2170. Minimum Operations to Make the Array Alternating

Medium | Greedy | Array | Hash Table | Counting | Leetcode Link

Problem Description

You are provided with an array `nums` which contains `n` positive integers, and it is indexed starting from 0. These types of arrays have a particular characteristic to be called "alternating". For an array to be considered alternating, it must satisfy two conditions:

1. Every element at an even index (except the first) must be equal to the element two places before it. Mathematically, this is expressed as `nums[i - 2] == nums[i]` for `i` ranging from 2 to `n - 1`.
2. Every element at an odd index must be different from the element immediately preceding it. This is expressed as `nums[i - 1] != nums[i]` for `i` ranging from 1 to `n - 1`.

The goal is to calculate the minimum number of changes needed to be made to the array to achieve the "alternating" property. Each operation allows you to choose an index `i` and change the element `nums[i]` to any positive integer of your choosing.

Intuition

To minimize the number of operations needed to make the array alternating, we should make the most frequent elements in even and odd positions stay the same as much as possible since changing them would require more operations.

The solution involves the following steps:

1. Separate the numbers in the array into two groups based on their indices: one consisting of the numbers at even indices and the other consisting of those at odd indices.

2. Count the frequency of each number in both even and odd indexed groups. Find the most common elements (we might need the two most common elements from each group in case the most common elements in both groups are the same).

3. Calculate the minimum operations needed by considering the most common elements. If the most common element in the even-indexed group is different from the most common in the odd-indexed group, we'll keep those and change the rest. If they are the same, we have to look at the second most common elements to decide which will require fewer changes.

4. The minimum number of operations is the total length of the array minus the sum of the counts of the chosen elements for even and odd indices.

The provided code determines the most common elements and their frequencies for even and odd positions separately and then uses a generator to compute the minimum necessary changes, considering the cases where the most common even and odd elements might be the same.

Solution Approach

In the provided solution, several key programming concepts and data structures are utilized:

1. **Counter from collections module**: The `Counter` class in Python's collections module is used to count the frequency of each element in both the even-indexed and odd-indexed subarrays of `nums`. By calling `Counter(nums[i::2]).most_common(2)`, we get the top two most common elements and their respective counts for the subarray starting at index `i` with a step of 2 (meaning every other element starting from `i`).

2. **List slicing**: The slice operation `nums[i::2]` generates a new list containing every second element of `nums` starting from index `i`. This is used to separate the original array into the aforementioned even-indexed and odd-indexed subarrays.

3. **Tuple unpacking and conditional expression**: Tuples are used to represent the elements and their counts returned by `most_common()`. The conditional expression is used to return a default count of zero when there is only one common element found by `most_common()`.

4. **List comprehensions and generator expressions**: The `min()` function is used with a generator expression that iterates over all possible combinations of elements and their counts for even and odd positions. The generator expression is `min(n - (n1 + n2) for a, n1 in get(0) for b, n2 in get(1) if a != b)`. If the element for the even position (`a`) is the same as for the odd position (`b`), that particular combination is ignored (`if a != b`). Otherwise, `n1` and `n2` are the frequency counts of the chosen elements, and `n - (n1 + n2)` computes the number of changes needed.

5. **Function definition**: The `get` function is defined to encapsulate the functionality of getting the two most common elements from either the even-indexed or odd-indexed numbers. It handles edge cases where there might be less than two elements returned from the Counter.

By combining these tools and methods, the solution efficiently calculates the minimum number of operations needed to make the input array alternating. The use of Counter and most_common methods has a time complexity of O(N), where N is the length of the array. Since we iterate over at most four combinations (two from each list split), the total complexity remains O(N).

Example Walkthrough

Let's consider a small example to illustrate the solution approach described above. Suppose we are given the following array `nums`:

```
nums = [1, 2, 2, 3, 1, 5]
```

During the walkthrough, we'll use the steps detailed in the solution approach.

Step 1: Separation into even and odd indexed groups
Separate `nums` into two groups based on indices:
Even-indexed group: `[1, 2, 1]` (elements at indices 0, 2, 4)
Odd-indexed group: `[2, 3, 5]` (elements at indices 1, 3, 5)

Step 2: Count the frequency
Count the frequency of elements in each group:
Even-indexed group: `1` appears 2 times, `2` appears 1 time
Odd-indexed group: `2`, `3`, and `5` each appear 1 time

Step 3: Find the most common elements
Identify the most common elements:
Even-indexed most common element: `1`, with frequency 2
Odd-indexed most common element: Each element has the same frequency 1, but let's say we choose `2`
Since `2` is also present in even-indexed group but is not the most frequent, we consider it for the odd group. No conflict yet.

Step 4: Calculate the minimum operations
Calculate the minimum changes needed:

- If we keep element `1` in the even positions, we need to change one element (`2` at index 2).
- For the odd positions, if we choose to keep `2`, we don't need to make any changes.

Summarizing the changes:

- Total array length = 6
- Count of `1`'s in even-indexed group = 2
- Count of `2`'s in odd-indexed group = 1 (since we only need to consider odd-indexed positions)

Minimum number of operations = Total length - (Even count + Odd count)
= 6 - (2 + 1)
= 3 changes

In a scenario where the most frequent even and odd elements were the same, we would require looking at the second most common elements, but in our example, this isn't the case.

By applying the solution approach, we understand that to make `nums` alternating, we need to make 3 changes: replace the second element of the even group and two elements of the odd group that are not `2`.

Python Solution

```python
from collections import Counter  # Importing Counter from collections module

class Solution:
    def minimumOperations(self, nums: List[int]) -> int:
        # Internal function to get the two most common elements and their counts in alternating indices.
        def get_most_common_alternate(idx):
            # Count the elements at alternating indices starting from idx
            counts = Counter(nums[idx::2]).most_common(2)
            # If there are no elements, return two pairs of zeros
            if not counts:
                return [(0, 0), (0, 0)]
            # If there is only one element, return it with a pair of zeros
            if len(counts) == 1:
                return [counts[0], (0, 0)]
            # Otherwise return the two most common elements and their counts
            return counts

        # This stores the length of the nums list
        n = len(nums)
        # The result is the minimum count of operations which is obtained by taking the
        # total count 'n' minus the sum of counts of the two most common elements at
        # alternating indices that are not the same.
        return min(
            n - (count_even + count_odd)  # Calculate the minimum number
            for value_even, count_even in get_most_common_alternate(0)  # Get most common at even indices
            for value_odd, count_odd in get_most_common_alternate(1)  # Get most common at odd indices
            if value_even != value_odd  # Only consider pairs with different values
        )

```

Please note that `List` must be imported from `typing` if it hasn't been defined earlier in the code. Otherwise, Python will raise a `NameError`. Here's how you can import it:

```python
from typing import List

```

Java Solution

```java
class Solution {
    private int[] numbers; // An array of numbers
    private int arrayLength; // Length of the numbers array

    // Calculates the minimum number of operations required
    public int minimumOperations(int[] nums) {
        this.numbers = nums; // Assigns passed array to the numbers property
        arrayLength = nums.length; // Assigns array length to arrayLength property
        int minOperations = Integer.MAX_VALUE; // Initialize minimum operations with the maximum integer value

        // Generate frequency arrays for even and odd indices, and compare them
        for (int[] evenFreqElement : getFrequency(0)) {
            for (int[] oddFreqElement : getFrequency(1)) {
                // Ensure that we consider different values (from even and odd indices)
                if (evenFreqElement[0] != oddFreqElement[0]) {
                    // Calculate and update the minimum operations needed
                    minOperations = Math.min(minOperations, arrayLength - (evenFreqElement[1] + oddFreqElement[1]));
                }
            }
        }
        return minOperations; // return the minimum operations calculated
    }

    // Generates a 2D array containing the two most frequent elements and their counts at either even or odd indices
    private int[][] getFrequency(int start) {
        Map<Integer, Integer> frequencyMap = new HashMap<>(); // Map to keep track of frequency of elements
        // Loop through the numbers array, incrementing by 2 from the specified start index
        for (int i = start; i < arrayLength; i += 2) {
            frequencyMap.put(numbers[i], frequencyMap.getOrDefault(numbers[i], 0) + 1); // Update frequency map
        }
        int topValue = 0; // Most frequent number
        int topCount = 0; // Count of the most frequent number
        int secondValue = 0; // Second most frequent number
        int secondCount = 0; // Count of the second most frequent number

        // Iterate over the frequency map to find the most and second most frequent numbers
        for (Map.Entry<Integer, Integer> entry : frequencyMap.entrySet()) {
            int key = entry.getKey();
            int value = entry.getValue();
            // If current frequency is greater than the most frequent, update top and second top frequencies and values
            if (value > topCount) {
                secondValue = topValue;
                secondCount = topCount;
                topValue = key;
                topCount = value;
            } else if (value > secondCount) { // If current is only greater than second, update only second most frequent
                secondValue = key;
                secondCount = value;
            }
        }
        // Return an array of arrays, where each array is of format: {element value, frequency count}
        return new int[][] {{topValue, topCount}, {secondValue, secondCount}};
    }
}

```

C++ Solution

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

// Define a pair of integers for easier handling of operations.
typedef pair<int, int> IntPair;

class Solution {
public:
    int minimumOperations(vector<int>& nums) {
        int minOperations = INT_MAX; // Initialize with the maximum possible integer value.
        int size = nums.size(); // Get the size of the input array.
        // Get the frequency pairs of numbers appearing at even indices.
        for (const auto& [numEven, frequencyEven] : getFrequency(0, nums)) {
            // Get the frequency pairs of numbers appearing at odd indices.
            for (const auto& [numOdd, frequencyOdd] : getFrequency(1, nums)) {
                // Ensure we are not considering the same number for both even and odd positions.
                if (numEven != numOdd) {
                    // Calculate the minimum operations required and update minOperations.
                    minOperations = min(minOperations, size - (frequencyEven + frequencyOdd));
                }
            }
        }
        return minOperations; // Return the calculated minimum operations.
    }

    // Helper method to get top two frequent numbers and their counts from even or odd indices.
    vector<IntPair> getFrequency(int startIndex, vector<int>& nums) {
        unordered_map<int, int> frequencyMap; // Map to hold number frequencies.

        // Build frequency map for numbers at indices determined by startIndex (even or odd).
        for (int i = startIndex; i < nums.size(); i += 2) {
            ++frequencyMap[nums[i]];
        }

        // Variables to keep track of the two most frequent numbers and their counts.
        int firstNum = 0, firstCount = 0;
        int secondNum = 0, secondCount = 0;

        // Find the two most frequent numbers.
        for (const auto& [currentNum, count] : frequencyMap) {
            if (count > firstCount) {
                // Update second most frequent number before changing the most frequent one.
                secondNum = firstNum;
                secondCount = firstCount;
                firstNum = currentNum;
                firstCount = count;
            } else if (count > secondCount) {
                secondNum = currentNum;
                secondCount = count;
            }
        }

        // Return the top two frequent numbers and their counts.
        return {{firstNum, firstCount}, {secondNum, secondCount}};
    }
};

```

Typescript Solution

```typescript
/**
 * Finds the minimum number of operations required to make all the even-indexed elements
 * equal and all the odd-indexed elements equal, but not necessarily equal to each other
 * @param nums The input array of numbers
 * @returns The minimum number of operations required
 */
function minimumOperations(nums: number[]): number {
    const length = nums.length;
    const maxValue = 10 ** 5;

    // Initialize frequency arrays for odd-indexed and even-indexed elements
    let oddFrequencies = new Array(maxValue).fill(0);
    let evenFrequencies = new Array(maxValue).fill(0);

    // Populate the frequency arrays
    for (let i = 0; i < length; i++) {
        const current = nums[i];
        if (i % 2 === 1) {
            oddFrequencies[current]++;
        } else {
            evenFrequencies[current]++;
        }
    }

    // Find the max frequency element for odd and even indices
    let oddIndexMax = oddFrequencies.indexOf(Math.max(...oddFrequencies));
    let evenIndexMax = evenFrequencies.indexOf(Math.max(...evenFrequencies));

    // If the max frequency elements are different, the answer is straightforward
    if (oddIndexMax !== evenIndexMax) {
        return length - oddFrequencies[oddIndexMax] - evenFrequencies[evenIndexMax];
    } else {
        // Handle the case when the max frequency elements are the same

        let oddMaxFreq = oddFrequencies[oddIndexMax];
        let evenMaxFreq = evenFrequencies[evenIndexMax];

        // Zero out the max frequency elements to find the second max
        oddFrequencies[oddIndexMax] = 0;
        evenFrequencies[evenIndexMax] = 0;

        let oddSecondMaxIndex = oddFrequencies.indexOf(Math.max(...oddFrequencies));
        let evenSecondMaxIndex = evenFrequencies.indexOf(Math.max(...evenFrequencies));

        // The answer is the max of either changing the even-indexed or the odd-indexed elements
        return length - Math.max(
            oddMaxFreq + evenFrequencies[evenSecondMaxIndex],
            evenMaxFreq + oddFrequencies[oddSecondMaxIndex]
        );
    }
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code can be analyzed by examining the operations within the `minimumOperations` function.

1. `get` Function Call for Half of the Array (Based on Even and Odd Indices): This operation involves creating a Counter on half of the array, which takes `O(n/2)`, where `n` is the length of `nums`. Since the `get` function is called twice, once for even indices and once for odd indices, this results in a complexity of `O(n)`.

2. `Counter(nums[i::2]).most_common(2)`: This operation finds the two most common elements in the Counter object. The `Counter.most_common` method has `O(n log k)` complexity, where `k` is the number of most common elements requested. Since `k` is constant (2 in this case), the complexity for this step simplifies to `O(n)` (for half of the list, but effectively `O(n)` since called twice for both halves of the list).

3. The list comprehension iterates through all combinations of two most common elements for even and odd indices. In the worst case, there are 2 elements from the first half and 2 from the second half, creating 4 combinations to consider. This is a constant factor and does not depend on `n`, so it is `O(1)`.

Therefore, the overall time complexity of the provided code is `O(n)`.

Space Complexity

The space complexity can be determined by examining the memory use relative to the input size:

1. The `Counter` object in the `get` method stores the frequencies of elements for half of the array, assuming a worst-case scenario where all items are unique this would be `O(n/2)`, but since it's called twice, the space complexity is `O(n)` in total for both Counters.

2. The most common elements are stored as a list of tuples, with up to four tuples stored at any time (`most_common(2)` for two halves). This is a constant space `O(1)`.

Therefore, the overall space complexity of the code is `O(n)`.