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

**1. Heapsort Implementation and Analysis**

**1.1 Implementation of Heapsort**

The Heapsort algorithm was implemented using a **max-heap**. The basic idea is to first build a heap from the given array and then repeatedly extract the maximum element (the root) and place it in its correct position at the end of the array. The core components of the implementation include:

* **heapify(arr, n, i)**: This function maintains the heap property for the subtree rooted at index i by comparing the node with its left and right children, and if necessary, swapping the node with the largest child and recursively heapifying the affected subtree.
* **build\_max\_heap(arr)**: This function constructs the max-heap by calling heapify on each non-leaf node, starting from the last non-leaf node and moving upward to the root.
* **heapsort(arr)**: This is the main sorting function, which first builds the max-heap and then repeatedly extracts the maximum element, adjusting the heap after each extraction.

**1.2 Time Complexity Analysis**

* **Building the Max-Heap**: The build\_max\_heap function runs in **O(n)** time. This is because each heapify operation runs in **O(log n)** time, and there are **n / 2** non-leaf nodes in the array.
* **Extracting the Maximum Element**: During the heapsort process, the algorithm performs **n** extractions, and each extraction takes **O(log n)** time due to the need to maintain the heap property. Therefore, the time complexity for this phase is **O(n log n)**.
* **Overall Time Complexity**: The total time complexity of Heapsort is **O(n log n)** in all cases (best, average, and worst), as the time for heap construction and element extraction is dominated by the **O(n log n)** complexity of the heap operations.
* **Space Complexity**: The space complexity is **O(n)** because the input array is modified in place, and no extra data structures are used besides the array itself.

**1.3 Comparison with Other Sorting Algorithms**

To evaluate the performance of Heapsort, I compared its execution time with **Quicksort** and **Merge Sort** on arrays of different sizes and distributions, including **sorted**, **reverse-sorted**, and **random** arrays.

* **Time Complexity Comparison**:
  + **Quicksort**: Has an average time complexity of **O(n log n)**, but it can degrade to **O(n²)** in the worst case, particularly with poor pivot selection. This happens most often with reverse-sorted arrays.
  + **Merge Sort**: Guarantees **O(n log n)** time complexity in all cases, though it requires additional space for merging subarrays, making its space complexity **O(n)**.
  + **Heapsort**: Unlike Quicksort, Heapsort maintains **O(n log n)** time complexity in all cases. However, it is typically slower in practice than Quicksort, due to more frequent cache misses during the heap operations.
* **Empirical Results**: In practical tests, Heapsort was slower than both Quicksort and Merge Sort for large datasets, particularly on random and reverse-sorted arrays, due to Quicksort’s efficient divide-and-conquer strategy with good pivot selection.

**2. Priority Queue Implementation and Applications**

**2.1 Data Structure**

The priority queue was implemented using a **max-heap**, which ensures efficient extraction of the task with the highest priority. The heap was represented using a Python list, which provides **O(1)** access to the parent and child nodes using index calculations.

A **Task class** was created to represent individual tasks, with attributes such as **task ID**, **priority**, **arrival time**, and **deadline**. The \_\_lt\_\_ method was implemented in the Task class to allow comparison based on priority, making the heap a max-heap (highest priority first).

**2.2 Core Operations**

The following core operations were implemented for the priority queue:

* **Insert (insert(task))**: Adds a new task to the heap, then calls \_heapify\_up to maintain the heap property. This operation runs in **O(log n)** time.
* **Extract Max (extract\_max())**: Removes and returns the task with the highest priority (the root of the heap). The heap is adjusted by calling \_heapify\_down. This operation also runs in **O(log n)** time.
* **Increase Key (increase\_key(task, new\_priority))**: Modifies the priority of an existing task and adjusts its position in the heap using \_heapify\_up. This operation takes **O(log n)** time.
* **is\_empty()**: Checks whether the heap is empty. This is a constant-time operation, **O(1)**.

**2.3 Design and Implementation Justification**

* The decision to use a **max-heap** was based on the assumption that tasks with higher priorities should be extracted first, aligning with most scheduling algorithms (such as CPU scheduling and task scheduling).
* The binary heap is an efficient choice for the priority queue because it guarantees **O(log n)** time complexity for insertion and extraction operations. This is essential for real-time scheduling applications where efficiency is critical.

**2.4 Time Complexity Analysis for Priority Queue Operations**

* **Insert Operation**: Inserting a task involves adding it to the end of the heap and "bubbling it up" to the correct position. This operation takes **O(log n)** time.
* **Extract Max Operation**: Extracting the task with the highest priority involves swapping the root with the last element, removing the last element, and "bubbling down" the new root. This also takes **O(log n)** time.
* **Increase Key Operation**: Changing a task's priority involves adjusting its position in the heap by moving it upward, which takes **O(log n)** time.
* **is\_empty Operation**: Checking if the heap is empty is an **O(1)** operation.

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

* 1. **Implementation of heapsort**  
     Heaps was effectuated to sort arrays, being build from large to small, with insignificant output using the max-heap concept. Primarily, implementers were to construct a heap from the given array and then repeatedly take out the greatest element (the root) and place it in its appropriate position at the ending block of the array for as long as there was an additional element to be taken out.

• heapify(arr, n, i): This function, for the subtree rooted at i, recursively combines nodes toward its leaves and the upper levels of resultant tree, heapifying index i via a comparison with its respective left and right children.

• build\_max\_heap(arr): Calling heapify on each non-leaf node at and above the last leaf-node builds max-heap.

• Heapsort(arr): This is the chief sorting function that builds a max-heap first and then repeats the extraction of the maximum element, which alters the heap after each extraction.

**1.2 Time Complexity**

• Building Max-Heap: The time taken by build\_max\_heap is O(n) since every heapify operation takes O(log n) time, and there are n / 2 non-leaf nodes throughout the array.

• Extracting Maximum Element: n extractions are performed during the operation of heapsort, and each one of them takes O(log n) time in order to maintain the heap properties. So, the time complexity for that specific phase is O(n log n).

• Going forth: The total running time of Heapsort remains O(n log n) for all types of arrays (best, average, and worst) due to the fact that the rank of the heap is unjustly compared to the extractions, which cost O(n log n) anyway.

• Space Complexity: Since the input array is slightly modified in-place, with no external use of other function calls or associative data types, the space complexity is O(n).  
  
**1.3 Comparison with other sorting algorithms**

In order to compare the performance of Heapsort, I compared its implementation time with Quicksort and Mergesort for sizes of varying arrays within different distributions of sorted, reverse-sorted, and random arrays.

• **Time complexity comparison:**

o Quicksort: runtime is average O(n log n), with the worst runtime of O(n^2) in the worst case due to poor pivot selection, which is often seen when the data is in reverse-sorted condition.

o Mergesort: always runs at O(n log n)

operation time given additional space for merge-operation sub-arrays makes it space expenditure for O(n).

o Heapsort: O(n log n) for all cases in it, unlike Quicksort, with space complexity O(1). The implementation, however, even among the best, seems to be rather slow in terms of the run time compared to Quicksort due to the cache-miss incurred in the order of heap operations.

• Practical Results: Heapsort was slower in analyzing large-size test data than Quicksort and Mergesort, especially for random and reverse arrays, because the divide-and-conquer strategy of Quicksort is effective when a good pivot is favored.  
  
**2. Priority Queue Implementation and Applications**

**2.1 Data Structure**

Max-heap was used to implement a priority queue to facilitate the extraction of a task with the highest priority efficiently. Implementation of the heap was based on a Python list, as it sanctions the parent, child nodes' access in O(1) time; with the help of index calculations.

A Task class was formulated to task different tasks with attributes like task ID, task priority, task arrival time, etc. The \_\_lt\_\_ method was implemented within the Task class to allow the high-priority tasks to be at the top, maintaining a heap as a max-heap, while \_\_gt\_\_ operation, on the other hand, would have created a min-heap.

**2.2 Core Operations**

The following core operations were implemented for the priority queue

• Insert (insert(task)): The new task is put into the heap and thereafter \_heapify\_up that is run so as to maintain the heap property, takes 0(log n) time.

• Extract Max (extract\_max()): This action removes the highest-priority task (root of the tree's heap) and subsequently gathers the heap by using \_heapify\_down. This will only take 0(log n) time.

• Increase Key (increase\_key(task, new\_priority)): This action alters the priority of a displayed task; the priority for the task is accordingly adjusted through \_heapify\_up. It also requires 0(log n) time.

• is\_empty(): Checks if heap is empty; this is O(1) operation.  
  
**2.3 Design and Implementation Justification**

• The reason for preferring use of max-heap can be viewed as that in a schedule of most deterministic algorithms like CPU scheduling and task scheduling, jobs with higher priority should be sent first.

• Binary heaps come as an efficient selection for implementing the priority queue as they warranted O(log n) time complexity for insert and extract-min operations. This becomes necessary for real-time scheduling applications, where efficiency is integral.

**2.4 Time Complexity Analysis for Priority Queue Operations**

• Insert Operation: Inserting a task involves adding an element at the end of the heap and "bubbling it up" to the right place. This operation takes O(log n) time.

• Extract Max Operation: Extracting the maximal-priority task involves swapping the root and last elements, the deletion of the last element, and "bubbling down" the new root. This operation takes O(log n) time.

• Increase Key Operation: Changing a task's priority involves moving it upward with respect to its correct position in the heap, which takes O(log n) time.

• is\_empty Operation: Checking if the heap is empty is an O(1) operation.