**Heap Sort:**

Analysis:

Heapsort is a comparison-based sorting algorithm that works by first building a max heap (for sorting in ascending order) from the input data, and then repeatedly extracting the maximum element (the root of the heap) and adjusting the heap accordingly.

Heapsort consists of two main steps:

* Building the Max-Heap: Rearrange the input array into a max-heap.
* Heap Sort: Repeatedly extract the maximum element (root of the heap) and maintain the heap property.

Step 1: Building the Max-Heap

* The heapify function is called for all non-leaf nodes, starting from the last non-leaf node.
* For a node at depth d in the heap, heapify can take at most O(h−d), where h is the height of the heap.
* Summing over all nodes, the total work for building the max-heap is O(n).

Step 2: Heap Sort

* In each iteration of the sorting phase, we:
* Swap the root (maximum element) with the last element in the heap. This operation is O (1).
* Reduce the size of the heap by 1.
* Call heapify on the root to restore the heap property, which takes O(logn) in the worst case.
* This process is repeated n−1 times.
* Thus, the total time complexity for the sorting phase is:

(n−1) ⋅ O (log n) = O(n log n)

* Worst Case: At every step, heapify is called on the root, requiring O (log n). With n elements, this results in O (n log n).
* Average Case: The operations performed remain the same regardless of input, as the algorithm always builds the max-heap and performs the same number of swaps and heapify calls. This results in O (n log n).
* Best Case: Even if the array is already sorted, Heapsort does not take advantage of it. The same heap-building and sorting operations occur, resulting in O (n log n).

Space Complexity and additional Overheads.

* Space complexity analysis of Heapsort involves considering both in-place sorting and auxiliary space usage. Heapsort is an in-place sorting algorithm, meaning it constructs the heap using the input array itself and does not require additional storage for the heap. The only extra memory utilized is for the recursive calls during the heapify process. In the worst case, the recursion depth corresponds to the height of the heap, which is O (log n).
* Therefore, the total space complexity is O (1) if recursive calls are optimized to avoid unnecessary stack space, such as by using an iterative heapify approach.
* In terms of overheads, each recursive call in the heapify process introduces a small overhead to the stack, but this overhead is O (log n). As for array modifications, the algorithm requires frequent swaps, but these are constant-time operations O (1) and do not have a significant impact on the overall complexity.

Comparison:

Heapsort vs Quicksort: Time Complexity by Input Size and Distribution.

A table with numbers and numbers

Description automatically generated

Heap Sort:

* In theory, Heapsort consistently operates in O(nlogn) time, regardless of input distribution. This is reflected in the results, where Heapsort's performance remains relatively stable across sorted, reverse-sorted, and random inputs. For small input sizes (100), Heapsort's time is slightly higher than Quicksort’s, which is consistent with the fact that Heapsort performs the same number of operations regardless of the data distribution. As input size increases (8000, 12000, 30000), Heapsort’s execution time grows more significantly, which is expected due to its O(nlogn) complexity. The lack of optimization for sorted or reverse-sorted data results in a steady increase in time across distributions.

Quick Sort:

* Quicksort, theoretically, has an average-case time complexity of O(nlogn), but its practical performance often exceeds Heapsort’s due to more efficient partitioning and lower constant factors. In the observed results, Quicksort consistently outperforms Heapsort, especially for larger input sizes. For sorted and reverse-sorted distributions, Quicksort's performance is notably better, reflecting the algorithm’s ability to exploit these patterns effectively. The random distribution still sees Quicksort Outperforming Heapsort, indicating that Quicksort’s efficiency is not just input-dependent but also benefits from its partitioning strategy and pivot selection. As input size increases, the difference in performance becomes more pronounced, confirming Quicksort’s superiority for larger datasets.

Both the theoretical analysis and the observed results confirm that while both algorithms share the same worst-case time complexity, Quicksort performs better in practice due to its adaptive partitioning strategy and lower constant factors, especially for larger datasets. Heapsort, on the other hand, shows predictable but slower performance across all input distributions due to its rigid O(nlogn) structure.

**Part A: Priority Queue Implementation:**

**Using an Array for the Binary Heap:**

Efficiency: A binary heap can be efficiently implemented using an array because the parent-child relationships can be derived using simple index arithmetic:

* Parent index: (i−1)//2
* Left child index: 1
* 2i+1
* Right child index: 2i+2

**Task Class**: The Task class is designed to represent a task with the following attributes:

* task\_id: A unique identifier for the task.
* priority: The priority of the task, used to determine its execution order.
* arrival\_time: The time when the task is available for execution.
* deadline: The time by which the task should be completed

**MaxHeap Data Structure:** A custom MaxHeap class is implemented to maintain a max-heap structure, where the task with the highest priority is always at the root. This is fundamental for scheduling, as tasks with higher priority need to be executed before lower-priority tasks.

Implementation Details:

* Insertion (insert): Inserting a task into the heap requires appending the task to the end of the list and then using \_heapify\_up to maintain the max-heap property. This ensures that after insertion, the highest-priority task is at the root.
* Extraction (extract\_max): Extracting the task with the highest priority involves swapping the root task with the last task in the heap, removing the last task (which was previously the root), and reheapifying the heap using \_heapify\_down to restore the heap property.
* Priority Modifications: Both increase\_key and decrease\_key methods allow modification of a task's priority. When the priority is increased, the task is moved upwards in the heap using \_heapify\_up, and when the priority is decreased, the task is moved downwards using \_heapify\_down.

Time Complexity Analysis:

Insertion (insert):

* The task is added at the end of the heap, and then \_heapify\_up is invoked to maintain the heap property.
* In the worst case, the task may need to be compared with every other task in the heap, resulting in a time complexity of O (log n), where n is the number of tasks in the heap.

Extraction (extract\_max):

* The root task is swapped with the last task, and \_heapify\_down is called to restore the max-heap property.
* In the worst case, the task may need to be compared and swapped with every other task in the heap, leading to a time complexity of O (log n).

Priority Modification (increase\_key and decrease\_key):

* Both operations require searching through the heap for the task with the specified task\_id. This search is O(n) in the worst case.
* After locating the task, the heap is restructured using either \_heapify\_up or \_heapify\_down, both of which have a time complexity of O (log n).
* Therefore, the total time complexity for priority modification is O (n + log n) = O(n).