2436. Minimum Split Into Subarrays With GCD Greater Than One

Medium Greedy Array Math Dynamic Programming Number Theory Leetcode Link

Problem Description

The problem presents an array of positive integers and requires splitting this array into one or more disjoint subarrays. The split must be such that each element in the original array belongs to exactly one subarray. Additionally, for each subarray formed, the greatest common divisor (GCD) of its elements must be strictly greater than 1, meaning none of the subarrays should consist of elements that are only mutually prime. The objective is to find the minimum number of such subarrays that can be created following these rules.

Intuition

The intuition behind the solution is leveraged from the fact that the GCD of a subarray can only decrease or stay the same when more elements are added to it. Starting from the first element, we can try to extend the subarray as much as possible until the GCD becomes 1. When the GCD is 1, it means that adding any more elements would not allow us to maintain the condition that the GCD is greater than 1. Therefore, we start a new subarray beginning at that point.

The approach incrementally builds subarrays and keeps track of the current GCD. Whenever the GCD becomes 1, the subarray is finished, and a new one begins. The `gcd` function (presumably from `math` module or a similar implementation) calculates the GCD of two numbers and is repeatedly used to update the current GCD of each forming subarray. If at any point the GCD of the cumulative elements is 1, a split is made (indicated by incrementing the answer), and the GCD is reset to the value of the current element, thereby starting a new subarray. This process is repeated for all elements in the array, and the final answer denotes the minimum number of subarrays formed.

The reason the starting value of `ans` is 1 in the solution is because at least one subarray is always possible, given that all the integers are positive and the array cannot be empty.

Solution Approach

The solution uses a greedy approach, where we try to extend a subarray as much as possible before needing to create a new subarray so that the GCD of each subarray is greater than 1.

Here's a step by step breakdown of how the solution is implemented:

1. Initialize `ans`, the counter for minimum subarrays, to 1 because we can always form at least one subarray.
2. Initialize a variable `g` to 0, which will store the running GCD of the current subarray.
3. Iterate over each number `x` in the array `nums`.
   - Calculate the new GCD of the current subarray by taking the GCD of `g` (the GCD so far) and `x` (the current number). The GCD is updated using the statement `g = gcd(g, x)`.
   - If the GCD after adding `x` to the subarray becomes 1, then:
     - Increment `ans` by 1 as this signifies that a new subarray must start to fulfill the condition that each subarray must have a GCD greater than 1.
     - Also, reset `g` to the value of `x` because a new subarray is starting with `x` as its first element.
   - If the GCD does not become 1, it implies that `x` can be added to the current subarray without violating the condition, and we continue to the next iteration.
4. After iterating through all elements, return the value of `ans`.

The Python `gcd` function used in the code can be brought in by importing it from the `math` library (`from math import gcd`) or can be implemented if needed.

This algorithm does a single pass over the input array (`O(n)` time complexity, where `n` is the number of elements in the array) and only uses constant extra space (`O(1)` space complexity), making it efficient and suitable for large arrays as well.

By following this pattern, we can thus ensure that we always form subarrays with the maximum possible length without violating the GCD condition, which leads us to the minimum possible number of subarrays.

Example Walkthrough

Let's go through an example to illustrate the solution approach. Consider the array of positive integers `[12, 6, 9, 3, 5, 7]`.

Following the steps outlined in the solution approach:

1. Initialize `ans` to 1, as we can form at least one subarray.
2. Initialize `g` to 0, to keep track of the GCD of the current subarray.

Proceed with the array elements:

- For the first element `12`, we calculate the GCD of `g` and `12`. Since `g` is 0, the GCD is `12` (because the GCD of any number and 0 is the number itself). Now, `g` is updated to `12`.
- Moving to the second element `6`, the new GCD is the GCD of `12` and `6`, which is `6`. We update `g` to `6`.
- The third element is `9`. The GCD of `6` and `9` is `3`, so `g` is updated to `3`.
- Next is the number `3`. The GCD of `3` and `3` remains `3`, so we continue building this subarray.
- The fifth element is `5`, and here the GCD of `3` and `5` is `1`. Since the GCD has become `1`, we need to start a new subarray. We increment `ans` to 2 and reset `g` to `5` (the current element).
- The final element, `7`, has a GCD of `1` with `5` (since `5` and `7` are prime with respect to each other), which would again imply the start of a new subarray. We increment `ans` to 3 and set `g` to `7`.

Therefore, the minimum number of subarrays where each subarray has a GCD greater than 1 is `3`. These are `[12, 6, 9, 3]`, `[5]`, and `[7]`.

The example perfectly illustrates the efficiency and the greedy nature of the solution, where the algorithm tries to build the longest subarray possible before needing to start a new one due to encountering a GCD of `1`.

Python Solution

```python
from math import gcd
from typing import List

class Solution:
    def minimumSplits(self, nums: List[int]) -> int:
        # Initialize the variables:
        # 'split_count' to count the minimum splits needed
        # 'current_gcd' to keep track of the gcd of the current group
        split_count, current_gcd = 1, 0

        # Iterate over each number in the list
        for number in nums:
            # Calculate the gcd of the current group and the current number
            current_gcd = gcd(current_gcd, number)

            # When the gcd becomes 1, it's optimal to split here
            # because any next number can start a new group with gcd 1
            if current_gcd == 1:
                split_count += 1
                current_gcd = number  # Start a new group with the current number

        # Return the minimum number of splits needed
        return split_count

```

Java Solution

```java
class Solution {
    // Method to calculate the minimum number of splits in the array
    public int minimumSplits(int[] nums) {
        int answer = 1; // Start with a single split
        int currentGCD = 0; // Initialize GCD

        // Iterate through each number in the array
        for (int number : nums) {
            // Calculate the GCD of currentGCD and the current number
            currentGCD = gcd(currentGCD, number);

            // If the GCD is 1, a new split is required
            if (currentGCD == 1) {
                answer++; // Increase the number of splits
                currentGCD = number; // Reset the currentGCD to the current number
            }
        }
        // Return the total number of splits required
        return answer;
    }

    // Helper method to calculate the Greatest Common Divisor (GCD) of two numbers
    private int gcd(int a, int b) {
        // If second number is 0, the GCD is the first number
        return b == 0 ? a : gcd(b, a % b); // Recursively calculate gcd
    }
}

```

C++ Solution

```cpp
#include <vector> // Include the vector header for using the vector class

class Solution {
public:
    int minimumSplits(std::vector<int>& nums) {
        int splitsRequired = 1; // Start with a single split required
        int currentGcd = 0; // Initialize gcd to 0 representing an empty subsequence

        // Iterate over each number in the vector nums
        for (int num : nums) {
            // Update the current gcd to include the current number
            currentGcd = std::gcd(currentGcd, num);

            // If the gcd is 1, we need to start a new subsequence
            if (currentGcd == 1) {
                ++splitsRequired; // Increment the number of splits required
                currentGcd = num; // Start a new subsequence with the current number as the first element
            }
        }

        return splitsRequired; // Return the total splits required
    }
};

```

Typescript Solution

```typescript
// Calculate the minimum number of splits required
function minimumSplits(nums: number[]): number {
    let splits = 1; // Initialize splits count
    let currentGCD = 0; // Current greatest common divisor (GCD)

    // Iterate through each number in the array
    for (const num of nums) {
        currentGCD = gcd(currentGCD, num); // Update the GCD

        if (currentGCD == 1) {
            // If GCD is 1, increment split count and reset the current GCD
            splits++;
            currentGCD = num;
        }
    }

    // Return the total number of splits required
    return splits;
}

// Calculate the greatest common divisor of two numbers
function gcd(a: number, b: number): number {
    // If second number is zero, return the first number
    if (b === 0) {
        return a;
    }

    // Otherwise, continue the process recursively using Euclid's algorithm
    return gcd(b, a % b);
}

```

Time and Space Complexity

The provided Python code defines a function `minimumSplits` that calculates the minimum number of non-empty groups to split the input list `nums` such that the greatest common divisor (GCD) of all numbers in the same group is not equal to 1.

Time Complexity:

The time complexity of the code is determined by the number of iterations in the for loop and the complexity of the `gcd` function calls within the loop.

- The for loop runs once for each element in `nums`. If `n` is the number of elements in `nums`, the loop iterates `n` times.
- For each iteration, the GCD of the current running GCD `g` and the current element `x` is calculated using the `gcd` function. The time complexity of the `gcd` function is generally `O(log(min(a, b)))` where `a` and `b` are the inputs to the `gcd` function. In the worst case, `a` and `b` could be the last two elements of `nums`.

Since the GCD decreases or stays the same with each iteration, the time complexity is better than `O(n log m)` with `m` being the maximum element in `nums` due to the iterations where `g` is reduced to `1`, resetting the GCD calculation.

Overall, the worst-case time complexity of the `minimumSplits` function is `O(n log m)`.

Space Complexity:

The space complexity is determined by the additional space used by the function.

- The variables `ans` and `g` use constant space.
- Since Python's `gcd` function has no additional space that depends on the input size (assuming it doesn't use a recursive stack that depends on the size of the values), it can be considered to use constant space.
- No additional data structures are used that grow with the size of the input.

Thus, the space complexity is `O(1)` for constant extra space.