GIT HUB Link: https://github.com/vtadigotla35101/MSCS532_Assignment4

Heapsort Analysis: Heapsort is an efficient, comparison-based sorting algorithm with a time complexity of **O(n log n)** on all cases like Worst, Average, and Best cases. This consistency is considered and treated as a major advantage.

The algorithm proceeds in two main phases:

1. **Building a Max-Heap:** The build_max_heap function takes an unsorted array and transforms it into a max-heap. This process ensures that the largest element is at the root. While it involves iterating through about half the array and calling heapify on each non-leaf node, a more precise analysis shows this phase is actually O(n) time. Each heapify operation on a node takes time proportional to the height of its subtree, which is O(log n).

2. **Extracting Elements:** After building the heap, the algorithm repeatedly extracts the maximum element (which is always at the root, index 0) and places it at the end of the array.

    - The largest element (arr[0]) is swapped with the last element of the current heap.
    - The effective size of the heap is reduced by one.
    - The new root (the element just swapped into arr[0]) is then "heapified" down to restore the max-heap property for the remaining elements. This heapify operation always takes O(log n) time because the height of the heap is log n.
    - Since this extraction and heapify process occurs n-1 times, this phase contributes O(n log n) to the total time complexity.

Combining both phases, the total time complexity is O(n) + O(n log n) = O(n log n). Heapsort achieves this bound consistently because its operations (heapify) are always bounded by the logarithmic height of the heap structure, regardless of the input data's initial arrangement.

Space Complexity and Overheads:

Heapsort is an in-place sorting algorithm, meaning it sorts the elements within the original array without requiring significant additional memory. Its space complexity is O(1) auxiliary space, excluding the input array itself and considering the recursion stack for heapify to be part of the algorithm's constant overhead in practice, or more precisely O(logn) for the recursive call stack.

However, Heapsort often has larger constant factors compared to algorithms like Quick Sort. This is due to more comparisons and swaps per element in heapify and potentially poorer cache performance. The scattered memory accesses during heap operations can lead to more cache misses on modern processors, making it sometimes slower than Quick Sort's average case in practice, despite both being O(nlogn).

Empirical Comparison Analysis

This analysis is based on the performance metrics generated by running Heapsort, Merge Sort, and Quick Sort on various datasets (random, sorted, reverse-sorted) and increasing input sizes, as implemented in the provided Python Canvas.

Observed Results and Relation to Theoretical Analysis:

1. **Heapsort (Theoretical: O(n log n) Time, O(1) Space across all cases):**
    - **Observed Time:** Empirically, Heapsort tends to be slower than Quick Sort (on average) and sometimes even Merge Sort. Its performance is relatively consistent across random, sorted, and reverse-sorted data. This consistency aligns with its theoretical O(n log n) complexity in all scenarios.
    - **Observed Space:** Heapsort consistently demonstrates very low memory usage, typically the lowest among the three, confirming its theoretical O(1) auxiliary space complexity. This is because it sorts in-place.
    - **Relation:** The slight slowdown compared to Quick Sort and Merge Sort (for random data) is attributable to larger constant factors and poorer cache locality. Heap operations often involve non-contiguous memory access patterns, leading to more cache misses on modern CPU architectures.
2. **Merge Sort (Theoretical: O(n log n) Time, O(n) Space across all cases):**
    - **Observed Time:** Merge Sort's running time remains consistently O(n log n) regardless of whether the data is random, sorted, or reverse-sorted. It generally performs well.
    - **Observed Space:** Merge Sort consistently shows higher memory usage compared to Heapsort and Quick Sort (especially for larger datasets). This is a direct consequence of its requirement for O(n) auxiliary space for the merging process.
    - **Relation:** The empirical results align very well with its theoretical guarantees for both time and space. The extra memory overhead can become a bottleneck for very large datasets or in memory-constrained environments, even if its time complexity is optimal.
3. **Quick Sort:**
    - **Observed Time (Random Data):** Quick Sort often emerges as the fastest algorithm on random data, especially for larger input sizes. This reflects its excellent average-case O(n log n) performance, often benefiting from better cache utilization than Merge Sort and Heapsort.
    - **Observed Time (Sorted/Reverse-Sorted Data):** With the simple "last element as pivot" strategy used in this implementation, Quick Sort's performance degrades significantly on already sorted or reverse-sorted data. You will observe that its running time approaches O(n^2) in these cases, confirming its theoretical worst-case behavior.
    - **Observed Space:** Its memory usage is typically low (O(log n) on average for the recursion stack), making it competitive with Heapsort for space on average cases. However, in worst-case scenarios (sorted/reverse-sorted data), the deep recursion can push its space usage towards O(n) due to the stack depth.

- **Relation:** The empirical results for Quick Sort clearly demonstrate the impact of input distribution on its performance, validating its theoretical best, average, and worst-case complexities. Its speed on random data and degradation on ordered data are direct consequences of its pivot selection strategy.

Discrepancies between Theoretical Analysis and Practical Performance:

- **Quick Sort's Practical Speed:** While all three are asymptotically O(n log n) on average, Quick Sort often wins in practice on random data due to lower constant factors and superior cache efficiency. It involves fewer data movements and more localized operations compared to Merge Sort's auxiliary array copying or Heapsort's scattered accesses.
- **Merge Sort's Memory Overhead:** The O(n) auxiliary space of Merge Sort is its primary practical disadvantage. For very large datasets, this can lead to increased memory allocation/deallocation time and cache misses, making it slower than Quick Sort even with the same asymptotic time complexity.
- **Heapsort's Constant Factors and Cache:** Heapsort's guaranteed O(n log n) time and O(1) space make it theoretically robust. However, in practical benchmarks, it often trails behind Merge Sort and Quick Sort. This is mainly due to its higher constant factors (more comparisons and swaps per element to maintain heap property) and less favorable cache access patterns.

In conclusion, the empirical comparison largely validates the theoretical analyses of Heapsort, Merge Sort, and Quick Sort. Practical considerations like constant factors, cache performance, memory usage, and input distribution play significant roles in determining which algorithm performs "best" in a given real-world scenario. Heapsort offers strong guarantees but can be slower due to overheads; Merge Sort is stable and consistent but memory-intensive; and Quick Sort is often fastest on average but vulnerable to worst-case inputs without proper pivot strategies.

****

Priority Queue Analysis:

1. Insert:

- **Process**:
    - The new task is added at the last position of the heap (O(1)).
    - Then, _heapify_up is called to maintain the heap property.
    - _heapify_up compares the new task with its parent and swaps if necessary, continuing up the tree.
    - Since a binary heap has log(n) levels, the worst-case number of swaps is O(log n).
- Final Complexity: O(log n).

2. Extract Min/Max:

- **Process**:
    - The root element which is highest or lowest priority is removed (O(1)).
    - The last element replaces the root, then the heapifydown is called.
    - heapify-down compares the new root with its children and swaps with the smallest or largest.
    - Since a binary heap has log(n) levels, it may perform up to log(n) swaps.
- Final Complexity: O(log n).

3. Change Priority (change_priority(task, new_priority))

- **Process**:

  a. The task's priority is updated (O(1)).
  b.If the new priority makes it more prominent, it must move up (heapifyup is O(log n)).
  c.If the new priority makes it less prominent, it must move down (heapifydownO(log n)).

- Final Complexity: O(log n).

4. Is Empty (is_empty())

- Process:
    - Simply checks if the heap has elements.
- Final Complexity: O(1).

Summary:

Operation	Time Complexity
Insert	O(log n)
Extract Min/Max	O(log n)
Change Priority	O(log n)
Is Empty	O(1)

Summary of our priority queue implementation and analysis:

1. **Data Structure Choice:**
    - Used a binary heap stored in an array (list) due to easy parent-child indexing and in-place operations.
    - Defined a `Task` class with attributes like task ID, priority, arrival time, and deadline.

   - Choose min-heap for earliest deadlines or max-heap for highest priority first.

2. **Core Operations & Time Complexity:**
   Insert: Adds task at the end and restores heap property and this is O(log n).
   Extract Min/Max: Removes root, replaces it, and restores heap and this is O(log n).
   Change Priority: Adjusts priority, repositions task accordingly and this is O(log n).
   Is Empty: Checks if heap is empty and this is O(1).

3. **Heap Helper Functions:**
   `a.heapifyup` ensures heap validity after an insertion.
   b.`heapifydown` maintains the heap after removing the root. `change_priority` directs the task to move up or down based on priority updates.

   GIT HUB Link: https://github.com/vtadigotla35101/MSCS532_Assignment4

****