Sliding Window Maximum Problem

Problem Statement:

You are given an array of integers nums, and a sliding window of size k moves 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 maximum of each sliding window.

Approach Used:

The problem requires us to efficiently determine the maximum element for every sliding window of size k. A naive brute-force approach would involve checking all elements in every window, leading to O(N*K) complexity, which is inefficient for large arrays. Instead, we use a **deque (double-ended queue)** to optimize the process and achieve an **O(N) solution**.

Logic and Explanation:

To efficiently find the maximum in each window, we maintain a **monotonic decreasing deque**, which always holds the indices of the elements in the current window in decreasing order of their values. This ensures that the **maximum element** is always at the front of the deque.

Step-by-step breakdown of the logic:

1. **Maintaining the Deque:**
   - The deque will store indices of elements in nums, but in such a way that the **largest element in the current window is always at the front**.
2. **Processing each element nums[i] in the array:**
   - First, remove all elements from the **back of the deque** that are **smaller than nums[i]**, because they are useless (they will never be the max in the current or future windows).
   - Add the **current index i** to the **back of the deque**.
   - If the **front of the deque** is out of the current window's range (i.e., dq.front() < i + 1 - k), remove it.
3. **Extracting the Maximum:**
   - Once we have processed at least k elements (i >= k-1), the **maximum of the current window is the element at the front of the deque** (nums[dq.front()]).
   - Push this maximum into the result array.

Code Implementation:

```
vector<int> maxSlidingWindow(vector<int>& v, int k){
  if(k == 1) return v;
```

```
    int n = v.size();
    vector<int> ans;
    deque<int> dq;

    for(int i = 0; i < n; i++){
        // Remove elements smaller than v[i] from the back of the deque
        while(dq.size() > 0 && v[i] >= v[dq.back()]) dq.pop_back();

        // Push current index into the deque
        dq.push_back(i);

        // Remove elements that are out of the window
        while(dq.front() < i + 1 - k) dq.pop_front();

        // Store the maximum for the current window
        if(i >= k - 1){
            ans.push_back(v[dq.front()]);
        }
    }
    return ans;
}
```

Time and Space Complexity Analysis:

Time Complexity:

- Each element is pushed and popped from the deque **at most once**.
- Hence, the overall time complexity is **O(N)**.

Space Complexity:

- The deque stores indices of elements, at most k elements.
- The space complexity is **O(k)** (which is O(N) in the worst case).

Why This Approach is Efficient?

- **Better than Brute Force (O(N*K))**: Instead of checking every subarray separately, we maintain a deque for efficient retrieval of the max element.
- **Deque Operations are O(1)**: Each element is processed only once, making it linear time.
- **Sliding Window Optimization**: The deque helps in keeping track of max values dynamically.

Example Walkthrough:

Input:

nums = [1,3,-1,-3,5,3,6,7], k = 3

Deque Evolution:

Step	Window	Deque (indices)	Max Value
1	[1,3,-1]	[1]	3
2	[3,-1,-3]	[1,2]	3
3	[-1,-3,5]	[4]	5
4	[-3,5,3]	[4,5]	5
5	[5,3,6]	[6]	6
6	[3,6,7]	[7]	7

Output:

[3,3,5,5,6,7]

Edge Cases Considered:

1. k == 1: Directly return the input array as every element is its own window.
2. k == n: There is only one window, return the max of the entire array.
3. All elements are in increasing/decreasing order.
4. Handling of negative numbers in the array.

Final Thoughts:

This approach effectively uses deque to optimize the solution, reducing the brute force complexity from O(N*K) to O(N). This makes it a powerful technique for solving **Sliding Window Maximum** problems efficiently.

Alternative Approaches:

1. **Max Heap (Priority Queue) Approach**
   - Time Complexity: O(N log K), as insertion and deletion in a heap take O(log K).
   - Space Complexity: O(K), as we store at most k elements in the heap.
2. **Segment Tree Approach**
   - Preprocess the array in O(N log N).
   - Query the max in O(log N) for each window.
   - Overall Complexity: O(N log N), which is worse than the deque method for large inputs.

Thus, the **deque-based approach** remains the best for practical use!