697. Degree of an Array

Easy Array Hash Table

Leetcode Link

Problem Description

Given an array of non-negative integers `nums`, the task is to find the smallest length of a contiguous subarray that has the same degree as the entire array. The degree is defined as the maximum frequency of any one element in the array. For instance, if the number 2 appears five times in the array and no other number appears as frequently, then the degree of the array is 5. By finding a subarray with this degree, we're essentially looking for the shortest segment of the array where the most frequent element in the entire array appears its maximum number of times.

Intuition

The core idea behind the solution is to track the positions (indices) of each element in the array while also counting the occurrence (frequency) of each element. Since we need the subarray with the degree of the original array, we focus on elements that match the highest frequency.

Here's a step-by-step explanation of the thought process:

1. First, count the occurrences of each element in the array using a `Counter` object. This will help us establish the degree of the array, which is the highest frequency of any element.
2. Determine the degree by looking at the frequency of the most common element in the `Counter`.
3. Two dictionaries, `left` and `right`, are used to store the first and last occurrences (indices) of each element in the array.
4. Next, we iterate over each element in the array, updating the `left` dictionary with the index of the first occurrence, and the `right` dictionary with the index of the most recent occurrence of each element.
5. Initialize an answer variable `ans` with infinity (`inf`). This variable will ultimately contain the smallest length of the required subarray.
6. Iterate over the elements again, focusing on those with a frequency equal to the degree. Calculate the length of the subarray for these elements using their first and last occurrence indices.
7. Update `ans` with the smallest length found.
8. Once all elements have been processed, `ans` will have the length of the shortest subarray that has the degree of the original array.

With these steps, we efficiently find the shortest subarray by utilizing the frequency and positional information of elements and avoid unnecessary computations.

Solution Approach

The implementation of the solution can be deconstructed into several parts to understand how it efficiently solves the problem using algorithms and data structures.

1. **Counter Data Structure**: At the beginning of the solution, we use Python's `Counter` from the `collections` module to tally up the occurrences of each element. The `Counter` object `cnt` will count the frequency of each number in `nums`.

```
cnt = Counter(nums)
```

2. **Calculating the Degree**: Once we have the count of each number, we can easily determine the degree of the original array, which is simply the highest frequency found in `cnt`.

```
degree = cnt.most_common(1)[0][1]
```

3. **Tracking Indices with Dictionaries**: We make use of two dictionaries, `left` and `right`. The `left` dictionary stores the first occurrence index of each number, and the `right` dictionary keeps track of the last occurrence index. This step is crucial for understanding the range of each element.

```
left, right = {}, {}
for i, x in enumerate(nums):
    if x not in left:
        left[x] = i
    right[x] = i
```

4. **Finding the Shortest Subarray**: We initialize `ans` to infinity (`inf`) to ensure any valid subarray length found will be smaller.

```
ans = inf
```

We iterate over each key value `x` in the `Counter`:

```
for x in nums:
```

For those numbers whose count equals the degree, we calculate the length `t` as the difference between their last and first occurrence indices plus one.

```
if cnt[x] == degree:
    t = right[x] - left[x] + 1
```

We then compare `t` with `ans` to determine if we've found a smaller subarray. If `t` is smaller, we update `ans`.

```
if ans > t:
    ans = t
```

5. **Returning the Answer**: After the loop finishes, `ans` contains the length of the smallest possible subarray with the same degree as `nums`, and we return it.

```
return ans
```

This solution effectively leverages hash maps (dictionaries) to track the indices of each element while using the `Counter` to determine their frequencies. By doing so, it provides an O(n) time complexity solution where `n` is the number of elements in `nums`, because each element is processed a constant number of times.

Example Walkthrough

To illustrate the solution approach, let's walk through an example:

Suppose we have the following array `nums`:

```
nums = [1, 2, 2, 3, 1]
```

Firstly, we use `Counter` from the `collections` module to count the occurrences of each number in `nums`. The `Counter` object `cnt` is as follows:

```
cnt = Counter(nums)  # cnt = {1:2, 2:2, 3:1}
```

Next, we determine the degree of the array, which is the maximum count for any number in `cnt`. In our example:

```
degree = cnt.most_common(1)[0][1]  # degree = 2, since both 1 and 2 appear twice.
```

Now, we need to keep track of the first and last occurrences of each element. We use two dictionaries, `left` and `right`, to store these indices:

```
left = {}
right = {}
for i, x in enumerate(nums):
    if x not in left:
        left[x] = i  # left = {1:0, 2:1, 3:3}
    right[x] = i  # right = {1:4, 2:2, 3:3}
```

We initialize `ans` to infinity (`inf`) to ensure we can easily find the minimum subarray length:

```
ans = inf
```

We iterate over each number `x` in `nums`, and for those numbers whose count equals the degree, we compute the subarray length `t`:

```
# # Pseudo-code iteration
for x in nums:
    if cnt[x] == degree:
        t = right[x] - left[x] + 1

        # For number 1: t = right[1] - left[1] + 1 = 4 - 0 + 1 = 5
        # For number 2: t = right[2] - left[2] + 1 = 2 - 1 + 1 = 2
```

After we calculate `t` for both 1 and 2, we find that the subarray [2, 2] is the shortest subarray with the degree of the array:

```
# # Pseudo-code comparison and update
if ans > t:
    ans = t  # ans = min(inf, 5) = 5 for number 1, then ans = min(5, 2) = 2 for number 2

return ans  # Returns 2
```

Therefore, the smallest length of a contiguous subarray that has the same degree as the entire array is 2, which corresponds to the subarray `[2, 2]`, and we return this result:

This walkthrough demonstrates how the solution utilizes `Counter` to find the degree and uses dictionaries to find the range of each element. The subarray length is then computed and minimized to find the shortest subarray with the degree of the original array.

Python Solution

```
from collections import Counter
from typing import List

class Solution:
    def findShortestSubArray(self, nums: List[int]) -> int:
        # Count the frequency of each element in nums
        cnt = Counter(nums)
        # Find the degree of the array (the maximum frequency of any element)
        degree = max(freq for num, freq in cnt.items())

        # Initialize dictionaries to store the first and last positions of each element
        first_position = {}
        last_position = {}

        # Loop through the list and populate first_position and last_position
        for index, value in enumerate(nums):
            if value not in first_position:  # Record the first occurrence if not already done
                first_position[value] = index
            last_position[value] = index  # Always update to the last occurrence

        # Initialize answer with infinity (act as a very large number)
        shortest_subarray_length = float('inf')

        # Find the shortest subarray length that has the same degree
        for element in frequency.keys():
            if frequency[element] == degree:
                current_length = last_position[element] - first_position[element] + 1
                if shortest_subarray_length > current_length:
                    shortest_subarray_length = current_length

        # Return the length of the shortest subarray with the same degree as the entire array
        return shortest_subarray_length
```

Java Solution

```
class Solution {
    public int findShortestSubArray(int[] nums) {
        // HashMap to store the frequencies of numbers
        HashMap<Integer, Integer> count = new HashMap<>();
        // HashMap to store the first index at which a number appears
        HashMap<Integer, Integer> firstIndex = new HashMap<>();
        // HashMap to store the last index at which a number appears
        HashMap<Integer, Integer> lastIndex = new HashMap<>();

        // Variable to keep track of the degree of the array
        int degree = 0;

        // Iterate through the array to populate the maps and find the maximum degree
        for (int i = 0; i < nums.length; ++i) {
            int num = nums[i];
            count.put(num, count.getOrDefault(num, 0) + 1);
            degree = Math.max(degree, count.get(num));

            // Only set the firstIndex for the number if it hasn't been set before
            if (!firstIndex.containsKey(num)) {
                firstIndex.put(num, i);
            }

            // Always update the lastIndex when the number is encountered
            lastIndex.put(num, i);
        }

        // Variable to store the length of the shortest subarray with the same degree as the whole array
        int shortestSubarrayLength = Integer.MAX_VALUE;

        // Iterate over the array to find the shortest subarray length
        for (int number : nums) {
            if (count.get(number) == degree) { // If the current number contributes to the array degree
                int tempLength = lastIndex.get(number) - firstIndex.get(number) + 1; // Calculate subarray length
                if (shortestSubarrayLength > tempLength) { // Update shortest length if smaller subarray found
                    shortestSubarrayLength = tempLength;
                }
            }
        }

        // Return the shortest subarray length with degree equal to that of the whole array
        return shortestSubarrayLength;
    }
}
```

C++ Solution

```
#include <vector>
#include <unordered_map>
using namespace std;

class Solution {
public:
    int findShortestSubArray(vector<int>& nums) {
        // Maps to keep track of the frequency, first occurrence, and last occurrence of elements.
        unordered_map<int, int> count;
        unordered_map<int, int> firstOccurrence;
        unordered_map<int, int> lastOccurrence;

        // Degree of the array, which is the maximum frequency of any element.
        int degree = 0;

        // Iterate through the array to populate the maps.
        for (int i = 0; i < nums.size(); ++i) {
            int value = nums[i];

            // Increase the count for this value and update the degree.
            degree = max(degree, ++count[value]);

            // Record the first occurrence of this value.
            if (firstOccurrence.count(value) == 0) {
                firstOccurrence[value] = i;
            }

            // Update the last occurrence index of this value.
            lastOccurrence[value] = i;
        }

        // Initialize the answer as a large number.
        int minLength = INT_MAX;

        // Loop through the array again to find the shortest subarray with the same degree.
        for (const auto& [value, freq] : count) {
            int value = kv.first;
            if (count[value] == degree) {
                // Calculate the length of the subarray for this value.
                int length = lastOccurrence[value] - firstOccurrence[value] + 1;

                // Update the answer if this length is smaller.
                minLength = min(minLength, length);
            }
        }

        // Return the minimum length found.
        return minLength;
    }
};
```

Typescript Solution

```
// Import statement not required in TypeScript for built-in types.

// Function to find the smallest subarray length that has the same degree as the entire array.
function findShortestSubArray(nums: number[]): number {
    // Object to keep track of the frequency of elements.
    const count: Record<number, number> = {};
    // Object to keep track of the first occurrence index of each element.
    const firstOccurrence: Record<number, number> = {};
    // Object to keep track of the last occurrence index of each element.
    const lastOccurrence: Record<number, number> = {};

    // Variable to store the degree of the array (maximum frequency of any element).
    let degree: number = 0;

    // Iterate through the array to populate the objects.
    nums.forEach((value, index) => {
        // Initialize the value in the count object if it doesn't exist.
        if (count[value] === undefined) {
            count[value] = 0;
        }
        // Increase the frequency count for this value and update the degree.
        degree = Math.max(degree, ++count[value]);

        // Record the first occurrence index of this value.
        if (firstOccurrence[value] === undefined) {
            firstOccurrence[value] = index;
        }

        // Update the last occurrence index of this value.
        lastOccurrence[value] = index;
    });

    // Variable to store the minimum length of the subarray with the same degree as the entire array.
    let minLength: number = Infinity; // 'Infinity' is used here as a large number.

    // Loop through the elements in the count object to find the length of the shortest subarray with the same degree.
    Object.keys(count).forEach(key => {
        const value = parseInt(key);
        // Check if the element has the same frequency as the degree of the array.
        if (count[value] === degree) {
            // Calculate the length of the subarray for this element.
            const length = lastOccurrence[value] - firstOccurrence[value] + 1;
            // Update the minimum length if this subarray is shorter.
            minLength = Math.min(minLength, length);
        }
    });

    // Return the minLength found.
    return minLength;
}
```

Time and Space Complexity

The time complexity of the code is primarily determined by several factors: iterating over the list once which is $O(n)$, the `Counter` operation, the `most_common()` operation, and the second iteration over the unique elements with respect to the degree.

- Constructing the counter `cnt` from the `nums` list has a time complexity of $O(n)$ as it requires a full pass through the list to count the occurrences of each element.
- `most_common()` is then called on the `Counter`, which in the worst case can take $O(k \log k)$ time complexity where k is the number of unique elements since it sorts the elements based on their counts.
- Then we have the second for-loop that iterates through each element in `nums` to update `left` and `right` dictionaries. This loop runs once for each of the n elements in the list, and hence it is $O(n)$.
- Finally, we have another for loop that goes through each of the unique elements of the `nums` list to find the shortest subarray length. In the worst case, this loop runs for k unique elements leading to $O(k)$ time complexity.

Combining these, the total worst-case time complexity of the code is $O(n + k \log k + n + k)$. Usually n will be much smaller than n as it's the count of the unique elements, so it can be approximated to $O(n + n * \log k)$ which simplifies to $O(n + k \log k)$.

For space complexity:

- We have `cnt` which is a `Counter` of all elements in `nums` and it will take $O(k)$ space.
- Two dictionaries `left` and `right` that also will store at most k elements each, contributing to another $O(k + k)$ which is $O(2k)$ space.
- There are no other significant data structures that use up space.

Thus, the total space complexity is $O(k)$ when combining `cnt`, `left`, and `right` since constants are dropped in the Big O notation.

So, the final time and space complexities are $O(n + k \log k)$ and $O(k)$ respectively.