2401. Longest Nice Subarray

Medium | Bit Manipulation | Array | Sliding Window

Leetcode Link

Problem Description

This LeetCode problem requires us to find the longest subarray in a given array `nums` of positive integers where the subarray is considered *nice*. A subarray is defined as nice if the bitwise AND of every pair of elements at different positions is equal to `0`. A subarray is a contiguous part of the array, and it is given that any subarray with a length of `1` is always nice.

Intuition

The solution for the problem leverages the properties of bitwise AND operation. The bitwise AND of any two numbers is `0` only if they do not share any common set bits in their binary representation.

The approach taken here uses a sliding window technique to iterate over the array, using two pointers (or indices) to define the current window which is examined for being a nice subarray. As we iterate from `0` to `n-1` (where `n` is the length of the array), we maintain a variable `mask` which is the bitwise OR of all numbers in the current window. This mask helps in checking if any incoming number will create a pair with a non-zero AND with any number in the current window.

When we attempt to extend the window by including the next number, we check if the bitwise AND of this new number (`x`) with the current `mask` is zero. If it is not zero, this means the number shares at least one common set bit with one of the numbers in the current window and hence cannot be included to maintain it a nice subarray. We then shrink the window from the left by removing the leftmost element and updating the mask until the mask AND the new number is `0`.

The variable `ans` keeps track of the length of the longest nice subarray encountered so far, and we update it every time we find a larger nice subarray. The current window's length is calculated as `i - j + 1`, where `i` is the end of the current window and `j` is the start.

By scanning and adjusting the window this way until the end of the array is reached, we ensure that the longest nice subarray is found.

Solution Approach

The reference solution uses a sliding window technique to find the length of the longest nice subarray. Here's how the implementation unfolds:

- An integer `ans` is initialized to `0` to keep track of the length of the maximum nice subarray found so far.
- Two pointers, `i` and `j`, are initialized to track the starting and ending index of the current subarray, starting from `0`.
- An integer `mask` is also initiated to `0`. This `mask` will store the bitwise OR of all numbers currently in the window.

The implementation follows these steps:

1. Enumerate through each element `x` in `nums` using its index `i`.
2. While the current number `x` has a non-zero AND with the `mask` (i.e., `mask & x` is not `0`), i.e., if the current element shares a common set bit with any element in the subarray represented by `mask`, it suggests that the subarray is not nice anymore with the addition of `x`:
   - Exclude the leftmost element from the subarray to make room for `x` by XOR-ing the leftmost element `nums[j]` with the `mask`. This effectively removes the bits of `nums[j]` from `mask`.
   - Increment `j` to shrink the subarray from the left.
3. After ensuring that including `x` will not break the nice property (the AND of every pair is `0`), we can:
   - Update the `ans` variable with the maximum of its current value and the size of the window which is `i - j + 1`.
   - Include `x` in the `mask` by performing a bitwise OR (`mask |= x`).

This process continues for every element in the array. By iteratively adjusting the window and updating the `mask`, the algorithm ensures it never includes a pair that would result in a non-zero AND, hence maintaining the nice property of the subarray.

After completing the iteration over all elements, the final value of `ans` yields the length of the longest nice subarray.

Example Walkthrough

Let's walk through an example using the solution approach.

Consider the following array `nums`: `[3,6,1,2]`

- Initialize `ans` to 0.
- Start with `i = 0`, `j = 0`, and `mask = 0`.

Now, we'll iterate over the array while applying the steps outlined in the solution approach:

1. For `i = 0`, `x = nums[0] = 3`.
   - `mask & x` is `0` since `mask` starts at `0` and any number AND 0 is 0.
   - Update `ans` to `max(ans, i - j + 1)`, which is `1`.
   - Update `mask` to `mask | x`, which now becomes `3`.
2. Move to `i = 1`, `x = nums[1] = 6`.
   - `mask & x` is `2` (binary 0010), which is not `0`, so we need to adjust the subarray.
   - We shrink the window by removing `nums[j]` (`3`) from `mask`. New `mask` is `3 XOR 3 = 0`.
   - Increment `j` to `1`, and now the subarray is empty.
   - We retry with `x = 6`, `mask & x` is now `0`, so we can proceed.
   - Update `ans` to `max(ans, i - j + 1)`, which is `1`.
   - Update `mask` to `mask | x`, which now becomes `6`.
3. Move to `i = 2`, `x = nums[2] = 1`.
   - `mask & x` is `0`, it's already a nice subarray upon adding `x`.
   - Update `ans` to `max(ans, i - j + 1)`, which is now `2` as the subarray from `nums[1]` to `nums[2]` is nice.
   - Update `mask` to `mask | x`, which becomes `7`.
4. Move to `i = 3`, `x = nums[3] = 2`.
   - `mask & x` is `2`, which is not `0`, so the subarray is not nice with the addition of `x`.
   - We shrink the window from the left by excluding `nums[j]` (`6`) from `mask`. After XOR with `6`, the new `mask` is `1`.
   - Increment `j` to `2`, and now the subarray starts at `nums[2]`.
   - We retry with `x = 2`, `mask & x` is now `0`, so we can proceed.
   - Update `ans` to `max(ans, i - j + 1)`, which remains `2`.
   - Include `x` in the `mask`, new `mask` is `1 | 2 = 3`.

After going through the array, the final value of `ans` is `2`, indicating the length of the longest nice subarray is `2`, which corresponds to the subarray `[1,2]`.

Python Solution

```python
from typing import List

class Solution:
    def longestNiceSubarray(self, nums: List[int]) -> int:
        # Initialize the maximum length of the subarray, the current start of the subarray,
        # and the mask used to track unique bits
        max_length = start_index = bit_mask = 0

        # Enumerate over the list of numbers
        for end_index, number in enumerate(nums):
            # While there is a common bit set in bit_mask and the current number
            while bit_mask & number:
                # Remove the bits of nums[start_index] from bit_mask
                bit_mask ^= nums[start_index]
                # Move the start_index forward as those bits are no longer part of the subarray
                start_index += 1
            # Update max_length if we've found a longer subarray that's nice
            max_length = max(max_length, end_index - start_index + 1)
            # Include the bits of the current number in the bit_mask
            bit_mask |= number

        # Return the length of the longest nice subarray
        return max_length
```

Java Solution

```java
class Solution {
    public int longestNiceSubarray(int[] nums) {
        // Initialize the answer to track the length of the longest nice subarray
        int longestNiceLength = 0;

        // Create a bitmask to keep track of the bits in the current subarray
        int currentMask = 0;

        // Two pointers - j represents the start of the current subarray
        // i represents the current end of the subarray being considered
        for (int startIdx = 0, endIdx = 0; endIdx < nums.length; ++endIdx) {
            // Keep removing numbers from the start of the subarray until
            // the current number can fit in without sharing any common set bits
            while ((currentMask & nums[endIdx]) != 0) {
                // XOR operation removes the bits of nums[startIdx] from currentMask
                currentMask ^= nums[startIdx++];
            }

            // Update the longestNiceLength if the current subarray is longer
            longestNiceLength = Math.max(longestNiceLength, endIdx - startIdx + 1);

            // Include the current number's bits into the currentMask
            currentMask |= nums[endIdx];
        }

        // Return the length of the longest nice subarray
        return longestNiceLength;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the length of the longest nice subarray
    int longestNiceSubarray(vector<int>& nums) {
        int maxLength = 0; // Variable to store the maximum length found
        int currentMask = 0; // Bitmask to store the current state of numbers in the subarray

        // Two pointers, i for the end of the current subarray, j for the start
        for (int end = 0, start = 0; end < nums.size(); ++end) {
            // While there is a bit overlapping (i.e., not nice subarray), remove the starting number
            while (currentMask & nums[end]) {
                // ^ is the bitwise XOR, which removes nums[start] from currentMask
                currentMask ^= nums[start++];
            }
            // Update maxLength with the larger of the two: previous maxLength or the current subarray length
            maxLength = max(maxLength, end - start + 1);
            // |= is bitwise OR and equals, which adds nums[end] to currentMask
            currentMask |= nums[end];
        }
        // Return the maximum length of nice subarray found
        return maxLength;
    }
};
```

Typescript Solution

```typescript
function longestNiceSubarray(nums: number[]): number {
    let currentSubarrayBitmask = 0;  // bitmask to represent the current subarray

    let maxLength = 0; // this will store the length of the longest nice subarray
    let start = 0;     // start index of the current subarray

    // Iterating through the given array of numbers
    for (let end = 0; end < nums.length; ++end) {
        // Continue removing numbers from the start of the current subarray
        // until the current number and the subarray have no common set bits.
        while ((currentSubarrayBitmask & nums[end]) !== 0) {
            // Using XOR to remove the start number's bits from the bitmask
            currentSubarrayBitmask ^= nums[start++];
        }

        // Update the maximum length of a nice subarray if current subarray is longer
        maxLength = Math.max(maxLength, end - start + 1);

        // Add the current number's bits to the bitmask
        currentSubarrayBitmask |= nums[end];
    }

    // Return the length of the longest nice subarray
    return maxLength;
}
```

Time and Space Complexity

The time complexity of the provided code is `O(N)`, where `N` is the length of the `nums` array. This is because the code iterates over all the elements of `nums` exactly once with the outer loop (`for i, x in enumerate(nums):`). The inner while-loop (`while mask & x:`) only processes each element at most once across the entire runtime because once an element has been removed from the mask (`mask ^= nums[j]`), it does not get reprocessed. Thus, each element contributes at most two operations: one for adding it to the `mask` and one for possibly removing it from the `mask`, leading to a linear runtime overall.

The space complexity of the code is `O(1)` since the amount of extra space used by the algorithm does not scale with the input size `N`. The data structures used (`ans`, `j`, and `mask`) require a constant amount of space regardless of the input size.