1182. Shortest Distance to Target Color

Medium Array Binary Search Dynamic Programming Leetcode Link

Problem Description

You are presented with an array called `colors`, in which the elements are integers representing different colors. The integers 1, 2, and 3 correspond to three distinct colors. Along with this array, a set of queries is provided, each query being a pair of integers `(i, c)` where `i` is an index in the `colors` array and `c` is one of the three colors. Your task is to find the shortest distance from the position `i` to the location of the requested color `c` in the array. The distance is measured as the number of steps needed to move from index `i` to the index of the target color `c`. If the target color `c` does not exist in the array, you should return -1.

Intuition

The essence of the solution is to precompute and store the closest occurrence of each color to every index in the array, both from the left and the right directions. Once we have this precomputed information, any query can be answered quickly by comparing the distances from the index `i` to the nearest occurrences of color `c` on both sides.

Here's a step-by-step breakdown of the intuition:

1. We create two tables (arrays), `right` and `left`, that will contain, for each index `i` in the colors array, the nearest index on the right and left respectively where each of the three colors is found.
2. Starting from the right end of the `colors` array, we fill the `right` table with indices of the closest occurrence of each color as we move leftward. We initialize the table with infinity (`inf`) to signify that initially, we haven't found any colors.
3. Similarly, starting from the left end of the `colors` array, we fill the `left` table with indices of the closest occurrence of each color as we move rightward. The table is initially filled with `-inf` to indicate no colors found.
4. For each query `(i, c)`, we determine the distance to the color `c` by taking the minimum between the distance to the closest occurrence of color `c` on the right and on the left of the index `i`.
5. If the computed minimum distance is greater than the size of the `colors` array, this indicates that there is no such color `c` in the `colors` array, and we return -1 for that query. Otherwise, we return the computed distance.

This approach makes it possible to answer each query in constant time after the initial precomputation, which is done in linear time relative to the size of the colors array. This is much more efficient than a straightforward approach where you might check the entire array for each individual query.

Solution Approach

The solution provided above does not use a binary search approach as mentioned in the Reference Solution Approach. Instead, it relies on dynamic programming with precomputation to find the nearest occurrence of each color to every index, from both directions. The steps can be explained as follows:

1. Two 2D tables, `right` and `left`, are created to record the indexes of the nearest color occurrence on the right and left sides of each index `i`. Both are initialized with appropriate values to indicate that initially, no color has been found yet (using `inf` for the right direction, and `-inf` for the left).
2. The `right` table is filled by traversing the `colors` array in reverse, from `n - 1` to `0`. At each index `i`, it updates only the column corresponding to `colors[i] - 1` (since colors are 1-indexed, we subtract 1 to use zero-based indexing) with the current index `i`. This implies updating the position of the encountered color and leaving the others as previously computed.
3. The `left` table is populated by iterating through the `colors` array from the start, incrementing index `i`. Similar to the `right` table, it updates the column corresponding to `colors[i] - 1` at each step.
4. Once the tables are built, the solution evaluates each query `(i, c)`. It calculates the distances to color `c` from both the right and the left tables for the given index `i`. The shorter of the two distances is considered, as we are looking for the shortest path to the desired color.
5. If the smallest distance is greater than the length of the array, it indicates that the target color does not exist, and `-1` is returned for that query. Otherwise, the shortest calculated distance is returned.

The tables `right` and `left` exploit the fact that `colors` array indices and the colors themselves can be indexed and compared directly to store relevant distances quickly. These precomputation tables allow the queries to be answered swiftly in O(1) time after the initial setup, thus making the solution efficient and suitable for scenarios with multiple queries on the same `colors` array.

This approach enables the solution to bypass binary search, providing a direct access method to retrieve the shortest distance for each query. While binary search is a powerful tool for searching in sorted arrays, in this context, the precomputation strategy streamlines query handling and avoids repeated search operations.

Example Walkthrough

Let's illustrate the solution approach with a small example. Assume we have an array `colors` and a set of queries as follows:

`colors` array: `[1, 3, 2, 1, 2, 3, 3, 2]` queries: `[(4, 3), (2, 1), (5, 2)]`

Step 1: Initialize `right` and `left` tables

We create two 2D tables `right` and `left`. Since we have 3 colors, each table will have 3 columns, and the number of rows will be equal to the length of the `colors` array. We initialize `right` table with `inf` and `left` table with `-inf`.

Initial `right` table (8 rows, 3 columns, initialized with `inf`):

```
inf inf inf
inf inf inf
inf inf inf
inf inf inf
inf inf inf
inf inf inf
inf inf inf
inf inf inf
```

Initial `left` table (8 rows, 3 columns, initialized with `-inf`):

```
-inf -inf -inf
-inf -inf -inf
-inf -inf -inf
-inf -inf -inf
-inf -inf -inf
-inf -inf -inf
-inf -inf -inf
-inf -inf -inf
```

Step 2: Populate the `right` table

We traverse the `colors` array from right to left, updating the `right` table as we encounter each color.

Final `right` table:

```
 5   7   6
 5   7   6
 5   7   3
 5   7   3
 5   5   3
 5   5   5
-1   5   5
-1   7   5
```

Step 3: Populate the `left` table

We traverse the `colors` array from left to right, updating the `left` table with encountered colors.

Final `left` table:

```
-1  -1  -1
 0  -1   1
 0  -1   1
 3  -1   1
 3   4   1
 3   4   5
 3   4   6
 3   7   6
```

Step 4: Evaluate queries

For each query `(i, c)`, we calculate the two distances to color `c` from both `right` and `left` tables for index `i`, and we take the shorter one.

1. Query `(4, 3)` would check `right[4][2]` and `left[4][2]` since color '3' corresponds to the 3rd column (index 2). Thus, we have `right[4][2] = 5` and `left[4][2] = 1`. The shortest distance is `5 - 4 = 1` from the left.
2. Query `(2, 1)` checks `right[2][0]` and `left[2][0]`. We get `right[2][0] = 5` and `left[2][0] = 0`. The shortest distance is `2 - 0 = 2` from the left.
3. Query `(5, 2)` checks `right[5][1]` and `left[5][1]`. We find `right[5][1] = 5` and `left[5][1] = 4`. The shortest distance is `5 - 4 = 1` from the left.

Step 5: Return results for queries

The results of the queries are distances `[1, 2, 1]` correspondingly.

By precomputing the `right` and `left` tables, we efficiently answer each query in constant time with the precomputed values without the need for a binary search for each query. This is particularly advantageous when there are many queries, as the time savings can be significant.

Python Solution

```python
from typing import List

class Solution:
    def shortestDistanceColor(
        self, colors: List[int], queries: List[List[int]]
    ) -> List[int]:
        n = len(colors)

        # Precompute the nearest index of each color to the right of each element
        right_nearest = [[float('inf')] * 3 for _ in range(n + 1)]
        for i in range(n - 1, -1, -1):
            for color in range(3):
                right_nearest[i][color] = right_nearest[i + 1][color]
            # Update the nearest index for the current color
            right_nearest[i][colors[i] - 1] = i

        # Precompute the nearest index of each color to the left of each element
        left_nearest = [[-float('inf')] * 3 for _ in range(n + 1)]
        for i, color in enumerate(colors, 1):
            for color_index in range(3):
                left_nearest[i][color_index] = left_nearest[i - 1][color_index]
            # Update the nearest index for the current color
            left_nearest[i][color - 1] = i - 1

        # Process each query to find the shortest distance to the target color
        answers = []
        for index, color in queries:
            # Calculate the distance to the nearest occurrence of the desired color
            distance = min(index - left_nearest[index + 1][color - 1], right_nearest[index][color - 1] - index)

            # If the distance is greater than the number of colors, the color does not exist
            answers.append(-1 if distance > n else distance)

        return answers
```

Java Solution

```java
class Solution {
    public List<Integer> shortestDistanceColor(int[] colors, int[][] queries) {
        // Get the length of the colors array
        int n = colors.length;
        // Using 'inf' as a large value to represent infinity
        final int inf = 1 << 30;

        // Create 'right' 2D array to store closest color index on the right
        int[][] right = new int[n + 1][3];
        Arrays.fill(right[n], inf); // Initialize the right-most boundary with 'inf'
        for(int i = n - 1; i >= 0; --i) {
            for(int j = 0; j < 3; ++j) {
                right[i][j] = right[i + 1][j]; // Copy the previous values
            }
            right[i][colors[i] - 1] = i; // Update the closest color index
        }

        // Create 'left' 2D array to store closest color index on the left
        int[][] left = new int[n + 1][3];
        Arrays.fill(left[0], -inf); // Initialize the left-most boundary with negative 'inf'
        for(int i = 1; i <= n; ++i) {
            for(int j = 0; j < 3; ++j) {
                left[i][j] = left[i - 1][j]; // Copy the previous values
            }
            left[i][colors[i - 1] - 1] = i - 1; // Update the closest color index
        }

        // Initialize an answer list to store results for the queries
        List<Integer> answer = new ArrayList<>();
        for(int[] query : queries) {
            int index = query[0];  // Query index
            int color = query[1] - 1; // Query color, adjusted for 0-based indexing
            // Compute the distance to the closest color by comparing left and right distances
            int distance = Math.min(index - left[index + 1][color], right[index][color] - index);
            // If the computed distance is greater than the array length, there's no such color
            answer.add(distance > n ? -1 : distance);
        }

        // Return the list containing answers to the queries
        return answer;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    vector<int> shortestDistanceColor(vector<int>& colors, vector<vector<int>>& queries) {
        // n is the total number of colors in the array
        int n = colors.size();

        // Initialize a 2D array to store the nearest right index of each color
        vector<vector<int>> nearestRightIndex(n + 1, vector<int>(3, INT_MAX));
        for (int i = n - 1; i >= 0; --i) {
            for (int color = 0; color < 3; ++color) {
                nearestRightIndex[i][color] = nearestRightIndex[i + 1][color];
            }
            // Update the index of the current color
            nearestRightIndex[i][colors[i] - 1] = i;
        }

        // Initialize a 2D array to store the nearest left index of each color
        vector<vector<int>> nearestLeftIndex(n + 1, vector<int>(3, INT_MIN));
        for (int i = 1; i <= n; ++i) {
            for (int color = 0; color < 3; ++color) {
                nearestLeftIndex[i][color] = nearestLeftIndex[i - 1][color];
            }
            // Update the index of the current color
            nearestLeftIndex[i][colors[i - 1] - 1] = i - 1;
        }

        // Vector to store the answer to the queries
        vector<int> answer;
        for (auto& query : queries) {
            // Index and color for the current query
            int idx = query[0], color = query[1] - 1;

            // Calculate the distance to the nearest color both left and right
            int distanceToNearestColor = min(
                idx - nearestLeftIndex[idx + 1][color],
                nearestRightIndex[idx][color] - idx
            );

            // If the distance is greater than the number of colors, it means no such color found
            if (distanceToNearestColor > n) {
                answer.push_back(-1);
            } else {
                // Add the minimum distance to the answer vector
                answer.push_back(distanceToNearestColor);
            }
        }

        return answer;
    }
};
```

Typescript Solution

```typescript
function shortestDistanceColor(colors: number[], queries: number[][]): number[] {
    // Initialize variables
    const numColors = colors.length; // Length of the colors array
    const infinity = 1 << 30; // Large number representing infinity

    // Create two matrices - 'left' and 'right' to store distances to the nearest occurrence of each color
    const right: number[][] = Array.from({ length: numColors + 1 }, () => Array(3).fill(infinity));
    const left: number[][] = Array.from({ length: numColors + 1 }, () => Array(3).fill(-infinity));

    // Populate the 'right' matrix with distances to the nearest color occurrence from the right side
    for (let i = numColors - 1; i >= 0; --i) {
        for (let colorIndex = 0; colorIndex < 3; ++colorIndex) {
            right[i][colorIndex] = right[i + 1][colorIndex];
        }
        right[i][colors[i] - 1] = i;
    }

    // Populate the 'left' matrix with distances to the nearest color occurrence from the left side
    for (let i = 1; i <= numColors; ++i) {
        for (let colorIndex = 0; colorIndex < 3; ++colorIndex) {
            left[i][colorIndex] = left[i - 1][colorIndex];
        }
        left[i][colors[i - 1] - 1] = i - 1;
    }

    // Answer array to store the shortest distances for all queries
    const answers: number[] = [];

    // Calculate the shortest distance for each query
    for (const [index, color] of queries) {
        const distance = Math.min(index - left[index + 1][color - 1], right[index][color - 1] - index);
        // Add the calculated distance to the answers array, -1 if the color does not exist
        answers.push(distance > numColors ? -1 : distance);
    }

    // Return the result
    return answers;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by the operations performed:

1. The first loop iterates backward from `n - 1` to `0`, which is `O(n)`, where `n` is the length of the `colors` list. Inside this loop, there is another loop that runs 3 times (for each color), which is `O(1)`. So, combined this is `O(n)`.
2. The second loop iterates from `1` to `n`, which is `O(n)`. Similar to the first loop, it has an inner loop that runs 3 times, so overall this is also `O(n)`.
3. The last loop processes the `queries`. For each query, the code performs constant time comparisons to calculate the distance, which is `O(1)`. If there are `q` queries, this loop is `O(q)`.

Therefore, the total time complexity is the sum of these three loops: `O(n) + O(n) + O(q) = O(2n + q)`. Since constants are dropped in Big O notation, the simplified time complexity is `O(n + q)`.

Space Complexity

The space complexity is determined by the additional space used:

1. Two two-dimensional lists (`right` and `left`) are created, each with dimensions `(n + 1) x 3`. Thus, the space occupied by these lists is `2 x (n + 1) x 3`, which simplifies to `O(6n + 6)`.
2. The space for the output list `ans` depends on the number of queries `q`, which makes it `O(q)`.

Combined, the total space complexity is `O(6n + 6 + q)`. After removing constants and non-dominant terms, the space complexity simplifies to `O(n + q)`.