Maximum Element in Sliding Window LeetCode

You are given an array of integers nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

Example: [1,3,-1,-3,5,3,6,7], k = 3

Output: [3, 3, 5, 5, 6, 7]

Approach 1: Function to find the maximum element in each sliding window of size 'k' using brute force approach.

- **Functionality:**
    - Finds the maximum element in each sliding window of size 'k' using a brute force approach.
- **Explanation:**
    - Iterates through each sliding window.
    - Finds the maximum element within each window.
    - Stores the maximum element in the result vector.
- **Time Complexity:**
    - **maxSlidingWindowBruteForce**:
        - **Time complexity for each sliding window: O(k).**
        - **Number of sliding windows: O(n - k + 1), where n is the size of the input array.**
        - **Overall Time Complexity: O((n - k + 1) * k).**
- **Space Complexity:**
    - **O(1).**

Approach 2: Function to find the maximum element in each sliding window of size 'k' using deque-based approach.

- **Functionality:**
    - Finds the maximum element in each sliding window of size 'k' using a deque-based approach.
- **Explanation:**

    - Maintains a deque to store indices of maximum elements.
    - Updates the deque by removing out-of-window elements and smaller elements than the current element.
    - Adds the current index to the deque.
    - Once the window size reaches 'k', finds and stores the maximum element.
- **Time Complexity:**
    - **maxSlidingWindow**:
        - **Time complexity for each element: O(1).**
        - **Overall Time Complexity: O(n), where n is the size of the input array.**
- **Space Complexity:**
    - **O(k).**

Approach 3: Function to find the maximum element in each sliding window of size 'k' using max heap approach

- **Functionality:**
    - Finds the maximum element in each sliding window of size 'k' using a max heap approach.
- **Explanation:**
    - Utilizes a max heap to keep track of elements in the window along with their indices.
    - Pushes the current element into the max heap along with its index.
    - Removes elements from the max heap that are no longer in the current window.
    - Once the window size is reached, adds the maximum element in the window to the answer.
- **Time Complexity:**
    - **Overall Time Complexity: O(n log k).**
- **Space Complexity:**
    - **O(k) - The max heap stores at most 'k' elements.**

Conclusion

The **Deque-based approach** is often preferred for its efficiency with a linear time complexity. It strikes a good balance between simplicity and performance, making it a versatile choice for many scenarios.