2871. Split Array Into Maximum Number of Subarrays

Medium | Greedy | Bit Manipulation | Array

Problem Description

In this problem, we are given an array `nums` containing non-negative integers. We are looking to split the array into one or more subarrays so that two conditions are met:

1. Each element of the original array should belong to exactly one subarray.
2. The sum of the bitwise AND scores of these subarrays should be as small as possible.

A subarray is defined as a contiguous part of the original array. The score of a subarray is calculated by taking the bitwise AND of all the elements in that subarray, ranging from `nums[l]` to `nums[r]` where `l <= r`. The goal is to determine the maximum number of subarrays we can obtain from splitting the array while satisfying the above conditions.

Bitwise AND is a binary operation that takes two equal-length binary representations and performs a logical AND operation on each pair of corresponding bits. In this context, for a sequence of non-negative integers, the result of bitwise AND operation is also a non-negative integer. It only returns 1 for a bit position if both corresponding bits of operands are 1, otherwise, it returns 0.

Intuition

To approach this problem, we can utilize a greedy strategy that leverages the property of the bitwise AND operation. With bitwise AND, the score of a subarray can never be greater than the smallest number in the subarray since adding more numbers with bitwise AND operation either keeps the score the same or decreases it. Our aim is to minimize the sum of the scores of the subarrays, and the minimum score for a subarray is 0.

The key insight is to realize that, since we are looking for minimum scores, we should aim to form subarrays whose score is 0 whenever possible. This is because any non-zero score would contribute to the sum, whereas a score of 0 would not.

Given this, the strategy is fairly straightforward:

- We start with a score set to -1 because -1 represents a series of all 1s in binary, ensuring when we perform the first bitwise AND operation with any element of the array, the result is the number itself.
- We iterate through the array and perform a bitwise AND operation on the current score and the current element to update the score.
- If at any point the score becomes 0, we know we can split the subarray at this point and start a new one because we've achieved the minimum possible score for a subarray.
- Each time we start a new subarray, we increment our answer (`ans`) which represents the maximum number of subarrays we can obtain.
- The reason we return `ans - 1` instead of `ans` at the end is to account for the initial subarray count we start with at the beginning.

This greedy and bitwise approach efficiently allows us to partition the array to achieve the minimum sum of scores, thus enabling us to find the maximum number of subarrays that fulfill the conditions.

Solution Approach

The solution provided follows a simple yet effective method to achieve the objective defined in the problem statement. Here's the walkthrough of the implementation:

1. We initialize a variable `score` and set it to `-1`. The choice of `-1` is strategic because, in binary, `-1` corresponds to an infinite sequence of `1`s. This means that when we take the bitwise AND of `-1` with any number, the result is the number itself.
2. The variable `ans` is used to maintain the count of subarrays created as part of the solution and is initially set to `1`. This represents the first subarray that will include at least the first element of the array.
3. We then iterate through each number in the `nums` array, updating the `score` with the bitwise AND of the current `score` and the current element. In code, this is `score &= num`. What this does is it progressively calculates the bitwise AND of the elements of the forming subarray until the score reaches `0`.
4. The `if` statement within the loop checks if the `score` is `0`. When `score` becomes `0`, we know that we can split the array at that point since we cannot further minimize the score of the current subarray. We do this by resetting the `score` to `-1` and incrementing `ans`, which is counting the number of subarrays.
5. Once we've processed all elements in the `nums` array, we check the value of `ans`. If `ans` is `1`, it implies that there was no point in the array where the score reached `0`, and thus the whole array is a single subarray, and we return `1`. Otherwise, we return `ans - 1` as during iteration, `ans` increment also includes the count for the last subarray which might not have been explicitly split in the iteration.

This implementation effectively uses a single pass of the array and does not require any additional data structures, making it very space-efficient. It leverages the bitwise operation to keep track of the ongoing score of the currently considered subarray and to decide when to split into a new subarray, based on the score reaching `0`.

Furthermore, the solution is greedy in nature. Greedy algorithms make the optimal choice at each step as they attempt to find the global optimum. In this case, splitting whenever a subarray reaches a score of `0` guarantees the minimum possible sum of scores, thereby aligning with the problem's requirement to minimize the sum while maximizing the number of subarrays.

Example Walkthrough

Let's consider a small example with an array `nums` given as `[6, 1, 8, 7, 8]` to illustrate the solution approach.

1. We start by setting `score` to `-1` since this will allow us to bitwise-AND with any number without affecting its value.
2. We'll also initialize `ans` to `1`, as we start considering the array as one whole subarray first.
3. Start iterating over `nums`:
   - First element `6`: We perform `score &= 6`, which results in `score` being `6` as `-1 & 6 = 6`.
   - Second element `1`: `score &= 1`, and now `score` is `0` because `6 & 1 = 0`. Since `score` is now `0`, we increment `ans` to `2` and reset `score` to `-1`.
   - Third element `8`: `score &= 8`, resulting in `score` being `8`.
   - Fourth element `7`: `score &= 7`, and `score` remains `0` because `8 & 7 = 0`. We increment `ans` to `3` and reset `score` to `-1`.
   - Fifth element `8`: `score &= 8`, resulting in `score` being `8`. The loop ends here.
4. Finally, since we've reached the end of the array, we check our subarray count `ans`. We have incremented `ans` two times, so its value is `3`. According to our approach, we return `ans - 1`, which is `3 - 1 = 2`.

Our walkthrough of the `[6, 1, 8, 7, 8]` example demonstrates that the input array can be split into `2` subarrays to minimize the sum of the bitwise AND scores. The subarrays would be `[6, 1]` and `[8, 7, 8]`. This example illustrates how the solution approach strategically breaks down the array into subarrays with minimized AND scores.

Solution Implementation

Python

```python
class Solution:
    def maxSubarrays(self, nums: List[int]) -> int:
        # Initialize the current bitwise AND score and the count of maximum subarrays
        current_and_score, max_subarrays_count = -1, 1

        # Iterate over each number in the nums list
        for num in nums:
            # Perform bitwise AND operation with the current number and store the result
            current_and_score &= num

            # If the current and score becomes 0, reset it to -1 and increment subarray count
            if current_and_score == 0:
                current_and_score = -1
                max_subarrays_count += 1

        # If only 1 subarray, return 1, otherwise return one less than the counted subarrays
        # because if there's more than one, the first doesn't count (starts with -1, but 0 resets it)
        return 1 if max_subarrays_count == 1 else max_subarrays_count - 1
```

Java

```java
class Solution {
    public int maxSubarrays(int[] nums) {
        // Initialize score with all bits set (-1 has all bits set in two's complement representation)
        int score = -1;
        // Initialize answer to 1 since we have at least one subarray by default
        int answer = 1;

        // Iterate through the array.
        for (int number : nums) {
            // Perform bitwise AND operation with each number and store the result in score.
            score &= number;

            // If the score becomes 0, increment the answer.
            // This implies we start a new subarray as per the given logic.
            if (score == 0) {
                answer++;
                // Reset the score to -1 to consider the next subarray.
                score = -1;
            }
        }
        // If we only found one subarray, return 1.
        // Otherwise, subtract 1 from answer because we incremented it one time too many
        // due to the last iteration possibly setting score to 0.
        return answer == 1 ? 1 : answer - 1;
    }
}
```

C++

```cpp
#include <vector>

class Solution {
public:
    // Function to calculate the maximum number of subarrays
    // with non-zero bitwise AND score.
    int maxSubarrays(vector<int>& nums) {
        // Initialize the bitwise AND score to -1 since -1 has
        // all bits set to 1, which will not affect the initial AND operation.
        int andScore = -1;

        // Initialize the count of subarrays to 1.
        int subarrayCount = 1;

        // Iterate over each number in the given vector.
        for (int num : nums) {
            // Perform bitwise AND operation between the current andScore and the number.
            andScore &= num;

            // Check if the current andScore has become 0, indicating that
            // a subarray with non-zero AND score has ended.
            if (andScore == 0) {
                // Reset the andScore for a new subarray by setting it
                // to -1 (all bits set to 1).
                andScore = -1;

                // Increment the count of subarrays.
                ++subarrayCount;
            }
        }

        // Since the initial count was set to 1, we need to subtract 1 if multiple
        // subarrays are identified. If only one subarray exists, return 1.
        return subarrayCount == 1 ? 1 : subarrayCount - 1;
    }
};
```

TypeScript

```typescript
function maxSubarrays(nums: number[]): number {
    // Initialize the variable 'answer' to keep the count of maximum subarrays.
    // Initialize the variable 'score' to keep track of the bitwise AND accumulation.
    let [answer, score] = [1, -1];

    // Iterate through each number in the nums array
    for (const num of nums) {
        // Apply bitwise AND operation between 'score' and 'num'.
        score &= num;

        // When 'score' becomes 0, reset it to -1 and increment the 'answer'.
        if (score === 0) {
            score = -1;
            answer++;
        }
    }

    // If answer is 1, it means we have not found any sequence that resets score
    // Hence, return 1. Otherwise, return 'answer' minus 1 since we started from 1.
    return answer === 1 ? 1 : answer - 1;
}
```

```python
class Solution:
    def maxSubarrays(self, nums: List[int]) -> int:
        # Initialize the current bitwise AND score and the count of maximum subarrays
        current_and_score, max_subarrays_count = -1, 1

        # Iterate over each number in the nums list
        for num in nums:
            # Perform bitwise AND operation with the current number and store the result
            current_and_score &= num

            # If the current and score becomes 0, reset it to -1 and increment subarray count
            if current_and_score == 0:
                current_and_score = -1
                max_subarrays_count += 1

        # If only 1 subarray, return 1, otherwise return one less than the counted subarrays
        # because if there's more than one, the first doesn't count (starts with -1, but 0 resets it)
        return 1 if max_subarrays_count == 1 else max_subarrays_count - 1
```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the array `nums`. This is because the code iterates through each element of the array exactly once with a single for-loop, performing constant time operations within the loop.

The space complexity of the code is `O(1)` which implies that the space required by the algorithm does not depend on the size of the input array. The variables `score` and `ans` use a fixed amount of space, and no additional data structures are dependent on the input size are used.