**Heapsort Implementation & Analysis**

**Implementation**

<https://github.com/nhemani33090/MSCS532_Assignment4/blob/main/heapsort_comparision.py>

**Analysis of Implementation**

Heapsort has a consistent time complexity of O(n log n) across all cases(worst, average, and best). The process consists of two phases:

1. Heap construction takes O(n) time. This is because heapsort is applied from the last non-leaf node, adjusting the heap for each element.
2. Extraction takes O(n log n) time as it involves repeatedly removing the root (largest element) and re-adjusting the heap.

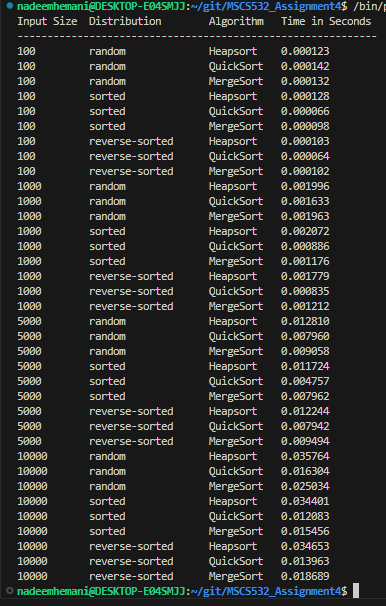
Heapsort’s time complexity is O(n log n) for all cases as the heap construction and extraction process do not optimize for specific input distributions.

**Space Complexity**

* Heapsort is an in-place sorting algorithm, using constant extra space O(1).
* The only additional memory usage comes from the recursion stack used in heapsort, but this is minimal O(log n).
* Therefore, Heapsort is highly memory-efficient compared to other algorithms that require additional space for temporary arrays or merging.

**Empirical Comparison of Heapsort, Quicksort, and MergeSort**

Based on the test results, here’s how Heapsort compares to Quicksort and MergeSort in terms of runtime:



1. **Quicksort:**

* **Fastest for All Cases**: Quicksort performed the fastest across all data types (random, sorted, and reverse-sorted). Its O(n log n) average-case time complexity is reflected in the results, where Quicksort outperformed both Heapsort and MergeSort. Since the pivot selection was efficient, we avoided the worst-case O(n²) performance.
* **Consistent Performance**: Even on reverse-sorted data, Quicksort performed faster than MergeSort and Heapsort, suggesting that the pivot selection in this implementation was efficient, avoiding the worst-case O(n²) performance.

1. **MergeSort:**

* **Consistent, but Slower than Quicksort**: MergeSort consistently performed with O(n log n) time complexity, regardless of input type, which aligns with its theoretical complexity. However, MergeSort was slower than Quicksort for random and sorted data due to the extra memory and merging steps involved.
* **Stable Performance**: It performed similarly on sorted and reverse-sorted data, reflecting the stability of MergeSort in terms of time complexity.

1. **Heapsort:**

* Stable Performance, but Slower: Heapsort showed consistent O(n log n) performance across all input types, which matches the theoretical analysis. However, it was slower than both Quicksort and MergeSort in most cases. This is because the heap structure requires more operations to maintain, making it slower despite having the same overall time complexity.
* Memory Efficiency: Heapsort's primary advantage is its O(1) space complexity, making it ideal for scenarios where memory is limited.

**Conclusion**

* Quicksort was the fastest in our test cases, with better performance for random, sorted, and reverse-sorted data but if the pivot always ends up being the smallest or largest element, the array will not be divided evenly, and Quicksort will have to do more work, leading to O(n²) time complexity in the worst case so it is usually avoided.
* MergeSort provided stable and predictable performance but was slower than Quicksort in practice.
* Heapsort performed reliably with O(n log n) time complexity, but it was generally slower than Quicksort and MergeSort due to additional overhead in maintaining the heap structure but is extremely reliable as the runtime is always O(n log n).

**Priority Queue Implementation**

<https://github.com/nhemani33090/MSCS532_Assignment4/blob/main/priority_queue.py>

A computer screen shot of a program

Description automatically generated

**Data Structure**

**Choice of Data Structure (Array or List)**

Array is the better choice to represent a binary heap because:

* **Efficient Indexing:** Binary heaps are implemented using arrays because accessing left and right children of a node is easy using array indices.
* **Memory Efficiency:** Arrays allow efficient, contiguous memory allocation.
* **Simpler Implementation:** The insert, delete, and heapify operations are simpler with an array-based binary heap.

**Task Class (for Task Representation):**

The Task class stores relevant information:

* **identifier** Unique identifier for each task.
* **priority\_level:** Priority of the task, determining its order in the queue.
* **arrival\_time:** Time when the task arrives.
* **due\_time:** The deadline by which the task needs to be completed.

**Max-Heap vs. Min-Heap:**

We use a max-heap to ensure that the task with the highest priority is always at the root. This makes it easy to extract the highest-priority task efficiently.

**Explanation and Time Complexity Analysis**

**Insert (task):**

* **Logic:** Adds a task to the heap, maintaining the heap property by "bubbling up" the task.
* **Time Complexity:** O(log n), because in the worst case, the task may need to move up through the heap from the last position to the root, which takes log n steps.

**Extract Max (or Min):**

* **Logic:** Removes and returns the task with the highest priority (the root of the heap). The root is replaced with the last element, and the heap property is restored by "bubbling down" the new root.
* **Time Complexity:** O(log n), because in the worst case, the root element may need to move down the heap to a leaf node, which takes log n steps.

**Increase/Decrease priority (task\_identifier, updated\_priority):**

* **Logic:** Changes the priority of an existing task. If the priority increases, the task is moved up; if it decreases, the task is moved down to maintain the heap property.
* **Time Complexity:** O(log n), as the task may need to move up or down through the heap, which takes log n steps.

**is\_heap\_empty():**

* **Logic:** Simply checks if the heap is empty by checking its length.
* **Time Complexity:** O(1), as checking the length of the heap is a constant-time operation.

**Summary of Core Operations:**

* **Insert (task):** O(log n) time complexity.
* **Extract Max (or Min):** O(log n) time complexity.
* **Increase/Decrease Key:** O(log n) time complexity.
* **is\_heap\_empty():** O(1) time complexity.