487. Max Consecutive Ones II

Medium | Array | Dynamic Programming | Sliding Window | Leetcode Link

Problem Description

The problem is about finding the longest sequence of consecutive `1`s in a binary array, under the condition that we are allowed to flip at most one `0` to `1`. This task tests our ability to manipulate subarrays in a binary context and optimize our approach to account for small alterations in the array to achieve the desired outcome.

Intuition

To find the solution to this problem, we look to a two-pointer approach. Essentially, we're trying to maintain a window that can include up to one `0` to maximize the length of consecutive ones.

Let's think of this as a sliding window that starts from the left end of the array and expands to the right. As we move the right pointer (`r`) through the array, we keep track of the number of `0`s we've included in the window. We initialize `k` to `1`, because we're allowed to flip at most one `0`.

When we encounter a `0`, we decrement `k`. If `k` becomes negative, it means we've encountered more than one zero, and thus, we need to shrink our window from the left by moving the left pointer (`l`) to the right.

We continue expanding and shrinking the window as needed to always maintain at most one `0` within it. Throughout the process, we keep track of the maximum window size we've seen, which gives us the longest sequence of consecutive `1`s with at most one `0` flipped.

When we have finished iterating through the array with our right pointer, the length of the window (`r - l`) is our maximum number of consecutive `1`s, accounting for flipping at most one `0`. The code does not explicitly keep a maximum length variable; instead, it relies on the fact that the window size can only increase or stay the same throughout the iteration, because once the window shrinks (when `k` becomes negative), it will then begin to expand again from a further position in the array.

Solution Approach

The solution uses a two-pointer technique to efficiently find the maximum length of a subarray with consecutive `1`s after at most one `0` has been flipped.

Here is a step-by-step breakdown of the implementation:

1. Initialize two pointers, `l` and `r`, to point at the start of the array. These pointers represent the left and right bounds of our sliding window, respectively.
2. Define a variable `k` with an initial value of `1`, which represents how many `0`s we can flip. Since we are allowed to flip at most one `0`, we start with `k = 1`.
3. Iterate through the array by moving the right pointer `r` towards the end. This loop continues until `r` reaches the end of the array.
4. Within the loop, check if the current number pointed to by `r` is a `0`. If it is, decrement `k`.
5. If `k` becomes less than 0, it signifies that our window contains more than one `0`, which is not allowed. Thus, we need to increment `l`, our left pointer, to narrow the window and possibly discard a `0` from the window:
   - Inside another loop, we check if the number at the current `l` position is a `0`. If it is, we increment `k` because we are "flipping" the `0` back and excluding it from the current window.
   - Then, we move `l` one step to the right by increasing its value by `1`.
6. Increment `r` each time to examine the next element in the array.
7. After the while loop exits, calculate the length of the current window (`r - l`). Since the right pointer has reached the end of the array, this value equals the maximum size of the window we found throughout our traversal.

Important notes regarding the code and algorithm:

- There is no need to explicitly keep track of the maximum window size during the loop because the window can only grow or maintain its size. It shrinks only when necessary to exclude an excess `0`.
- The solution leverages the problem constraint. Knowing that only one `0` can be flipped, it eliminates the need for complex data structures or algorithms. A straightforward integer (`k`) is enough to keep track of the condition being met.
- This solution has a time complexity of `O(n)` where `n` is the length of the array. This is because each element is checked once by the right pointer `r`.
- The space complexity of this algorithm is `O(1)` as it operates with a constant number of extra variables regardless of the input size.

By keeping the algorithm simple and avoiding unnecessary data structures, the solution remains elegant and efficient.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Consider the binary array `[1, 0, 1, 1, 0, 1, 1, 1, 0, 1]`, where we are allowed to flip at most one `0` to achieve the longest sequence of `1`s.

Following the solution steps:

1. Initialize two pointers `l` and `r` to `0` (the start of the array).
2. Set `k` to `1` since we can flip at most one `0`.
3. As we start iterating with `r`, the subarray from `l` to `r` initially grows without any issue since the first element is a `1`.
4. When `r=1`, the element is `0`. We decrement `k` to `0` because we used our allowance to flip a `0`.
5. We continue to move `r`. The window now has consecutive `1`s and looks like this: `[1, 1 (flipped), 1, 1]`.
6. At `r=4`, we encounter another `0` and `k` is already `0`. Now we must shrink the window from the left. We move `l` one step to the right. The subarray does not contain zero at `l=1`, hence no change in `k`.
7. Moving `l` again to `2`, we encounter our previously flipped `0`. We increment `k` back to `1`.
8. The window is now `[1, 1, 0, 1, 1]` from `l=2` to `r=5`, and we can flip the `0` at `r=4` because `k=1`.
9. We continue this process, moving `r` to the end, and each time we hit a `0` with `k=0`, we shift `l` to maintain only one flipped `0`.
10. At the end, when `r` is just past the end of the array, the length of the window is calculated by `r - l`, giving us the longest sequence of consecutive `1`s where at most one `0` was flipped.

In this example, the longest subarray we can form after flipping at most one `0` is `[1, 1, 0 (flipped), 1, 1, 1]` which starts at `l=2` and ends just before `r=8`, resulting in a length of `6`.

Python Solution

```python
from typing import List

class Solution:
    def findMaxConsecutiveOnes(self, nums: List[int]) -> int:
        # Initialize the left and right pointers
        left_pointer = right_pointer = 0
        zero_count = 1  # Variable to allow flipping of one '0' to '1'

        # Traverse the array while the right_pointer is within the array bounds
        while right_pointer < len(nums):
            # Decrease zero_count when a '0' is encountered
            if nums[right_pointer] == 0:
                zero_count -= 1

            # If zero_count is less than 0, slide the window to the right
            # by moving the left_pointer to the next position
            if zero_count < 0:
                # When we move the left_pointer forward, we need to check if
                # we are passing over a '0' and if so, increase zero_count
                if nums[left_pointer] == 0:
                    zero_count += 1
                # Move the left pointer to the right
                left_pointer += 1

            # Move the right pointer to the right
            right_pointer += 1

        # The length of the longest subarray of 1's (possibly with one flipped 0)
        # is the difference between the right and left pointers.
        return right_pointer - left_pointer

```

Java Solution

```java
class Solution {
    public int findMaxConsecutiveOnes(int[] nums) {
        int left = 0; // Initialize the left pointer
        int right = 0; // Initialize the right pointer
        int zerosAllowed = 1; // Initialize the number of zeros allowed to flip to ones

        // Loop through the array using the right pointer
        while (right < nums.length) {
            // If the current element is 0, decrement the number of zeros allowed
            if (nums[right++] == 0) {
                zerosAllowed--;
            }
            // If no zeros are allowed and the left element is 0, increment the left pointer
            // and the number of zeros allowed
            if (zerosAllowed < 0 && nums[left++] == 0) {
                zerosAllowed++;
            }
        }
        // Compute the length of the longest sequence of 1s (with at most one 0 flipped to 1)
        return right - left;
    }
}

```

C++ Solution

```cpp
#include<vector>

class Solution {
public:
    int findMaxConsecutiveOnes(vector<int>& nums) {
        int left = 0; // Left pointer for the window
        int right = 0; // Right pointer for the window
        int zeroCount = 1; // Initial max number of zeroes allowed to flip

        int maxConsecutive = 0; // Variable to store the maximum length of consecutive ones

        // Iterate over the array
        while (right < nums.size()) {
            // If we encounter a zero, decrement zeroCount
            if (nums[right] == 0) {
                zeroCount--;
            }

            right++; // Expand the window to the right

            // If zeroCount is negative, it means we have encountered
            // more than one zero in our window. We then need to shrink
            // the window from the left
            while (zeroCount < 0) {
                if (nums[left] == 0) { // If we're moving past a zero, increment zeroCount
                    zeroCount++;
                }
                left++; // Shrink the window from the left
            }

            // Calculate the length of the current window and update maxConsecutive if it's larger
            int windowLength = right - left;
            maxConsecutive = max(maxConsecutive, windowLength);
        }

        return maxConsecutive; // Return the maximum length of consecutive ones found
    }
};

```

Typescript Solution

```typescript
function findMaxConsecutiveOnes(nums: number[]): number {
    let left = 0; // Left pointer for the sliding window
    let right = 0; // Right pointer for the sliding window
    let zeroCount = 1; // Number of zeroes allowed to flip (fixed at one according to the problem)

    let maxConsecutive = 0; // To store the maximum length of consecutive ones

    // Iterate over 'nums' using the right pointer as the leader of the sliding window
    while (right < nums.length) {
        // If a zero is encountered, decrement 'zeroCount'
        if (nums[right] === 0) {
            zeroCount--;
        }

        // Move the right pointer to the right, expanding the window
        right++;

        // If 'zeroCount' is less than 0, more than one zero has been encountered
        // Start shrinking the window from the left until 'zeroCount' is not negative
        while (zeroCount < 0) {
            // If we move past a zero on the left, increment 'zeroCount'
            if (nums[left] === 0) {
                zeroCount++;
            }
            // Increment the left pointer to shrink the window from the left
            left++;
        }

        // Calculate the current window length
        const windowLength = right - left;

        // Keep track of the maximum length of ones encountered
        maxConsecutive = Math.max(maxConsecutive, windowLength);
    }

    // Return the maximum number of consecutive ones found
    return maxConsecutive;
}

```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the input list `nums`. This efficiency is achieved because the code uses a two-pointer technique that iterates through the list only once. The pointers `l` (left) and `r` (right) move through the array without making any nested loops, thus each element is considered only once during the iteration.

The space complexity of the code is `O(1)`, which means it uses constant additional space regardless of input size. No extra data structures that grow with the input size are used; the variables `l`, `r`, and `k` take up a constant amount of space.