239. Sliding Window Maximum

Hard Queue Array Sliding Window Monotonic Queue Heap (Priority Queue) Leetcode Link

Problem Description

In this problem, you're given an array `nums` of integers, and an integer `k` which represents the size of a sliding window. This sliding window moves across the array from left to right, one element at a time. At each position of the window, you can only see `k` numbers within it. The objective is to return a new array that contains the maximum value from each of the windows as it slides through the entire array.

To illustrate, imagine a section of `k` numbers from the array `nums` as a frame that only allows you to view those `k` numbers. As the frame moves to the right by one element, the oldest number in the frame is replaced with a new number from the array. You need to find the maximum number in the frame after each move and record it.

Intuition

The intuition behind the solution arises from the need to efficiently manage the current candidates for the maximum value within the sliding window. We need a way to update the candidates as the window slides through the array, adding new elements and removing old ones.

One efficient approach to solve this problem is to use a double-ended queue (deque) that maintains a descending order of values (indices of `nums`) from the largest at the front to the smallest at the back. The front of the deque will always hold the index of the maximum number within the current window. This allows us to quickly retrieve the maximum number by looking at the element at the front of the deque.

Here are the steps to arrive at the solution:

1. Initialize a deque `q` to hold indices of `nums` and an empty list `ans` to hold the maximums.
2. Iterate through the indices and values of `nums` using a loop.
3. Check if the deque is non-empty and if the element at the front of the deque is out of the sliding window's range. If so, pop it from the front to remove it from our candidates.
4. While there are elements in the deque and the value at index `i` of `nums` is greater than or equal to the value at the indices stored in the deque's back, pop elements from the back of the deque. This is because the new value could potentially be the new maximum, and thus we remove all the values which are less than the current value since they cannot be the maximum for future windows.
5. Append the current index `i` to the deque.
6. If `i` is greater than or equal to `k-1`, it means we've filled the first window. The current maximum (the value at the front of the deque) is added to `ans`.
7. Continue this process until the end of the array is reached.
8. Return the `ans` list.

Using a deque allows us to efficiently remove indices/values from both ends and keep our window updated. This solution ensures that we have a near constant time operation for each window, making it an efficient solution.

Solution Approach

The key to the efficient solution is using a double-ended queue (deque), a powerful data structure that allows insertion and deletion from both the front and the back in `O(1)` time complexity.

Here's a step-by-step explanation of the algorithm:

1. **Initialization**: We start by initializing an empty deque `q` and an empty list `ans` which will store the maximum values of the current sliding window.

```python
q = deque()
ans = []
```

2. **Iterating through `nums`**: We use a for loop to iterate over the array. Each iteration represents a potential new frame of the sliding window:

```python
for i, v in enumerate(nums):
```

3. **Maintain Window Size**: Before processing the new element at each step, we ensure that the deque's front index belongs to the current window:

```python
if q and i - k + 1 > q[0]:
    q.popleft()
```

4. **Maintain Deque Properties**: We pop indices from the back of the deque as long as the current value is larger than the values at the indices at deque's rear. This ensures that the deque is decreasing so the front always has the largest element:

```python
while q and nums[q[-1]] <= v:
    q.pop()
```

5. **Add New Index**: We then add the index of the current element to the deque back.

```python
q.append(i)
```

6. **Record the Maximum**: Once we reach at least the `k`th element (`i >= k - 1`), we find a maximum for the full-sized window which we record by adding the value at the front of the deque to `ans` list:

```python
if i >= k - 1:
    ans.append(nums[q[0]])
```

7. **Return Result**: After the loop completes, `ans` contains the maximums for each sliding window, which we return.

This algorithm hinges crucially on the deque's properties to keep the indices in a descending order of their values in `nums`. Keeping the deque in descending order ensures that the element at the front is the largest and should be included in `ans`. When the window moves to the right, indices that are out of the window or not relevant (because there's a larger or equal value that came later) are popped out from the deque, maintaining the required properties.

Example Walkthrough

Consider the array `nums = [1,3,-1,-3,5,3,6,7]` and a sliding window size `k = 3`. Let's walk through the solution step by step:

1. **Initialization**: An empty deque `q` and an empty list `ans` are initialized.

```python
q = deque() # deque to hold indices of nums
ans = []    # list to hold max values of each window
```

2. **Iterating through `nums`**: We start iterating over the array with indices and values.

3. **First Iteration (i=0, v=1)**:
   - No indices to remove from deque, since it's empty.
   - Append index 0 to the deque.
   - Window is not yet full (`i < k-1`), so we don't record the maximum.
4. **Second Iteration (i=1, v=3)**:
   - The deque still contains only one index `[0]`.
   - Since `3` (nums[1]) is greater than `1` (nums[q[-1]]), we pop index `0` and then append index `1` to the deque.
   - Window is not yet full (`i < k-1`), so we don't record the maximum.
5. **Third Iteration (i=2, v=-1)**:
   - The deque contains `[1]` and since `-1` is not greater than `3`, we only append index `2`.
   - Now `i >= k-1`, so we record the current maximum `nums[q[0]]`, which is `3`.
6. **Subsequent Iterations**:
   - Continue sliding the window, maintaining the deque by removing out-of-range indices and values smaller than the current one.
   - After each slide, if the window is full, add the maximum to `ans`.
7. **After Sliding Through Entire Array**:
   - The final `ans` list will be `[3, 3, 5, 5, 6, 7]`.

This walkthrough demonstrates the solution using a deque to efficiently track the maximum of the current sliding window in the array `nums`.

Python Solution

```python
from collections import deque

class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        # Initialize a deque for storing indices of elements
        index_queue = deque()
        # List to store the maximums for each sliding window
        max_values = []

        # Iterate over each element, with its index
        for current_index, value in enumerate(nums):
            # If the first element in the queue is outside the current window, remove it
            if index_queue and current_index - k + 1 > index_queue[0]:
                index_queue.popleft()

            # Remove elements from the back of the queue if they are smaller than
            # or equal to the current element since they will not be needed anymore
            while index_queue and nums[index_queue[-1]] <= value:
                index_queue.pop()

            # Add the current index to the queue
            index_queue.append(current_index)

            # If we have reached or passed the first complete window, record the maximum
            if current_index >= k - 1:
                max_values.append(nums[index_queue[0]])

        # Return the list of maximums
        return max_values

```

Java Solution

```java
class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        int numsLength = nums.length;
        int[] result = new int[numsLength - k + 1]; // Array to store the max values for each window
        Deque<Integer> deque = new ArrayDeque<>(); // Double-ended queue to maintain the max element indices

        for (int i = 0, j = 0; i < numsLength; ++i) {
            // Remove the indices of elements from the deque that are out of the current window
            if (!deque.isEmpty() && i - k + 1 > deque.peekFirst()) {
                deque.pollFirst();
            }

            // Remove indices of elements from the deque that are less than
            // the current element nums[i] since they are not useful
            while (!deque.isEmpty() && nums[deque.peekLast()] <= nums[i]) {
                deque.pollLast();
            }

            // Add current element's index to the deque
            deque.offer(i);

            // When we've hit size k, add the current max to the result
            // This corresponds to the index at the front of the deque
            if (i >= k - 1) {
                result[j++] = nums[deque.peekFirst()];
            }
        }

        // Return the populated result array containing max of each sliding window
        return result;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <deque>
using namespace std;

class Solution {
public:
    // This function computes the maximum value in each sliding window of size k in the array nums
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque<int> windowIndices;  // Deque to store indices of elements in the current window
        vector<int> maxValues;     // Vector to store the maximum value for each window

        for (int i = 0; i < nums.size(); ++i) {
            // Remove indices of elements not in the current window
            if (!windowIndices.empty() && i - k >= windowIndices.front()) {
                windowIndices.pop_front();
            }

            // Maintain the elements in decreasing order in the deque
            // Pop elements from the back that are less than or equal to the current element
            while (!windowIndices.empty() && nums[windowIndices.back()] <= nums[i]) {
                windowIndices.pop_back();
            }

            // Push current element's index onto the deque
            windowIndices.push_back(i);

            // If we've reached the end of the first window, record the max for the current window
            if (i >= k - 1) {
                maxValues.emplace_back(nums[windowIndices.front()]);
            }
        }

        return maxValues;  // Return the list of maximum values
    }
};

```

Typescript Solution

```typescript
/**
 * Calculates the max sliding window for an array of numbers.
 * @param {number[]} nums - The input array of numbers.
 * @param {number} k - The size of the sliding window.
 * @return {number[]} - Array of the maximum numbers for each sliding window.
 */
function maxSlidingWindow(nums: number[], k: number): number[] {
    // Initialize an array to hold the maximum values for each sliding window.
    let maxValues: number[] = [];

    // Initialize a deque to store indices of elements in nums.
    // Elements in the deque are in decreasing order from the start (front) to the end (back).
    let deque: number[] = [];

    // Iterate through nums using 'i' as the right boundary of the sliding window.
    for (let i = 0; i < nums.length; ++i) {
        // If the left boundary of the window has exceeded the left-most index in the deque,
        // remove the left-most index as it's no longer in the window.
        if (deque.length && i - k + 1 > deque[0]) {
            deque.shift();
        }

        // While the deque is not empty and the current element is greater than the
        // last element indexed in deque, remove the last element from the deque.
        // This ensures elements in the deque are always in decreasing order.
        while (deque.length && nums[deque[deque.length - 1]] <= nums[i]) {
            deque.pop();
        }

        // Add the current index to the deque.
        deque.push(i);

        // If we have hit the size of the window, append the maximum value (front of the deque)
        // to the maxValues array.
        if (i >= k - 1) {
            maxValues.push(nums[deque[0]]);
        }
    }

    // Return the array containing the maximum for each sliding window.
    return maxValues;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the number of elements in the array `nums`. This is because the code iterates through each element in `nums` once. Each element is added to the deque `q` once, and it's potential removal is also done at most once due to the while loop condition. Despite the nested while loop, the overall number of operations for each element remains constant, ensuring a linear time complexity.

Space Complexity

The space complexity is `O(k)`, where `k` is the size of the sliding window. The deque `q` is used to store indices of elements and its size is restricted by the bounds of the sliding window, which at most can have `k` indices stored at a time. Additionally, the output list `ans` grows linearly with `n - k + 1` (the number of windows), but it does not depend on the input size in a nested or exponential manner, so it does not affect the asymptotic space complexity with respect to `k`.