1004. Max Consecutive Ones III

Medium Array Binary Search Prefix Sum Sliding Window

Problem Description

Given an array of binary numbers `nums` (i.e., containing only `0`s and `1`s) and an integer `k`, the goal is to find the maximum number of consecutive `1`s that can be achieved in the array if you are allowed to flip at most `k` `0`s to `1`s. A "flip" means changing a `0` to a `1`. The task is to apply these flips in the most optimal way to maximize the length of a continuous segment of `1`s in the array.

For example, if your array is `[1,1,0,0,1]` and `k` is `1`, the longest sequence of `1`s you can create by flipping at most one `0` is `3` (flip the first `0` and the sequence becomes `[1,1,1,0,1]`).

Intuition

The problem hints at a sliding window approach where we iterate over the array while maintaining a window of consecutive numbers which can potentially include up to `k` zeroes.

Here's the thinking process behind the solution:

- We aim to find the longest sequence of 1s which may include `k` or fewer flipped 0s.
- We initialize two pointers `l` (left) and `r` (right) which define the current window. The window starts with no elements (both pointers at `-1`).
- We iterate through the array with the `r` pointer incremented in each step examining each element. We expand the window by moving `r` to the right.
- If a `0` is encountered, we decrease the count of `k` because we are envisioning a flip happening here to turn it into a `1` for the sake of our current sequence.
- If `k` becomes negative, it means our window has too many `0`s (more than the allowed `k` flips). We need to shrink the window from the left by incrementing `l`. Each time we pass a `0` while moving `l` to the right, we increment `k` back as we are effectively undoing a flip.
- Throughout this process, we are implicitly tracking the length of the current window. The maximum length encountered during the iteration is the length of the longest sequence we are looking for.
- The length of the window is always `r - l`, but since our pointers start from `-1`, when the right pointer has reached the end of the array (index `len(nums) - 1`), the window length is simply `r - l` without needing to add one for usual zero-based index arrays.

In the provided code, the final return statement seems incomplete as it doesn't consider the fact that the length of the window should be `r - l + 1`. However, considering that the looping mechanism does not immediately stop when `k` becomes less than `0` (it needs one more iteration where it could potentially adjust the window size appropriately), the code might yield the correct result through careful pointer manipulation and implicit window size calculation.

Solution Approach

The solution approach uses a two-pointer technique to implement a sliding window algorithm, which is particularly efficient for problems involving contiguous subarrays or subsequences.

Here's a breakdown of the implementation steps and logic:

- Initialize two pointers, `l` and `r`, that will represent the window's left and right boundaries, respectively. The variables `l` and `r` both start at `-1` because the window is initially empty.

- Iterate over the array with a while loop. The right boundary `r` of the window will increase with each iteration (the window expands to the right) until it reaches the end of `nums`.

- For each number encountered, check if it is a `0` and, if it is, decrement `k`. Each time `k` is decremented, it's as though we flipped a `0` to `1` within our current window. We are allowed to have a maximum of `k` such flips.

- When `k` becomes negative (which means the current window has one more `0` than allowed), increment the left boundary `l` (the window contracts from the left), and if the leftmost element is a `0`, increment `k` again, essentially 'unflipping' that zero.

- The conditional check for `k < 0` doesn't necessarily equate to an else condition following the `if nums[r] == 0`; it's a separate check because the sliding window can continue to slide, and `k` can go negative at different points in the loop.

- In terms of data structures, the solution does not require anything beyond the basic built-in variables and the array itself. It's the algorithmic pattern (two pointers for the sliding window) that carries the weight of the logic.

- The length of the window, and thus the maximum number of consecutive `1`'s, can always be represented by the difference `r - l`. Upon leaving the while loop, since `r` has moved one past the end of the array, the subtraction directly yields the size of the window, without the need for additional adjustment.

It is important to note that the apparent simplicity of the final `return r - l` statement may obscure the fact that such a result is valid because of the particular initial conditions for `l` and `r` and the way the algorithm carefully adjusts the window. An accurate count of the longest stretch of `1`'s is maintained throughout the algorithm's execution without a need for an explicit length variable or further calculations.

The code solution effectively exploits the sliding window pattern to perform a single pass through the array, yielding an efficient `O(n)` runtime complexity (because each element is visited once) and `O(1)` space complexity since no additional data structure is used.

Example Walkthrough

Let's illustrate the solution approach using a small example where the input array is `nums = [1, 0, 1, 0, 1, 0, 1]` and `k = 2`, meaning we can flip at most 2 `0`s to `1`s.

1. Initialize two pointers, `l` and `r`, to `-1`. These will mark the boundaries of our sliding window.
2. Start iterating over the array. The window will expand as `r` moves to the right.
3. On each iteration, check each number:
   - For `r = 0`, `nums[r] = 1`. No action is required since the number is already `1`. Window remains the same.
   - For `r = 1`, `nums[r] = 0`. Decrement `k` to `1`. We're envisioning a flip here so the window can keep growing.
   - For `r = 2`, `nums[r] = 1`. No action, and the window grows further.
   - For `r = 3`, `nums[r] = 0`. Decrement `k` to `0`. Another envisioned flip and the window continues.
   - For `r = 4`, `nums[r] = 1`. No action required.
   - For `r = 5`, `nums[r] = 0`. Our `k` is now `-1`, which means we've exceeded the maximum number of flips permitted. Now we shrink the window from `l`.
4. While `k` is negative, increment `l` to shrink the window from the left until `k` is no longer negative.
   - Move `l` from `-1` to `0`, no change since `nums[0] = 1`.
   - Move `l` to `1`, and since `nums[1] = 0`, increment `k` to `0` (unflipping that zero).
   - The window is still too large (as `k` is `0`, not positive), so continue moving `l`.
   - Move `l` to `2`. There is no need to adjust `k` since `nums[2] = 1`.
   - Move `l` to `3`, `k` becomes `1` because `nums[3] = 0`, and we 'unflip' it to go back to suit our flip limit.
5. Continue iterating and growing the window to the right.
   - For `r = 6`, `nums[r] = 1`. No flips needed.
6. Iteration ends when `r` has reached the last element.

We've grown and shrunk our window according to the flips we can perform, and we've kept track of the window size implicitly through the difference `r - l`. In this example, our largest window size occurs at the end of the iteration: `r = 6`, `l = 3`, giving us a window length of `6 - 3 = 3` (which is the number of `1`'s in a row after flipping at most `k` `0`'s).

So, the maximum number of consecutive `1`s achievable by flipping at most `k` `0`s is `3`, and the sequence where that happens after flipping is `[1, 1, 1, 0, 1, 1, 1]`.

Python Solution

```python
from typing import List

class Solution:
    def longestOnes(self, nums: List[int], k: int) -> int:
        # Initialize the left and right pointer to -1
        left = right = -1

        # Slide the window to the right until the end of the list is reached
        while right < len(nums) - 1:
            right += 1  # Move the right pointer to the right

            # If a zero is encountered, decrement k (number of flips allowed)
            if nums[right] == 0:
                k -= 1

            # If k is negative, too many zeros have been flipped
            # thus, move the left pointer to the right
            if k < 0:
                left += 1  # Move the left pointer to the right

                # If the left pointer is pointing to a zero, increment k
                if nums[left] == 0:
                    k += 1

        # The maximum length of subarray with all ones after flipping k zeros is
        # the difference between the right and left pointers
        return right - left

```

Java Solution

```java
class Solution {

    // Method to find the longest continuous segment with '1's after flipping at most 'k' zeroes
    public int longestOnes(int[] nums, int k) {
        int left = 0;    // Initialize the left pointer of the window
        int right = 0;   // Initialize the right pointer of the window
        int maxOnesCount = 0; // Variable to store the count of continuous ones

        while (right < nums.length) { // Iterate until the right pointer reaches the end of the array
            // If the element at the right pointer is 0, we have to use one flip
            if (nums[right] == 0) {
                --k;
            }

            right++; // Move the right pointer to the next element

            // If we have used all our flips, we need to move the left pointer forward to find a new window
            while (k < 0) {
                // If we are moving the left pointer over a zero, we can have one more flip available
                if (nums[left] == 0) {
                    ++k;
                }
                left++; // Move the left pointer to the next element
            }

            // Update the maximum number of continuous ones found
            maxOnesCount = Math.max(maxOnesCount, right - left);
        }

        return maxOnesCount; // Return the length of the longest continuous segment of ones possible
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int longestOnes(vector<int>& nums, int k) {
        int left = 0;   // Initialize left pointer
        int right = 0;  // Initialize right pointer
        int maxOnesLength = 0;  // Variable to store the maximum length of subarray

        // Iterate through the array with the right pointer
        while (right < nums.size()) {
            // If we encounter a 0, decrement k (the flip count)
            if (nums[right] == 0) {
                --k;
            }

            // Move the right pointer to the next element
            ++right;

            // If k is negative, it means we've flipped more 0s than allowed
            while (k < 0) {
                // If the left element is 0, we increment k
                // since we are moving past the flipped zero
                if (nums[left] == 0) {
                    ++k;
                }
                // Move the left pointer to the right, effectively shrinking the window
                ++left;
            }

            // Update maxOnesLength if the current window is larger
            maxOnesLength = max(maxOnesLength, right - left);
        }

        // Return the maximum length of the subarray
        return maxOnesLength;
    }
};

```

Typescript Solution

```typescript
// The function finds the maximum number of consecutive 1's in the array if you
// can flip at most k 0's to 1's.
function longestOnes(nums: number[], k: number): number {
    // Total number of elements in the nums array.
    const lengthOfNums = nums.length;
    // Initialize the left pointer for the sliding window.
    let leftPointer = 0;
    // Iterate through the nums array.
    for (const num of nums) {
        // If we encounter a 0, reduce the count of flips remaining (k).
        if (num === 0) {
            k--;
        }
        // If k is negative, it means we have flipped more than k 0's.
        // We need to adjust the leftPointer until k is non-negative.
        if (k < 0 && nums[leftPointer++] === 0) {
            // By moving leftPointer forward and encountering a 0, we flip it back to 0 ('unflip'),
            // thus incrementing k.
            k++;
        }
    }
    // Calculate the maximum length of consecutive 1's that can be achieved by flipping
    // at most k 0's. This is done by subtracting the leftPointer from the total length
    // of the array, which gives us the size of the current window.
    return lengthOfNums - leftPointer;
}

```

Time and Space Complexity

The given code snippet uses a sliding window approach to solve the problem of finding the longest subarray with at most `k` zeroes that can be flipped to `1`s.

Time Complexity: The algorithm essentially processes each element of the array twice: once when extending the right end of the window (`r`) and once when shifting the left end of the window (`l`). This results in a linear pass over the array with `n` elements total. The complexity is, therefore, `O(n)`, where `n` is the number of elements in the input list `nums`.

Space Complexity: The solution uses a few variables to track the indices of the window (`l`, `r`, and `k`) and does not utilize any data structures that scale with the input size. Therefore, the space complexity of this solution is `O(1)` as it requires a constant amount of extra space.