

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` ($\text{freq}(x) * 2 > m$, 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 $0 \leq i < n - 1$ (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:

- Finding the dominant element using a counter to determine the most common element in the array.
- Iterating through the array, keeping track of the count of occurrences of the dominant element (`cur`).
- 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 $(\text{cnt} - \text{cur}) * 2 > \text{len}(\text{nums}) - i$, where `cnt` is the total occurrences of the dominant element, and $\text{len}(\text{nums}) - i$ 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.

```
1 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.

```
1 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:

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

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

- 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 $(\text{cnt} - \text{cur}) * 2 > \text{len}(\text{nums}) - i$. This ensures that even after removing `cur` instances of `x`, we have enough left for `x` to remain dominant.

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

```
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:

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

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

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

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

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

```
1 from typing import List
2 from collections import Counter
3
4 class Solution:
5     def minimumIndex(self, nums: List[int]) -> int:
6         # Get the most common element and its count
7         most_common_element, count_most_common = Counter(nums).most_common(1)[0]
8
9         # 'current_count' will keep track of the occurrences of the most common element
10        current_count = 0
11
12        # Enumerate over the array to find the split index
13        for index, value in enumerate(nums):
14            # Increment the count if current value is the most common element
15            if value == most_common_element:
16                current_count += 1
17
18            # Calculate the current index (1-indexed in the given code, so index + 1)
19            current_index = index + 1
20
21            # Check if our current count is greater than the half of the current index
22            # and if the remaining count of the most common element is greater
23            # than the half of the rest of the list
24            if (current_count * 2 > current_index) and \
25                ((count_most_common - current_count) * 2 > (len(nums) - current_index)):
26                # Return the index (convert it back to 0-indexed by subtracting 1)
27                return index
28
29        # If no such index is found, return -1
30        return -1
31
32 # Example usage:
33 # solution = Solution()
34 # result = solution.minimumIndex([1, 3, 2, 3, 2, 3, 3])
35 # print(result) # It should print the index if it satisfies the condition, otherwise -1
36
```

Java Solution

```
1 class Solution {
2
3     // Method to find the minimum index where the most frequent number occurs
4     // more frequently than all other numbers both to the left and to the right of the index.
5     public int minimumIndex(List<Integer> nums) {
6         int mostFrequentNum = 0; // variable to store the most frequent number
7         int maxFrequency = 0; // variable to store the maximum frequency count
8         Map<Integer, Integer> freqMap = new HashMap<>(); // map to store the frequency of each number
9
10        // Count frequencies of each number in nums using a hashmap and record the most frequent number.
11        for (int value : nums) {
12            int currentFrequency = freqMap.merge(value, 1, Integer::sum);
13            if (maxFrequency < currentFrequency) {
14                maxFrequency = currentFrequency;
15                mostFrequentNum = value;
16            }
17        }
18
19        // Iterate over the list to find the minimum index where the most frequent number
20        // is more common than other elements to its left and right.
21        int currentFreqCount = 0; // to keep the running count of the most frequent number
22        for (int i = 1; i <= nums.size(); i++) {
23            if (nums.get(i - 1).equals(mostFrequentNum)) {
24                currentFreqCount++;
25                // Check if the most frequent number is more frequent than the remaining numbers
26                // on both sides of the current index.
27                if (currentFreqCount * 2 > i && (maxFrequency - currentFreqCount) * 2 > nums.size() - i) {
28                    return i - 1; // Return the index if condition is met
29                }
30            }
31        }
32
33        return -1; // Return -1 if no such index is found
34    }
35 }
36
```

C++ Solution

```
1 #include <vector>
2 #include <unordered_map>
3 using namespace std;
4
5 class Solution {
6 public:
7     int minimumIndex(vector<int>& nums) {
8         int x = 0; // Variable to keep track of the most frequent value encountered.
9         int countMaxFreq = 0; // Counter to store the maximum frequency of an element.
10        unordered_map<int, int> freqMap; // Map to keep track of the frequency of each element in 'nums'.
11
12        // Count the frequency of each element and find the element with the highest frequency.
13        for (int value : nums) {
14            ++freqMap[value];
15
16            // Update the most frequent element and its count accordingly
17            if (freqMap[value] > countMaxFreq) {
18                countMaxFreq = freqMap[value];
19                x = value;
20            }
21        }
22
23        int currentCount = 0; // Counter to track the number of occurrences of 'x' encountered so far.
24
25        // Loop through the array to find the index where 'x' becomes the majority element
26        // in both the prefix (left of index) and the suffix (right of index).
27        // The condition to check for majority is that the number of occurrences of 'x' should
28        // be more than half of the current index when considering prefix and more than half of
29        // the remaining elements when considering the suffix.
30        for (int i = 1; i <= nums.size(); ++i) {
31            if (nums[i - 1] == x) {
32                ++currentCount;
33                // Check if 'x' is the majority element in both the prefix and suffix.
34                if (currentCount * 2 > i && (countMaxFreq - currentCount) * 2 > nums.size() - i) {
35                    return i - 1; // Found the index, returning the 0-based index.
36                }
37            }
38        }
39
40        // If no such index is found, return -1.
41        return -1;
42    };
43};
```

Typescript Solution

```
1 function minimumIndex(nums: number[]): number {
2     // Initialize the majority element 'majorityElement' and its count 'majorityCount'.
3     let [majorityElement, majorityCount] = [0, 0];
4     // Create a Map to store the frequency of each element in 'nums'.
5     const frequencyMap: Map<number, number> = new Map();
6
7     // Calculate the frequency of each element and find the majority element.
8     for (const value of nums) {
9         const updatedCount = (frequencyMap.get(value) ?? 0) + 1;
10        frequencyMap.set(value, updatedCount);
11
12        // Update the majority element if the current value becomes the new majority.
13        if (updatedCount > majorityCount) {
14            [majorityElement, majorityCount] = [value, updatedCount];
15        }
16    }
17
18    // Initialize the count of majority element encountered so far.
19    let currentCount = 0;
20    // Iterate over the array to find the minimum index that satisfies the conditions.
21    for (let i = 1; i <= nums.length; ++i) {
22        // If the current element is the majority element, increment count.
23        if (nums[i - 1] === majorityElement) {
24            currentCount++;
25            // Check if the element is in majority in both the parts of the array.
26            if (
27                (currentCount * 2 > i) && // Majority in first part
28                (majorityCount - currentCount) * 2 > nums.length - i // Majority in second part
29            ) {
30                // Return the index if both parts have the same majority element.
31                return i - 1;
32            }
33        }
34    }
35
36    // Return '-1' if no such index is found.
37    return -1;
38}
```

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.