2780. Minimum Index of a Valid Split

Medium | Array | Hash Table | Sorting | Leetcode Link

Problem Description

In this problem, we are provided with an integer array `arr` where a dominant element is defined as an element `x` that appears more than half of the time in `arr` (`freq(x) * 2 > m`, where `m` is the length of `arr`). The array `nums` we're working with is guaranteed to contain one such dominant element.

The task is to find a valid split for the array `nums` into two subarrays where both subarrays contain the same dominant element. A split at index `i` is valid if `0 <= i < n - 1` (where `n` is the length of `nums`), and both resulting subarrays (`nums[0, ..., i]` and `nums[i + 1, ..., n - 1]`) have the same dominant element. The goal is to find the minimum index `i` at which such a valid split can occur. If there is no possible valid split, we should return `-1`.

To give a simple example, if `nums` is `[2,2,1,2,2]`, splitting after the second `2` results in `[2,2]` and `[1,2,2]`, both of which have `2` as the dominant element. So the index before which we split would be `2-1` (because it's 0-indexed), hence the minimum valid split index would be `1`.

Intuition

The provided solution approach relies on the definition of a dominant element and some properties of an array with exactly one dominant element:

- Because there is only one dominant element, once we find it, we know it will be the dominant element in any valid split.
- Since the dominant element appears more than half the time, the first candidate for a possible split can only occur after we first encounter this condition in the running count of the dominant element.

Given these principles, the steps to arrive at the solution involve:

1. Finding the dominant element using a counter to determine the most common element in the array.
2. Iterating through the array, keeping track of the count of occurrences of the dominant element (`cur`).
3. As we iterate, we check two conditions at each index `i`:
   - If `cur * 2 > i`, it means the dominant element is currently more than half of the elements from `0` to `i-1`.
   - Simultaneously, we must check if the dominant element will remain dominant in the second subarray. This is done by checking if `(cnt - cur) * 2 > len(nums) - i`, where `cnt` is the total occurrences of the dominant element, and `len(nums) - i` is the length of the second subarray.

As soon as both conditions are satisfied, it indicates we've found a valid split. The solution outputs the index `i - 1`, accounting for the 0-indexed array.

By iterating over the array just once and checking the conditions, the solution effectively finds the minimum index for a valid split, thereby solving the problem in linear time with respect to the length of the array.

Solution Approach

The solution begins by using a `Counter` from the `collections` module to find the most common element in the array `nums`, which is the dominant element by the problem's definition. The `Counter` is a dictionary subclass designed for counting hashable objects, and the `most_common(1)` method retrieves the most common element along with its count.

```
x, cnt = Counter(nums).most_common(1)[0]
```

In this line of the code, `x` stands for the dominant element, while `cnt` is the number of times `x` appears in `nums`.

Next, the code uses a single-pass for loop to iterate over all the elements of the array while keeping track of two things:

- The count of the dominant element seen so far (`cur`).
- The index at which this count is processed (`i`), which is used for checking whether a split is valid.

```
for i, v in enumerate(nums, 1):
```

Here, `enumerate` is used with a start index of `1` to keep the index `i` synchronized with the length of the subarray being considered, since we're interested in the possibility of a split before the `i-th` element.

The iteration follows this logic:

1. Increment `cur` each time the dominant element is encountered.

```
if v == x:
    cur += 1
```

2. Check if we can make a valid split at index `i-1`. This check has two parts:

- To ensure that the dominant element is indeed dominant in the first subarray, we check if `cur * 2 > i`. This guarantees that there are more instances of `x` than all other elements combined until the current index.

- To ensure that the dominating element is also dominant in the second subarray after the split, we check if `(cnt - cur) * 2 > len(nums) - i`. This ensures that even after removing `cur` instances of `x`, we have enough left for `x` to remain dominant.

```
if cur * 2 > i and (cnt - cur) * 2 > len(nums) - i:
    return i - 1
```

If both conditions are satisfied, the current `i` represents a 1-indexed position at which an array can be split. Since the requested output should be 0-indexed, `i - 1` is returned.

If the for loop completes without returning, it means no valid split exists that satisfies both conditions, and in this case, the function returns `-1`.

```
return -1
```

By employing a count tracking approach, the conditionals are checked in `O(1)` time per element, leading to an overall time complexity of `O(n)`, where `n` is the number of elements in the array. The space complexity is also `O(n)` since we use a Counter to store the frequency of each element in the array.

Example Walkthrough

Let's consider a small example with the following array of numbers:

```
nums = [3, 3, 4, 2, 3]
```

Using this array, we'll walk through the solution approach described above.

1. First, we need to determine the dominant element in the array. We do this by utilizing the `Counter` class from the `collections` module:

```
from collections import Counter
x, cnt = Counter(nums).most_common(1)[0]
```

In our example, `x` will be `3` since it's the most common element, and `cnt` will be `3` because `3` appears three times in the array.

2. Then, we proceed to iterate over the array and count the occurrences of the dominant element while simultaneously checking if a valid split is possible:

```
cur = 0
for i, v in enumerate(nums, 1):
    if v == x:
        cur += 1
    if cur * 2 > i and (cnt - cur) * 2 > len(nums) - i:
        return i - 1
```

3. We examine each element in the loop:

   - At `i = 1` (`3`): `cur` becomes `1`. We cannot split yet because `cur * 2` is not greater than `cnt`.
   - At `i = 2` (`3`): `cur` becomes `2`. We cannot split yet because, although `cur * 2` is now `4` and greater than `2`, the second condition `(cnt - cur) * 2 > len(nums) - i` is not met (as `1 * 2` is not greater than `3`).
   - At `i = 3` (`4`): `cur` remains `2`. We still can't split because the first condition is no longer fulfilled.
   - At `i = 4` (`2`): `cur` remains `2`. Once again, we can't split due to the first condition not being met.
   - At `i = 5` (`3`): `cur` becomes `3`. This is the first time a split is possible as `cur * 2` is `6` and greater than `5`, and `(cnt - cur) * 2` is `0`, which is less than `len(nums) - i` which is `0`.

```
# Since we have reached the end of the array,
# and there is no point where both conditions were true,
# the result of the function would be:
return -1
```

In this example, there is no valid split because at no point do both conditions satisfy during the iteration. Thus, according to our solution, the function would return `-1`, meaning no valid minimum index split exists.

Python Solution

```python
from typing import List
from collections import Counter

class Solution:
    def minimumIndex(self, nums: List[int]) -> int:
        # Get the most common element and its count
        most_common_element, count_most_common = Counter(nums).most_common(1)[0]

        # 'current_count' will keep track of the occurrences of the most common element
        current_count = 0

        # Enumerate over the array to find the split index
        for index, value in enumerate(nums):
            # Increment the count if current value is the most common element
            if value == most_common_element:
                current_count += 1

            # Calculate the current index (1-indexed in the given code, so index + 1)
            current_index = index + 1

            # Check if our current count is greater than the half of the current index
            # and if the remaining count of the most common element is greater
            # than the half of the rest of the list
            if (current_count * 2 > current_index) and \
               ((count_most_common - current_count) * 2 > (len(nums) - current_index)):
                # Return the index (convert it back to 0-indexed by subtracting 1)
                return index

        # If no such index is found, return -1
        return -1

# Example usage:
# solution = Solution()
# result = solution.minimumIndex([1, 3, 2, 3, 2, 3, 3])
# print(result)  # It should print the index if it satisfies the condition, otherwise -1

```

Java Solution

```java
class Solution {

    // Method to find the minimum index where the most frequent number occurs
    // more frequently than all other numbers both to the left and to the right of the index.
    public int minimumIndex(List<Integer> nums) {
        int mostFrequentNum = 0;  // variable to store the most frequent number
        int maxFrequency = 0;     // variable to store the maximum frequency count
        Map<Integer, Integer> frequencyMap = new HashMap<>();  // map to store the frequency of each number

        // Count frequencies of each number in nums using a hashmap and record the most frequent number.
        for (int value : nums) {
            int currentFrequency = frequencyMap.merge(value, 1, Integer::sum);
            if (maxFrequency < currentFrequency) {
                maxFrequency = currentFrequency;
                mostFrequentNum = value;
            }
        }

        // Iterate over the list to find the minimum index where the most frequent number
        // is more common than other elements to its left and right.
        int currentFreqCount = 0; // to keep the running count of the most frequent number
        for (int i = 1; i <= nums.size(); i++) {
            if (nums.get(i - 1).equals(mostFrequentNum)) {
                currentFreqCount++;
                // Check if the most frequent number is more frequent than the remaining numbers
                // on both sides of the current index.
                if (currentFreqCount * 2 > i && (maxFrequency - currentFreqCount) * 2 > nums.size() - i) {
                    return i - 1; // Return the index if condition is met
                }
            }
        }

        return -1; // Return -1 if no such index is found
    }
}

```

C++ Solution

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

class Solution {
public:
    int minimumIndex(vector<int>& nums) {
        int x = 0; // Variable to keep track of the most frequent value encountered.
        int countMaxFreq = 0; // Counter to store the maximum frequency of an element.
        unordered_map<int, int> freqMap; // Map to keep track of the frequency of each element in 'nums'.

        // Count the frequency of each element and find the element with the highest frequency.
        for (int value : nums) {
            ++freqMap[value];
            // Update the most frequent element and its count accordingly
            if (freqMap[value] > countMaxFreq) {
                countMaxFreq = freqMap[value];
                x = value;
            }
        }

        int currentCount = 0; // Counter to track the number of occurrences of 'x' encountered so far.

        // Loop through the array to find the index where 'x' becomes the majority element
        // in both the prefix (left of index) and the suffix (right of index).
        // The condition to check for majority is that the number of occurrences of 'x' should
        // be more than half of the current index when considering prefix and more than half of
        // the remaining elements when considering the suffix.
        for (int i = 1; i <= nums.size(); ++i) {
            if (nums[i - 1] == x) {
                ++currentCount;
                // Check if 'x' is the majority element in both the prefix and suffix.
                if (currentCount * 2 > i && (countMaxFreq - currentCount) * 2 > nums.size() - i) {
                    return i - 1; // Found the index, returning the 0-based index.
                }
            }
        }

        // If no such index is found, return -1.
        return -1;
    }
};

```

Typescript Solution

```typescript
function minimumIndex(nums: number[]): number {
    // Initialize the majority element 'majorityElement' and its count 'majorityCount'.
    let [majorityElement, majorityCount] = [0, 0];
    // Create a Map to store the frequency of each element in 'nums'.
    const frequencyMap: Map<number, number> = new Map();

    // Calculate the frequency of each element and find the majority element.
    for (const value of nums) {
        const updatedCount = (frequencyMap.get(value) ?? 0) + 1;
        frequencyMap.set(value, updatedCount);
        // Update the majority element if the current value becomes the new majority.
        if (updatedCount > majorityCount) {
            [majorityElement, majorityCount] = [value, updatedCount];
        }
    }

    // Initialize the count of majority element encountered so far.
    let currentCount = 0;
    // Iterate over the array to find the minimum index that satisfies the conditions.
    for (let i = 1; i <= nums.length; ++i) {
        // If the current element is the majority element, increment count.
        if (nums[i - 1] === majorityElement) {
            currentCount++;
            // Check if the element is in majority in both the parts of the array.
            if (
                (currentCount * 2 > i) && // Majority in first part
                ((majorityCount - currentCount) * 2 > nums.length - i) // Majority in second part
            ) {
                // Return the index if both parts have the same majority element.
                return i - 1;
            }
        }
    }

    // Return `-1` if no such index is found.
    return -1;
}

```

Time and Space Complexity

Time Complexity

The given Python function `minimumIndex` first determines how many times the most common number `x` occurs in the array `nums` by using the `Counter` class from the `collections` module and retrieves the count `cnt`. This operation is `O(n)` where `n` is the length of the list `nums`, as it involves iterating through the entire list to compute the frequency of each element.

Following that, the function proceeds to iterate through the list `nums` while maintaining a cumulative count `cur` of how often it has encountered `x` so far. During iteration, it performs a constant time check in each iteration to determine if `cur` and `cnt - cur` are both more than half of the numbers seen so far and the numbers remaining, respectively.

The loop also iterates `n` times in the worst case (if it does not return early), so the total time complexity of the entire function thus remains `O(n)`, where `n` is the length of the input list.

The exact time complexity is therefore `O(n)`.

Space Complexity

In terms of space, the function uses additional memory for the `Counter` object to store the frequency of each element in `nums`. The space complexity for this part is `O(m)`, where `m` denotes the number of unique elements in `nums`. In the worst case where all elements are unique, `m` would be equivalent to `n`, resulting in `O(n)` space complexity.

Additionally, the space used for the index counter `i`, the most common element and its count `(x, cnt)` and the running count `cur` is `O(1)`, as they do not depend on the size of the input list but are merely constant size variables.

Thus, the overall space complexity of the function is `O(m)`, which is `O(n)` in the worst case when all elements in `nums` are unique.

The exact space complexity is `O(m)` with a note that it simplifies to `O(n)` in the worst-case scenario.