2963. Count the Number of Good Partitions

Hard Array Hash Table Math Combinatorics

Problem Description

You have an array `nums` with positive integers and the challenge is to find out how many ways you can partition this array into contiguous subarrays so that no subarray contains the same number more than once. If you imagine a case with repeated numbers, you'd want to keep repeats in the same subarray to avoid violating the condition. The total number of ways of partitioning the array should be given modulo `10^9 + 7` to keep the number manageable size-wise, as it can get very large.

Intuition

The main idea is to group the same numbers together in a subarray because of the requirement that no subarray should contain duplicate numbers. So, we start by recording the last occurrence of each number using a hash table. With this information, we can determine the potential ends of each subarray.

We iterate through the `nums` array, updating where the current group could end (this is the furthest last occurrence of any number encountered so far in the array). If the current index aligns with the furthest last occurrence, it means we've reached the end of a potential subarray partition.

Each partition gives us two choices moving forward - either continue with another subarray or group the next number into the current subarray. Except for the first number, every other number has this choice, naturally leading to a calculation of `2^(number of subarrays - 1)` to find all the different ways we can partition the array into subarrays that meet the requirements. Since the result can be very large, we use modular exponentiation to give the answer modulo `10^9 + 7`. This technique is efficient and proficient for dealing with large powers in a modular space.

Solution Approach

In the given solution, we utilize a hash map called `last` to keep track of the last index where each number appears in the array `nums`. This directly relates to the problem statement where we need to ensure that each subarray has unique elements. By knowing the last occurrence, we can determine the maximum bounds of where a unique element's subarray could end.

To implement this approach, we proceed as follows:

- Initialize the `last` hash map by iterating over the array `nums` and setting `last[x]` to `i` (the current index) for each element `x`. This way, after the iteration, each key in `last` directly maps to the last position of that key in `nums`.
- The variables `j` and `k` are used, where `j` keeps track of the end of the current subarray (initially `-1`) and `k` keeps count of the total number of subarrays that can be created (initially `0`).
- As we iterate through `nums`, for each number `x` at index `i`, we update the current subarray end `j` to the maximum of `j` and the last occurrence of `x` from the `last` hash map. If `j` equals `i` at any point during the iteration, it signifies that the current position can be the end of a subarray, at which point we increment `k` by 1.
- After completing the iteration over the array, we calculate the total number of good partitions. Each subarray provides a choice to either split or not split at its end (except for the first element of 'nums' which provides no such choice). Hence, for `k` subarrays, there are `k - 1` split choices, leading to a total of `2^(k - 1)` ways to partition the array.
- The answer is potentially very large, so we use the modulo operation with `10^9 + 7` to keep the numbers in a reasonable range. In Python, the power function `pow` is used with three arguments `pow(2, k - 1, mod)`, which efficiently computes `2^(k - 1)` modulo `mod` using fast exponentiation.

This algorithm uses constant space for variables and `O(n)` space for the hash table, where `n` is the length of `nums`. The time complexity is `O(n)` because we pass through the array only twice: once for creating the hash table and again for determining the partition points and counting subarrays.

Example Walkthrough

Let's say our array `nums` is `[4, 3, 2, 1, 4, 3]`.

Following the solution approach outlined in the content provided:

1. **Initialize the `last` hash map:**

   We iterate over `nums` and update the `last` hash map:

   ```
   last[4] = 4 (index of the last 4)
   last[3] = 5 (index of the last 3)
   last[2] = 2 (index of the last 2)
   last[1] = 3 (index of the last 1)
   ```

2. **Set up variables for tracking subarray ends and count:**

   We have variable `j` initialized to `-1` and `k` initialized to `0`.

3. **Iterate through `nums` to determine subarrays:**

   As we pass through `nums`, we will update `j` and `k` accordingly:

   - `i = 0` (value `4`) Set `j = max(j, last[4])` → `j = 4`. (No end of subarray reached)
   - `i = 1` (value `3`) Set `j = max(j, last[3])` → `j = 5`. (No end of subarray reached)
   - `i = 2` (value `2`) Keep `j = 5`. (No end of subarray reached)
   - `i = 3` (value `1`) Keep `j = 5`. (No end of subarray reached)
   - `i = 4` (value `4`) We are at the last occurrence of '4', and `i` matches `j`. We've reached the end of a subarray therefore increment `k` (now `k = 1`).
   - `i = 5` (value `3`) Again, `i` matches `j`, marking the end of another subarray so increment `k` to `2`.

4. **Count the number of ways to partition `nums`:**

   Since we have two subarrays, there are `k - 1 = 1` decision point on whether or not to split them. There are (2^(k-1) = 2^1 = 2) ways to partition our `nums` array.

5. **Apply modulo operation:**

   We calculate the total number of partition ways modulo (`10^9 + 7`): `result = pow(2, k - 1, 10^9 + 7)` → `result = 2`.

Thus, there are two ways to partition the array `[4, 3, 2, 1, 4, 3]` into contiguous subarrays where no subarray contains the same number more than once.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def numberOfGoodPartitions(self, nums: List[int]) -> int:
        # Store the last occurrence index of each number in the list
        last_occurrence_index = {value: index for index, value in enumerate(nums)}
        mod = 10**9 + 7  # Define the modulo value

        partition_end = -1  # Initialize the end position of the current partition
        good_partitions_count = 0  # Counter for the number of good partitions

        # Loop through the numbers in the list and their corresponding indices
        for index, value in enumerate(nums):
            # Update the end position of the current partition to be the maximum
            # between the current end and the last occurrence index of the number
            partition_end = max(partition_end, last_occurrence_index[value])

            # If the current index is equal to the partition end, it signifies the end
            # of a partition and we found a "good partition"
            # Increment the good partitions count accordingly
            if index == partition_end:
                good_partitions_count += 1

        # The total number of combinations is 2^(good_partitions_count - 1),
        # and we return this number modulo mod
        return pow(2, good_partitions_count - 1, mod)
```

Java

```java
class Solution {
    public int numberOfGoodPartitions(int[] nums) {
        // Use a map to store the last occurrence of each number
        Map<Integer, Integer> lastOccurrence = new HashMap<>();
        int length = nums.length;

        // Populate the map with the last occurrence of each number
        for (int i = 0; i < length; ++i) {
            lastOccurrence.put(nums[i], i);
        }

        // Define the modulus for large prime numbers, as per the problem statement
        final int modulus = (int) 1e9 + 7;

        // Initialization of pointers for the partitioning logic
        int maxLastOccurrenceIndex = -1;
        int partitionCount = 0;

        // Iterate through the array to find good partitions
        for (int i = 0; i < length; ++i) {
            // Update the maxLastOccurrenceIndex to the maximum of current and last occurrence of nums[i]
            maxLastOccurrenceIndex = Math.max(maxLastOccurrenceIndex, lastOccurrence.get(nums[i]));

            // If the current index is equal to the maxLastOccurrenceIndex,
            // the partition can end here, so increment the partition counter
            partitionCount += i == maxLastOccurrenceIndex ? 1 : 0;
        }

        // Calculate the power (2^(partitionCount - 1) mod modulus) using quick exponentiation
        return quickPower(2, partitionCount - 1, modulus);
    }

    // Method to perform quick exponentiation (a^n % mod)
    private int quickPower(long base, int exponent, int mod) {
        long result = 1;
        for (; exponent > 0; exponent >>= 1) {
            // If the exponent's least significant bit is 1, multiply the result by base
            if ((exponent & 1) == 1) {
                result = result * base % mod;
            }
            // Square the base and take modulus
            base = base * base % mod;
        }
        return (int) result;
    }
}
```

C++

```cpp
#include <vector>
#include <unordered_map>

class Solution {
public:
    // Function to count the number of good partitions in the vector `nums`
    int numberOfGoodPartitions(std::vector<int>& nums) {
        std::unordered_map<int, int> lastIndex;  // Create a map to store the last index of each number.
        int n = nums.size();                     // Get the size of the input vector.
        // Fill the `lastIndex` map with the last index at which each number appears.
        for (int i = 0; i < n; ++i) {
            lastIndex[nums[i]] = i;
        }

        const int MOD = 1e9 + 7;                 // Initialize the modulus for the result.
        int i = -1;                              // Initialize the variable `i` to keep track of the last index of the current partition.
        int partitionsCount = 0;                 // Initialize the counter for partitions.

        // Iterate over the numbers and count the partitions.
        for (int j = 0; j < n; ++j) {
            i = std::max(i, lastIndex[nums[j]]);  // Update `i` to be the maximum of itself and the last index of current number.
            // Increase partitions count if `i` and `j` match.
            partitionsCount += i == j;
        }

        // Define a lambda function to calculate quick power modulo.
        auto quickPower = [&](long long base, int exponent, int mod) -> int {
            long long result = 1;
            for (; exponent; exponent >>= 1) {
                // If current exponent bit is 1, multiply result by base modulo MOD.
                if (exponent & 1) {
                    result = (result * base) % mod;
                }
                // Square the base modulo MOD for the next iteration.
                base = (base * base) % mod;
            }
            return static_cast<int>(result);
        };

        // Return the number of ways to partition the array, which is 2^(partitionsCount-1) modulo MOD.
        return quickPower(2, partitionsCount - 1, MOD);
    }
};
```

TypeScript

```typescript
function numberOfGoodPartitions(nums: number[]): number {
    // Helper function to calculate (a^b) % mod efficiently using binary exponentiation.
    const quickPower = (base: number, exponent: number, mod: number): number => {
        let result = 1;
        base %= mod;

        // Iterate over bits of the exponent
        while (exponent > 0) {
            // If the current bit is set, multiply the result with the base
            if (exponent & 1) {
                result = Number((BigInt(result) * BigInt(base)) % BigInt(mod));
            }

            // Square the base
            base = Number((BigInt(base) * BigInt(base)) % BigInt(mod));

            // Shift exponent to the right by 1 bit to check the next bit
            exponent >>= 1;
        }

        return result;
    };

    // Map to track the last position for each distinct number in the array
    const lastPositionMap: Map<number, number> = new Map();

    // Get the length of the input array
    const arrayLength = nums.length;

    // Populate last position map with the index of the last occurrence of each number
    for (let i = 0; i < arrayLength; ++i) {
        lastPositionMap.set(nums[i], i);
    }

    // Define the modulus value for the answer
    const modValue = 1e9 + 7;

    let maxLastPosition = -1; // Tracks the max last position of elements seen so far
    let partitionCount = 0;   // Counts the number of good partitions

    // Iterate through the array to determine the number of good partitions
    for (let i = 0; i < arrayLength; ++i) {
        // Update the maxLastPosition with the larger of current or last occurrence of nums[i]
        maxLastPosition = Math.max(maxLastPosition, lastPositionMap.get(nums[i])!);

        // If the current index i is the last occurrence of nums[i], increment partition count
        if (i === maxLastPosition) {
            partitionCount++;
        }
    }

    // Since the first partition doesn't count as splitting, compute 2^(partitionCount-1) % mod
    return quickPower(2, partitionCount - 1, modValue);
}
```

```python
from typing import List

class Solution:
    def numberOfGoodPartitions(self, nums: List[int]) -> int:
        # Store the last occurrence index of each number in the list
        last_occurrence_index = {value: index for index, value in enumerate(nums)}
        mod = 10**9 + 7  # Define the modulo value

        partition_end = -1  # Initialize the end position of the current partition
        good_partitions_count = 0  # Counter for the number of good partitions

        # Loop through the numbers in the list and their corresponding indices
        for index, value in enumerate(nums):
            # Update the end position of the current partition to be the maximum
            # between the current end and the last occurrence index of the number
            partition_end = max(partition_end, last_occurrence_index[value])

            # If the current index is equal to the partition end, it signifies the end
            # of a partition and we found a "good partition"
            # Increment the good partitions count accordingly
            if index == partition_end:
                good_partitions_count += 1

        # The total number of combinations is 2^(good_partitions_count - 1),
        # and we return this number modulo mod
        return pow(2, good_partitions_count - 1, mod)
```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)` since it iterates through all `n` elements of the input list `nums` just once. The loop builds a dictionary `last` that records the last occurrence of each element, and then it iterates over `nums` once more, updating `j` and `k`. Since both operations within the loop are constant time operations, the total time complexity remains linear with respect to the length of `nums`.

Space Complexity

The space complexity is also `O(n)` due to the storage requirements of the dictionary `last`, which, in the worst-case scenario, stores an entry for each unique element in `nums`. Since `nums` has `n` elements and all could be unique, the space required for `last` can grow linearly with `n`. No other data structures in the algorithm scale with the size of the input.