2672. Number of Adjacent Elements With the Same Color

Medium Array Leetcode Link

Problem Description

In this problem, we are given an array called `nums` that has a length `n` and is initialized with all elements being uncolored (value of `0`). We are also provided with a list of queries where each query is represented as `[index_i, color_i]`. For each query, we are asked to color the element at `index_i` in `nums` with `color_i`. After processing each query, we need to count the number of pairs of adjacent elements that have the same color and are not uncolored.

To clarify, we need to return an array where each element corresponds to a query from the given list, and it indicates how many pairs of adjacent elements in `nums` are matching in color and not uncolored, after applying that query.

Intuition

The main challenge in this problem is efficiently updating the color counts after each query since looking at all elements after every query might be too slow. To optimize this, we can only focus on the element at `index_i` that is being colored by the current query and its neighbors, since the rest of the array remains unchanged.

When we color `nums[i]` with `color_i`, we need to consider the following:

- If `nums[i]` was already colored (not zero) and had the same color as its neighbor(s), we had a pre-existing pair(s) of same-colored adjacent elements. Changing the color of `nums[i]` will break these pairs, so we should decrement our count by the number of such pairs.
- After we recolor `nums[i]`, it might form a new pair(s) of same-colored adjacent elements if it matches the color of its neighbor(s). In that case, we should increment our count relative to the new pairs formed.

We avoid iterating over the whole array and keep a running total of the same-colored pairs. To implement this:

1. Keep a variable `x` to count the total number of same-colored adjacent pairs at any given stage.
2. Iterate through the queries, and for each query:
   - Check the existing color at `index_i`. If it is the same as the color of its left (if `i > 0`) or right neighbor (if `i < n - 1`), decrement `x` for each match before updating `nums[i]` color, since it will break that pair.
   - Update `nums[i]` with `color_i`.
   - Then, check if `nums[i]` formed a new same-colored pair with its neighbors after being recolored. If so, increment `x` for each new pair formed.
   - Record the current count `x` in the `ans` array, which corresponds to the state after the current query is processed.
3. Return the `ans` array once all queries have been processed.

Solution Approach

The solution follows a straightforward approach by keeping track of the current count of adjacent same-colored pairs as it processes each query. The main focus is on the effect each query has on the number of these pairs. To understand how the solution works, let's walk through the implementation steps with reference to the given solution code:

1. Initialize a list `nums` of length `n` with all values set to `0` to represent the uncolored array, and `ans` with the length of queries to store the result after each query.
2. Initialize a variable `x` to `0`. This variable will keep track of the number of adjacent same-colored (not uncolored) pairs present in `nums` at any point.
3. Loop through each query provided in `queries`, where `k` is the index of the query, and `(i, c)` represents the `index_i` and `color_i` of that particular query:
   - If the element at `index_i` (`nums[i]`) is already colored (not equal to `0`) and has the same color as its left neighbor (`nums[i - 1]`), it means we currently have a same-colored pair that will be broken by recoloring. So, decrement `x` as this pair will no longer exist after the current query.
   - Similarly, if `nums[i]` has the same color as its right neighbor (`nums[i + 1]`), and it's not uncolored, decrement `x` as this pair will also be dissolved.
4. Now, apply the query. Color the element at `index_i` in `nums` with `color_i` (`nums[i] = c`).
5. After applying the query, check for new same-colored pairs:
   - If the newly colored `nums[i]` matches the color of its left neighbor (`nums[i - 1]`), we should increment `x`, since a new same-colored pair has been formed.
   - Do the same for the right neighbor. If `nums[i + 1]` matches the new color of `nums[i]`, increment `x` again for this new pair.
6. Store the updated value of `x` into `ans[k]` to reflect the current number of adjacent same-colored pairs after the execution of the `k`th query.

The algorithm makes use of:

- **Array Data Structure**: To store the initial state of the array (`nums`) and the result after each query (`ans`).
- **Looping and Conditional Logic**: To iterate through the queries and apply the necessary updates based on adjacent element colors.

By focusing only on the immediate neighbors of the index being colored in each query, the solution avoids any unnecessary operations on the rest of the array, allowing for an efficient update of the same-colored pairs count after each query.

Here's the final template filled with the content:

```

The solution to this LeetCode problem uses an array to represent the initial state of uncolored elements and processes a series of cc

1. Initialize an array `nums` to represent the uncolored elements and `ans` to store the number of same-colored adjacent pairs after

2. Initialize a variable `x` to maintain a running total of the same-colored adjacent pairs.

3. Loop over each query, where `index_i` is the array index to be colored and `color_i` is the color to apply.

   - Before changing the color, check if the current color at `index_i` forms same-colored pairs with its neighbors. If such pairs ex

4. Color the element at `index_i` with `color_i`.

5. Check if the newly colored element forms new same-colored pairs with its neighbors. If new pairs are formed, increment `x`.

6. Save the updated count `x` into the `ans` array corresponding to the current query's result.

The efficient check for pairs before and after each query allows the algorithm to maintain an accurate count without iterating throug
```

Example Walkthrough

Let's consider a small example to illustrate how the solution works.

Suppose we have an array `nums` with `n = 5` elements, all initialized to `0` (uncolored). Let's consider `queries = [[1, 3], [2, 3], [1, 3], [3, 1]]`.

- After the first query `[1, 3]`, color index `1` with color `3`. `nums` becomes `[0, 3, 0, 0, 0]`. No adjacent pairs are matching, so `ans[0] = 0`.
- After the second query `[2, 3]`, color index `2` with color `3`. `nums` now is `[0, 3, 3, 0, 0]`. There is a new adjacent pair `3,3` at indices `1` and `2`. So we have one matching pair. `ans[1] = 1`.
- The third query `[1, 3]` asks us to color index `1` with color `3` again. But it's already color `3`, so there is no change. The array remains `[0, 3, 3, 0, 0]`, and the number of matching pairs is also unchanged. `ans[2] = 1`.
- Finally, the fourth query `[3, 1]` asks us to color index `3` with color `1`. `nums` changes to `[0, 3, 3, 1, 0]`. Now, there are no adjacent pairs with the same color, as the new color at index `3` has broken the existing pair. `ans[3] = 0`.

We end up with the final `ans` array as `[0, 1, 1, 0]`.

Breaking this down step-by-step according to the solution approach:

1. Initialize `nums` as `[0, 0, 0, 0, 0]` and `ans` as an empty array to hold the results after each query.
2. Set `x` to `0`. No matching pairs yet.
3. Process the first query `(1, 3)`:
   - `nums[1]` is `0`, updating it to `3` does not break any pair, so `x` remains `0`.
   - Update `nums` to `[0, 3, 0, 0, 0]`.
   - No new pairs, update `ans` to `[0]`.
4. Process the second query `(2, 3)`:
   - `nums[2]` is `0`, so again updating it to `3` breaks no pairs, leaving `x` as `0`.
   - Update `nums` to `[0, 3, 3, 0, 0]`.
   - A new pair `3,3` is formed, increment `x` to `1`.
   - Update `ans` to `[0, 1]`.
5. Process the third query `(1, 3)`:
   - `nums[1]` is already `3`, updating it to `3` changes nothing.
   - `nums` remains `[0, 3, 3, 0, 0]`.
   - No pairs are broken or formed, so `x` stays `1`.
   - Update `ans` to `[0, 1, 1]`.
6. Process the fourth query `(3, 1)`:
   - `nums[3]` is `0`, updating it to `1` does not break any pairs, so `x` remains `1`.
   - Update `nums` to `[0, 3, 3, 1, 0]`.
   - The existing pair `3,3` is now broken, so decrement `x` to `0`.
   - Update `ans` to `[0, 1, 1, 0]`.

In the end, the `ans` array reflects the number of matching adjacent pairs after each query, demonstrating how the solution efficiently processes queries to dynamically maintain the count of matching pairs.

Python Solution

```python
class Solution:
    def colorTheArray(self, size: int, queries: List[List[int]]) -> List[int]:
        # Initialize the array with zeros indicating no color.
        array = [0] * size
        # Initialize the answer list to store results of each query.
        result = [0] * len(queries)
        # Initialize a variable to track the number of adjacent pairs with the same color.
        adjacent_same_color_count = 0

        # Iterate over the queries to process them sequentially.
        for query_index, (position, color) in enumerate(queries):
            # Decrement the count if the current position has a color and
            # the previous position's color is the same as the current color.
            if position > 0 and array[position] and array[position - 1] == array[position]:
                adjacent_same_color_count -= 1
            # Decrement the count if the current position has a color and
            # the next position's color is the same as the current color.
            if position < size - 1 and array[position] and array[position + 1] == array[position]:
                adjacent_same_color_count -= 1

            # Increment the count if the previous position's color is the same as the new color.
            if position > 0 and array[position - 1] == color:
                adjacent_same_color_count += 1
            # Increment the count if the next position's color is the same as the new color.
            if position < size - 1 and array[position + 1] == color:
                adjacent_same_color_count += 1

            # Record the count in the result after processing the query.
            result[query_index] = adjacent_same_color_count
            # Update the color of the current position.
            array[position] = color

        # Return the result list after processing all queries.
        return result
```

Note: To use this code, you would need to have the appropriate typing imports at the top of the file:

```python
from typing import List

```

Java Solution

```java
class Solution {
    public int[] colorTheArray(int n, int[][] queries) {
        int numQueries = queries.length;                 // Number of queries
        int[] arrayColors = new int[n];                 // Array to keep track of colors
        int[] answer = new int[numQueries];             // Array to store answers

        // Initialize count of pairs with same color
        int sameColorPairsCount = 0;

        // Iterate through all queries
        for (int queryIndex = 0; queryIndex < numQueries; ++queryIndex) {
            int position = queries[queryIndex][0];    // Position to color
            int color = queries[queryIndex][1];       // Color to apply

            // Decrease count if removing a pair of the same color to the left
            if (position > 0 && arrayColors[position] > 0 && arrayColors[position - 1] == arrayColors[position]) {
                --sameColorPairsCount;
            }
            // Decrease count if removing a pair of the same color to the right
            if (position < n - 1 && arrayColors[position] > 0 && arrayColors[position + 1] == arrayColors[position]) {
                --sameColorPairsCount;
            }
            // Increase count if creating a new pair of the same color to the left
            if (position > 0 && arrayColors[position - 1] == color) {
                ++sameColorPairsCount;
            }
            // Increase count if creating a new pair of the same color to the right
            if (position < n - 1 && arrayColors[position + 1] == color) {
                ++sameColorPairsCount;
            }

            // Store the new count after this query
            answer[queryIndex] = sameColorPairsCount;

            // Apply the new color
            arrayColors[position] = color;
        }

        // Return the array with answers to the queries
        return answer;
    }
}

```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Method that takes the size of the array (n) and a list of queries
    // Each query is a vector with two integers: index i and color c
    std::vector<int> colorTheArray(int n, std::vector<std::vector<int>>& queries) {
        // Create an array (nums) of size n to keep track of the colors
        std::vector<int> nums(n, 0);
        // Create an array (ans) to store the answers to the queries
        std::vector<int> ans;

        // Variable to keep track of the number of adjacent pairs with the same color
        int adjacentPairsCount = 0;

        // Iterate through each query in queries
        for (auto& query : queries) {
            // Extract index (i) and color (c) from the query
            int index = query[0], color = query[1];

            // Check if the current index has a left neighbor with the same color
            // and if its color is already set (value greater than 0),
            // decrement the count of adjacent pairs
            if (index > 0 && nums[index] > 0 && nums[index - 1] == nums[index]) {
                --adjacentPairsCount;
            }

            // Check if the current index has a right neighbor with the same color
            // and if its color is already set (value greater than 0),
            // decrement the count of adjacent pairs
            if (index < n - 1 && nums[index] > 0 && nums[index + 1] == nums[index]) {
                --adjacentPairsCount;
            }

            // Set the color at the current index to the color specified in the query
            nums[index] = color;

            // Check if the new color at the current index creates a new adjacent pair
            // with the left neighbor, increment the count of adjacent pairs
            if (index > 0 && nums[index - 1] == color) {
                ++adjacentPairsCount;
            }

            // Check if the new color at the current index creates a new adjacent pair
            // with the right neighbor, increment the count of adjacent pairs
            if (index < n - 1 && nums[index + 1] == color) {
                ++adjacentPairsCount;
            }

            // Add the current count of adjacent pairs to the answers array
            ans.push_back(adjacentPairsCount);
        }

        // Return the answers array after processing all queries
        return ans;
    }
};

```

Typescript Solution

```typescript
function colorTheArray(n: number, queries: number[][]): number[] {
    // Initialize the array representing the colors of the n elements
    const colors: number[] = new Array(n).fill(0);

    // This will store the answer to how many neighboring pairs match after each query
    const result: number[] = [];

    // Variable to track the number of matching neighbors
    let matchingNeighbors = 0;

    // Iterate over each query where each query is an array [index, color]
    for (const [index, color] of queries) {
        // Decrease matchingNeighbors count if the current color matches the previous
        // and there was a color (not 0) before the change
        if (index > 0 && colors[index] > 0 && colors[index - 1] === colors[index]) {
            --matchingNeighbors;
        }

        // Decrease matchingNeighbors count if the current color matches the next
        // and there was a color (not 0) before the change
        if (index < n - 1 && colors[index] > 0 && colors[index + 1] === colors[index]) {
            --matchingNeighbors;
        }

        // Change the color at the given index
        colors[index] = color;

        // Increase matchingNeighbors count if the new color matches the previous
        if (index > 0 && colors[index - 1] === color) {
            ++matchingNeighbors;
        }

        // Increase matchingNeighbors count if the new color matches the next
        if (index < n - 1 && colors[index + 1] === color) {
            ++matchingNeighbors;
        }

        // Add the current number of matching neighbors to the result array
        result.push(matchingNeighbors);
    }

    // Return the result array
    return result;
}

```

Time and Space Complexity

Time Complexity

The presented algorithm iterates through the `queries` list once, processing each query in what is largely a constant time operation. The primary operations within the loop include:

- Accessing and updating elements in the `nums` list based on index, which is an `O(1)` operation.
- Checking conditions and incrementing or decrementing `x`, which is also an `O(1)` operation.

Since these `O(1)` operations are all that occur in the loop and the loop runs for each query in `queries`, the time complexity of the algorithm is `O(q)`, where `q` is the number of queries.

Space Complexity

The space complexity of the algorithm includes:

- The `nums` list which is initialized with `n` elements, resulting in `O(n)` space.
- The `ans` list which is also proportional to the number of queries `q`, which makes it `O(q)` space.

Since these two lists are not dependent on each other, the total space complexity of the algorithm is `O(n + q)`, accounting for both the `nums` array of `n` elements and the `ans` array of `q` elements.