the need to restore the heap after extraction.
* **Increase\_key(task, new\_priority)**: This operation increases the priority of an existing task and adjusts its position in the heap by "bubbling it up." The time complexity is O(log n), as the task may need to move up the heap to maintain the heap property.
* **Is\_empty()**: This operation simply checks whether the heap is empty by evaluating the length of the heap array. It runs in O(1) time.

**3. Scheduling Results:**

The priority queue’s use case is to manage tasks based on priority, ensuring that the task with the highest priority is always processed first (Kobbaey et al., 2022). This is particularly useful in real-time scheduling systems, where tasks must be handled according to their urgency.

The sample output provided from the code demonstrates the functionality of the priority queue. Initially, the sorted array [1, 3, 4, 5, 10] is produced after applying Heapsort, indicating a correctly functioning sorting process (Kobbaey et al., 2022). Additionally, the highest-priority task, with an initial priority of 2, is successfully extracted from the priority queue. After increasing the priority of the task to 7, the priority queue reflects this change, confirming that the increase\_key operation works as expected.

**Conclusion:**

This assignment provides a comprehensive exploration of heap data structures and their real-world applications. The implementation of Heapsort, followed by a detailed analysis of its time and space complexity, reinforces the algorithm’s efficiency in sorting tasks. The priority queue, built using a max-heap, proves to be a valuable tool for scheduling tasks based on priority. The operations—such as insertion, extraction, and priority adjustment—are optimized to ensure efficient task management. Through this assignment, the theoretical analysis and empirical results illustrate the power of heaps in handling priority-based tasks and sorting efficiently.

**References**

Kobbaey, T., Xanthidis, D., & Bilquise, G. (2024). Data Structures and Algorithms with Python. In *Handbook of Computer Programming with Python* (pp. 207-272). Chapman and Hall/CRC.

Lafore, R., Broder, A., & Canning, J. (2022). *Data Structures & Algorithms in Python*. Addison-Wesley Professional.