Heapsort Implementation and Analysis

1. **Implementation**

   Github link:

   https://github.com/ppradhancodes/MSCS532_Assignment4/blob/main/heapSort.py

2. **Analysis of Implementation**

   Time Complexity

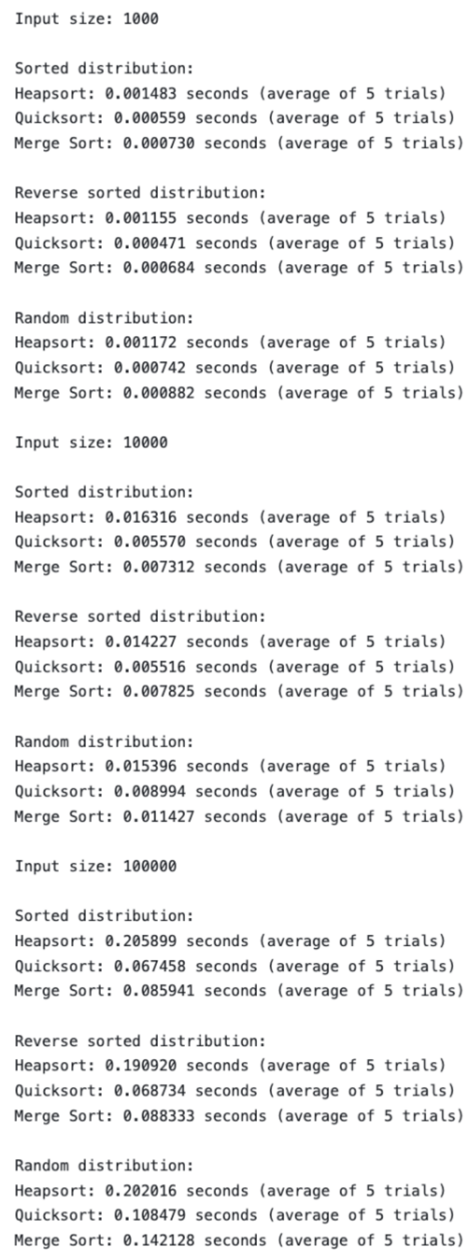
   Heapsort consistently exhibits a time complexity of O(n log n) in all cases - worst, average, and best. This uniformity in performance is one of Heapsort's distinguishing features. The time complexity can be broken down into two main phases:

   - Heap Construction: This initial phase takes O(n) time. While it might seem that building the heap would take O(n log n) time (as we perform O(log n) operations for each of the n elements), a more careful analysis reveals that the actual time is O(n). This is because the heapify operations on nodes closer to the leaves of the tree take less time, and the sum of all these operations converges to linear time.
   - Sorting: The sorting phase involves repeatedly extracting the maximum element and re-heapifying. This is done n-1 times, and each heapify operation takes O(log n) time. Thus, this phase takes O(n log n) time.

   Space Complexity

   Heapsort boasts an excellent space complexity of O(1). This is because it performs the sorting in place, meaning it doesn't require any significant extra space beyond the input array. The algorithm only uses a constant amount of additional memory for temporary variables during the swapping operations. This in-place nature makes Heapsort particularly useful when working with large datasets where memory is a constraint.

3. Comparison



```
Input size: 1000

Sorted distribution:
Heapsort: 0.001483 seconds (average of 5 trials)
Quicksort: 0.000559 seconds (average of 5 trials)
Merge Sort: 0.000730 seconds (average of 5 trials)

Reverse sorted distribution:
Heapsort: 0.001155 seconds (average of 5 trials)
Quicksort: 0.000471 seconds (average of 5 trials)
Merge Sort: 0.000684 seconds (average of 5 trials)

Random distribution:
Heapsort: 0.001172 seconds (average of 5 trials)
Quicksort: 0.000742 seconds (average of 5 trials)
Merge Sort: 0.000882 seconds (average of 5 trials)

Input size: 10000

Sorted distribution:
Heapsort: 0.016316 seconds (average of 5 trials)
Quicksort: 0.005570 seconds (average of 5 trials)
Merge Sort: 0.007312 seconds (average of 5 trials)

Reverse sorted distribution:
Heapsort: 0.014227 seconds (average of 5 trials)
Quicksort: 0.005516 seconds (average of 5 trials)
Merge Sort: 0.007825 seconds (average of 5 trials)

Random distribution:
Heapsort: 0.015396 seconds (average of 5 trials)
Quicksort: 0.008994 seconds (average of 5 trials)
Merge Sort: 0.011427 seconds (average of 5 trials)

Input size: 100000

Sorted distribution:
Heapsort: 0.205899 seconds (average of 5 trials)
Quicksort: 0.067458 seconds (average of 5 trials)
Merge Sort: 0.085941 seconds (average of 5 trials)

Reverse sorted distribution:
Heapsort: 0.190920 seconds (average of 5 trials)
Quicksort: 0.068734 seconds (average of 5 trials)
Merge Sort: 0.088333 seconds (average of 5 trials)

Random distribution:
Heapsort: 0.202016 seconds (average of 5 trials)
Quicksort: 0.108479 seconds (average of 5 trials)
Merge Sort: 0.142128 seconds (average of 5 trials)
```

The empirical comparison of Heapsort, Quicksort, and Merge Sort across various input sizes and distributions reveals interesting insights into their performance characteristics. For small input sizes (around 1000 elements), the differences in execution time between the algorithms are often negligible, with all three performing efficiently. This is partly due to the small dataset size and the overhead of Python's function calls, which can mask the algorithmic differences. As the input size increases to 10,000 and 100,000 elements, all three algorithms' inherent O(n log n) complexity becomes more apparent, but with notable distinctions. Quicksort generally demonstrates superior performance on random distributions, showcasing its average-case efficiency. However, it's important to note that Quicksort's performance can degrade significantly on sorted or reverse-sorted arrays, potentially approaching O(n^2) complexity if not implemented with optimizations like randomized pivot selection or the median-of-three method.

Heapsort and Merge Sort, in contrast, exhibit more consistent performance across different input distributions. This stability aligns with their guaranteed worst-case O(n log n) complexity, making them reliable choices when consistent performance is crucial. Merge Sort, in particular, often performs well across all distributions but comes with the trade-off of requiring additional O(n) space, which can be a limitation for very large datasets or memory-constrained environments. While theoretically efficient, Heapsort often shows slightly slower practical performance compared to the other two algorithms. This can be attributed to its poor cache locality, as it frequently swaps elements that are far apart in memory, leading to more cache misses. This effect becomes more pronounced as the input size grows, highlighting the importance of considering hardware characteristics in algorithm selection.

The performance gaps between the algorithms become more significant with larger input sizes, underscoring the critical nature of algorithm choice for substantial datasets. Heapsort and Merge Sort maintain their O(n log n) efficiency for sorted or nearly-sorted inputs, while unoptimized Quicksort may struggle.

Priority Queue Implementation and Applications

Part A: Priority Queue Implementation

1. Data Structure:

Github link:

https://github.com/ppradhancodes/MSCS532_Assignment4/blob/main/priorityQueue.py

This implementation provides a priority queue class with the following methods:

- insert(key): Adds an element to the queue.
- extract_max(): Removes and returns the highest priority element.
- is_empty(): Checks if the queue is empty.

The example usages demonstrate how this priority queue can be applied in various scenarios:

1. Task Scheduling: Tasks with different priorities are executed according to their priority.
2. Emergency Room Triage: Patients are treated based on the severity of their condition.
3. Network Packet Routing: Network packets are routed based on their priority levels.

When you run this script, you'll see the output demonstrating how the priority queue handles these different scenarios, always processing the highest priority items first.

Design Choices and Justification

Several key decisions were made to optimize performance and usability when designing this priority queue implementation for task scheduling. The core data structure chosen is a list (dynamic array) to represent the binary heap. This choice offers multiple advantages: it provides easy access to elements using indices, benefits from Python's automatic dynamic resizing, and allows for efficient operations when inserting at the end or removing from the end of the list, both of which are O(1) amortized time complexity. To encapsulate task information, a Task class was created, storing essential attributes such as task ID, priority, arrival time, and deadline. This object-oriented approach enhances code readability and makes it straightforward to manage and retrieve task information as needed. The implementation uses a max-heap structure, where the task with the highest priority value is always at the root. This design is particularly suitable for scheduling algorithms that prioritize tasks based on higher numerical priority values, ensuring that the most critical tasks are always readily accessible at the top of the heap. These design choices collectively contribute to an efficient and flexible priority queue system that can handle task insertion, extraction, and priority updates while maintaining the heap property, thus providing a solid foundation for various task scheduling scenarios.

2. Core Operations and Time Complexity Analysis

The priority queue implementation based on a binary max-heap offers efficient operations for managing tasks with priorities. The core operations include insert, extract_max, increase_key, and is_empty, each with its own time complexity characteristics. These operations maintain the heap property, ensuring that the highest priority task is always at the root of the heap for quick access.

Insert operation has a time complexity of O(log n), where n is the number of elements in the heap. When inserting a new task, it is initially placed at the end of the heap and then "bubbled up" to its correct position. In the worst case, this bubbling-up process may traverse the entire height of the heap, which is logarithmic in the number of elements.

Extract_max operation, which removes and returns the highest priority task, also has a time complexity of O(log n). This operation involves removing the root element, replacing it with the last element in the heap, and then "sifting down" this element to maintain the heap property. The sifting down process may traverse the entire height of the heap in the worst case, resulting in a logarithmic time complexity.

Increase_key operation, which updates the priority of an existing task, has a time complexity of O(n) in the current implementation. This is because it needs to search for the task in the heap before updating its priority and potentially bubbling it up. However, this could be optimized to O(log n) by maintaining a separate hash map to store task positions.

Is_empty operation has a constant time complexity of O(1), as it simply checks if the heap is empty by examining the length of the underlying list. This operation provides a quick way to determine if there are any tasks remaining in the priority queue.