2584. Split the Array to Make Coprime Products

Hard Array Hash Table Math Number Theory Leetcode Link

Problem Description

In this problem, you're given an integer array `nums` of length `n`. The task is to find the smallest index `i` (where `0 <= i <= n - 2`) such that splitting the array into two parts at index `i` results in the product of the first `i + 1` elements being coprime with the product of the remaining elements. Two numbers are coprime if their greatest common divisor (GCD) is equal to 1.

A split is considered valid if and only if the aforementioned condition of coprimality is satisfied. If no such index exists, the function should return `-1`.

For example:

- With `nums = [4,7,8,15,3,5]`, a valid split occurs at index `i = 2` where the products are `4*7*8 = 224` and `15*3*5 = 225`, and `gcd(224, 225) = 1`.
- With `nums = [4,7,15,8,3,5]`, no valid split can be found, and the function should return `-1`.

Intuition

The general approach to solving this problem is to iterate over the `nums` array and at each index, we need to track the prime factors of the cumulative product of elements up to that index, as well as maintain a mapping of the last index at which each prime factor appears.

Here's a step-by-step breakdown on how we arrive at the solution:

1. **Prime Factorization for each element**: We iterate through the array and perform prime factorization for each element. This will help us in understanding the prime factors that make up each number in the array.

2. **Tracking First and Last Occurrences**: While doing the prime factorization, we maintain two data structures. One dictionary (`first`) to store the first occurrence of a prime factor and another list (`last`) to note down the last occurrence of a prime factor that has been seen so far. If a prime factor appears again at a later index, we update the `last` list at the index of the first occurrence of this prime factor to the current index.

3. **Finding the Split Index**: After completing the prime factorization and tracking, we iterate over the `last` list to find the smallest index `i` where `last[i] < i`. This indicates that splitting the array at index `i` will have no common prime factors for the product of the first `i + 1` numbers and the product of the remaining numbers.

The index `mx` is used to track the maximum index in `last` that we have seen so far. If at any point `mx < i`, we immediately return `mx` as the answer - this is our split point. If we finish iterating over the list and don't find such an index, we return `-1`, which means no valid split exists.

This solution is efficient since it avoids explicitly calculating the product of the subarrays, which could result in very large numbers, and it allows us to determine if the array can be split validly based on the presence of common prime factors between the two subarrays.

Solution Approach

The solution to this problem uses prime factorization and dictionary mapping to efficiently find the smallest valid split index.

Here's how the implementation works:

- **Prime Factorization**: For each number `x` in the array `nums`, the algorithm finds its prime factors. Starting with the smallest prime number `2`, the code iterates through potential factors increasing `j` until `j` exceeds the square root of `x`. If `x` is divisible by `j`, it is repeatedly divided by `j` until it is no longer divisible. In this way, each element `x` is broken down into its prime factors.

- **Dictionary Mapping (first occurrence)**: The `first` dictionary keeps track of the first occurrence of each prime factor throughout the array. When a new prime factor is encountered, it's added to the dictionary with the current index as its value.

- **List Tracking (last occurrence)**: Simultaneously, the list `last` is updated to track the last occurrence of a previously seen prime factor. If a prime factor that's already in the `first` dictionary is found again, the `last` array at the index of the first occurrence of this factor (obtained from the `first` dictionary) is updated to the current index.

- **Finding the Valid Split**: After populating the `first` and `last` data structures, the algorithm searches for the smallest index `i` at which the split can occur. To keep track of the split index, the algorithm uses the variable `mx` to store the maximum value found so far in `last`. It iterates through the `last` array while updating `mx` to the current maximum index. If at any index `i`, `mx` is found to be less than `i`, it means that the common prime factors (if any) of `nums[0] to nums[i]` and `nums[i+1] to nums[n-1]` are only found before index `i`. Hence, `mx` is the valid smallest split index, and it is returned.

- **Returning the Result**: The iteration continues until the end of the list. If no valid split is found, the function returns `-1`.

The algorithm is efficient because it avoids the need to calculate the actual products of the subarrays, which can be very large and could lead to integer overflow. Instead, it relies on the presence of common prime factors to determine whether a split is valid. The use of a dictionary and list to track the first and last occurrence of prime factors allows for fast lookups and updates, resulting in an efficient solution.

The time complexity for this algorithm can be approximated as O(n * sqrt(m)), where `n` is the length of the array, and `m` is the maximum value in the array, since for each element we might need to iterate up to its square root to find all prime factors.

Example Walkthrough

Let's consider a small example where `nums = [6, 5, 10]`:

1. Initialize `first` dictionary and `last` array:

```
first = {}
last = [-1, -1, -1]  // -1 indicates that we haven't seen the prime factor yet
```

2. Perform prime factorization on each number in `nums` and update `first` and `last`:

- For `nums[0] = 6`, its prime factors are 2 and 3.

```
first = {2: 0, 3: 0}
last = [0, 0, -1]  // We've seen 2 and 3 at index 0
```

- For `nums[1] = 5`, its only prime factor is 5.

```
first = {2: 0, 3: 0, 5: 1}
last = [0, 0, -1]  // No need to update as 5 is a new prime factor
```

- For `nums[2] = 10`, its prime factors are 2 and 5. Factor 2 has been encountered before, so we update its last occurrence. Factor 5 has also been encountered before, but since its first and last occurrence is at the same index, there is no need to update.

```
first = {2: 0, 3: 0, 5: 1}
last = [2, 0, -1]  // Update last occurrence of 2 to index 2 where it's seen again
```

3. Search for the smallest index `i` as a valid split:

- Iterate through `last`. We use `mx` to store the maximum index found so far.
  - At `i = 0`, `last[0] = 2`, so `mx = 2`.
  - At `i = 1`, `last[1] = 0`, but `mx` is still 2 (`mx` remains the max of `last[0]` and `last[1]`).
- We encounter `i < mx` at `i = 1`. This means that all prime factors seen in the first `i+1` elements (up to index 1) and all following elements (from index 2) are unique.

Thus, the valid split index found is `i = 1`.

The algorithm concludes that the products of the first half `6 * 5` and the second half `10` do not have any common prime factors, so they are coprime. The smallest valid split index returned is `1`.

Python Solution

```python
from typing import List

class Solution:
    def findValidSplit(self, nums: List[int]) -> int:
        # Dictionary to keep track of the first occurrence index of each prime factor
        first_occurrence = {}
        n = len(nums)

        # List to keep track of the last occurrence index that can be reached
        # from the first occurrence of each prime factor.
        last_reachable = list(range(n))

        # Iterate over the list of numbers to update first and last occurrence indices
        for index, number in enumerate(nums):
            factor = 2
            # Factorize the number and update occurrences
            while factor <= number // factor:
                if number % factor == 0:
                    if factor in first_occurrence:
                        last_reachable[first_occurrence[factor]] = index
                    else:
                        first_occurrence[factor] = index
                    while number % factor == 0:
                        number //= factor
                factor += 1

            # Check for a prime number greater than 1 and update occurrences
            if number > 1:
                if number in first_occurrence:
                    last_reachable[first_occurrence[number]] = index
                else:
                    first_occurrence[number] = index

        # This will hold the maximum last occurrence index that we can reach
        max_reach = last_reachable[0]

        # Iterate over the last reachable list to find the valid split point
        for index, max_index_for_current_factor in enumerate(last_reachable):
            # If we find an index larger than the current maxValue, return the maximum
            # last occurrence index that we have, as this is the valid split point
            if max_reach < index:
                return max_reach
            max_reach = max(max_reach, max_index_for_current_factor)

        # If we couldn't find a valid split point, return -1
        return -1

```

Java Solution

```java
class Solution {
    public int findValidSplit(int[] nums) {
        // Map to track the first occurrence of each prime factor
        Map<Integer, Integer> firstOccurrence = new HashMap<>();
        int length = nums.length;
        // Array to track the last occurrence where each prime factor can be found
        int[] lastOccurrence = new int[length];

        // Initialize lastOccurrence to the current index for each element
        for (int i = 0; i < length; ++i) {
            lastOccurrence[i] = i;
        }

        // Iterate over the array to update the first and last occurrences of each prime factor
        for (int i = 0; i < length; ++i) {
            int x = nums[i];
            // Factorization by trial division
            for (int j = 2; j <= x / j; ++j) {
                if (x % j == 0) {
                    // If this factor has been seen before, update its last occurrence
                    if (firstOccurrence.containsKey(j)) {
                        lastOccurrence[firstOccurrence.get(j)] = i;
                    } else { // Otherwise, record its first occurrence
                        firstOccurrence.put(j, i);
                    }
                    // Remove this prime factor from x
                    while (x % j == 0) {
                        x /= j;
                    }
                }
            }
            // Check for the last prime factor which can be larger than sqrt(x)
            if (x > 1) {
                if (firstOccurrence.containsKey(x)) {
                    lastOccurrence[firstOccurrence.get(x)] = i;
                } else {
                    firstOccurrence.put(x, i);
                }
            }
        }

        // Variable to track the maximum last occurrence index found so far
        int maxLastOccurrence = lastOccurrence[0];

        // Iterate over the array to find a valid split
        for (int i = 0; i < length; ++i) {
            // If the current maximum last occurrence is before the current index, we found a valid split
            if (maxLastOccurrence < i) {
                return maxLastOccurrence;
            }
            // Update the maximum last occurrence
            maxLastOccurrence = Math.max(maxLastOccurrence, lastOccurrence[i]);
        }

        // If no valid split is found, return -1
        return -1;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>
#include <numeric> // for iota function
using namespace std;

class Solution {
public:
    // Method to find a valid split in the vector nums
    int findValidSplit(vector<int>& nums) {
        // Map to keep track of first occurrences of prime factors
        unordered_map<int, int> firstOccurrence;
        int n = nums.size();

        // Vector to keep track of the latest index where each prime appears
        vector<int> lastOccurrence(n);
        iota(lastOccurrence.begin(), lastOccurrence.end(), 0); // Initialize with indices

        // Iterate over all elements to populate first and last occurrence information
        for (int i = 0; i < n; ++i) {
            int x = nums[i];
            // Factorize the current number x using trial division
            for (int j = 2; j <= x / j; ++j) {
                if (x % j == 0) { // If j is a factor of x
                    // Update the occurrences for the prime factors
                    if (firstOccurrence.count(j)) {
                        lastOccurrence[firstOccurrence[j]] = i;
                    } else {
                        firstOccurrence[j] = i;
                    }
                    // Divide x by j as long as j is a factor
                    while (x % j == 0) {
                        x /= j;
                    }
                }
            }
            // If there are any prime factors left, handle the remaining prime factor
            if (x > 1) {
                if (firstOccurrence.count(x)) {
                    lastOccurrence[firstOccurrence[x]] = i;
                } else {
                    firstOccurrence[x] = i;
                }
            }
        }

        // Initialize the max last occurrence seen so far
        int maxLastOccurrence = lastOccurrence[0];

        // Iterate to determine the earliest valid split point
        for (int i = 0; i < n; ++i) {
            if (maxLastOccurrence < i) {
                return maxLastOccurrence; // This is a valid split point
            }
            maxLastOccurrence = max(maxLastOccurrence, lastOccurrence[i]);
        }

        // If no split point found, return -1
        return -1;
    }
};

```

Typescript Solution

```typescript
// Importing Map from the standard library to use as a hashmap
import { max, min } from "lodash";

// Function to find a valid split in the array nums
function findValidSplit(nums: number[]): number {
    // Map to keep track of the first occurrences of prime factors
    let firstOccurrence: Map<number, number> = new Map();

    // Array to keep track of the latest index where each prime appears
    let lastOccurrence: number[] = Array.from(nums.keys());

    // Iterate over all elements to populate first and last occurrence information
    for (let i = 0; i < nums.length; ++i) {
        let x = nums[i];

        // Factorize the current number x using trial division
        for (let j = 2; j <= Math.sqrt(x); ++j) {
            if (x % j === 0) { // If j is a factor of x
                // Update the occurrences for the prime factors
                if (firstOccurrence.has(j)) {
                    lastOccurrence[firstOccurrence.get(j) as number] = i;
                } else {
                    firstOccurrence.set(j, i);
                }
                // Divide x by j as long as j is a factor
                while (x % j === 0) {
                    x /= j;
                }
            }
        }

        // If there are any prime factors left, handle the remaining prime factor
        if (x > 1) {
            if (firstOccurrence.has(x)) {
                lastOccurrence[firstOccurrence.get(x) as number] = i;
            } else {
                firstOccurrence.set(x, i);
            }
        }
    }

    // Initialize the max last occurrence seen so far
    let maxLastOccurrence: number = lastOccurrence[0];

    // Iterate to determine the earliest valid split point
    for (let i = 0; i < nums.length; ++i) {
        maxLastOccurrence = max([maxLastOccurrence, lastOccurrence[i]]) as number;
        if (maxLastOccurrence < i) {
            return maxLastOccurrence; // This is a valid split point
        }
    }

    // If no split point is found, return -1
    return -1;
}

```

Time and Space Complexity

The provided code block has two main parts to consider when analyzing the time complexity:

1. The first loop where we iterate over `nums` and factorize each number, updating `first` and `last` based on the factors.
2. The second loop where we iterate over `last` to find the valid split.

For the first part, in the worst case, the factorization of each number can take `O(sqrt(x))` time where `x` is the value of the number being factorized. Since we are doing this for every number in `nums`, and with `n` being the size of `nums`, the total time complexity for this part is `O(n * sqrt(max(nums)))`.

The second part is a linear scan over the array `last`, having a time complexity of `O(n)`.

Therefore, the overall time complexity is `O(n * sqrt(max(nums))) + O(n)` which simplifies to `O(n * sqrt(max(nums)))` as `sqrt(max(nums))` is the dominating factor.

The space complexity of the code is affected by the `first` and `last` data structures, which hold at most `n` elements, giving us a space complexity of `O(n)`.