Sliding Window Maximum (LeetCode 239) - Optimized Approach

Problem Statement

Given an array `nums` and an integer `k`, there is a sliding window of size `k` that moves from the left to the right of the array. At each position, we can only see `k` elements inside the window. The task is to find the maximum in each window as it slides.

Approach

Since this is a **hard** problem, brute-force is not an optimal solution due to high time complexity. Instead, we optimize using **Next Greater Index (NGI) and a stack**.

Steps Taken:

1. **Precompute Next Greater Index (NGI)**:
- We traverse the array in reverse using a **monotonic decreasing stack** to store the **next greater element index**.
- If a number has no greater element on the right, we assign `n` as its NGI.
2. **Use NGI for Efficient Window Maximum Calculation**:
- Initialize `j = 0` to track the max element in the sliding window.
- Iterate from `i = 0` to `i <= n - k` to cover all window positions.
- For each window:
- If `j < i`, reset `j = i` to ensure it starts within the window.
- Move `j` using `ngi[j]` until it steps outside the window.
- The value at `nums[j]` is the max for this window.

Code Implementation

```cpp
class Solution {
public:
vector<int> maxSlidingWindow(vector<int>& v, int k) {
int n = v.size();
stack<int> st;
vector<int> ngi(n);
st.push(n-1);
ngi[n-1] = n;
for(int i = n-2; i >= 0; i--) {
while(st.size() > 0 && v[i] > v[st.top()]) {
st.pop();
}
if(st.size() > 0) {
ngi[i] = st.top();
} else if(st.size() == 0) {
ngi[i] = n;
}
st.push(i);
}

vector<int> ans;
int j = 0;
for(int i = 0; i <= n-k; i++) {
if(j < i) j = i;
int Max = v[i];
while(j < i + k) {
Max = v[j];
if(ngi[j] > i + k) break;
j = ngi[j];
}
ans.push_back(Max);
}
return ans;
}
};
```

Complexity Analysis

- **Precomputing NGI**: `O(n)` using a stack.
- **Finding max in each window**: `O(n)`, as `j` only moves forward.
- **Overall Complexity**: `O(n)`, making it highly efficient.

Summary

- Used **Next Greater Index (NGI) with a monotonic stack** to precompute the next greater element indices.
- Optimized the sliding window traversal by efficiently skipping unnecessary comparisons.
- Achieved a time complexity of `O(n)`, making it scalable for large inputs.

Notes

- **Key Idea**: Instead of checking all `k` elements in a window, use **precomputed NGI** to

jump to the next max efficiently.
- **Mistake to Avoid**: Always ensure `j` remains within bounds before accessing `nums[j]`.
- **Alternative Approaches**: Can also be solved using a **deque** for maintaining a decreasing sequence of indices.

This method ensures an optimal and scalable solution for finding the maximum in every sliding window!