Heap Data Structures: Implementation, Analysis, and Applications

# 1. Heapsort Implementation

Heapsort is implemented using a max-heap data structure. The heap is constructed in-place using the array representation, where the largest value is at the root. The algorithm follows two main steps: building the heap, and repeatedly extracting the maximum value and maintaining the heap property.

# 2. Heapsort Analysis

Time Complexity:  
- Worst Case: O(n log n)  
- Average Case: O(n log n)  
- Best Case: O(n log n)  
  
The reason Heapsort is O(n log n) in all cases is that building the heap takes O(n), and each of the n extractions takes O(log n) time.  
  
Space Complexity:  
- Heapsort is an in-place sorting algorithm with a space complexity of O(1). It does not require additional space for sorting.

# 3. Comparison with Quicksort and Merge Sort

To compare the performance of Heapsort, Quicksort, and Merge Sort, we tested each algorithm on arrays of sizes 1000, 5000, and 10000 with random, sorted, and reverse-sorted data. Empirically, Quicksort tends to be the fastest on average due to cache efficiency, while Heapsort remains consistent. Merge Sort performs well and guarantees O(n log n), but uses O(n) extra space.  
  
Observed results aligned with theoretical expectations: Heapsort is consistent, Quicksort can degrade to O(n^2) if not randomized, and Merge Sort uses more memory but performs steadily.

# 4. Priority Queue Implementation

A priority queue is implemented using Python’s built-in heapq module (which implements a min-heap). The priority queue manages tasks with attributes such as task ID, priority, arrival time, and deadline.  
  
We implemented the following operations:  
- insert(task): O(log n)  
- extract\_min(): O(log n)  
- decrease\_priority(task\_id, new\_priority): O(n) (linear scan then heapify)  
- is\_empty(): O(1)  
  
The Task class includes comparison logic for ordering tasks by priority. This structure simulates a scheduler system for task management.

# 5. Design Choices

We chose arrays/lists as the base data structure for both Heapsort and the priority queue for simplicity and performance. For the heap operations, using the array representation aligns with the standard approach and avoids pointer overhead. The decision to use a min-heap for the priority queue reflects typical use cases like task scheduling where lower values imply higher priority.  
  
The use of the heapq module provides reliable and efficient heap operations, while a custom class structure allows us to encapsulate task attributes cleanly.

# 6. Conclusion

This assignment demonstrates the power and efficiency of heap-based data structures in both sorting and priority-based scheduling scenarios. Heapsort, while not always the fastest in practice, guarantees O(n log n) time and performs consistently. The priority queue allows real-time task handling with predictable complexity, making it highly applicable to operating systems, simulations, and time-sensitive applications.