276. Paint Fence

Medium Dynamic Programming Leetcode Link

Problem Description

You are tasked with painting a fence that has `n` posts using `k` different colors. The challenge is to abide by two main rules while painting:

1. Each fence post must be painted with exactly one color.
2. You are not allowed to paint three or more consecutive fence posts with the same color.

The problem requires you to calculate the total number of different ways you can paint the fence adhering to the above rules. `n` represents the total number of posts and `k` represents the number of colors you have at your disposal.

Intuition

The solution to this problem can be approached using dynamic programming because the way you paint the current post is dependent on how you painted the previous ones. The key insight is that the number of ways to paint the last post depends on the color of the second to last post.

Let's define two scenarios:

1. The last two posts have the same color.
2. The last two posts have different colors.

Given that we cannot paint three consecutive posts with the same color, if the last two posts are of the same color, the current post can only be painted with any of the `k-1` remaining colors. If the last two posts have different colors, then the current post can also be painted with any of the `k-1` remaining colors. However, network need to consider different ways to reach these scenarios.

Thus, we define a dynamic programming table `dp` where `dp[i][0]` represents the number of ways to paint up to post `i` where the last two posts are of different colors, and `dp[i][1]` represents the number of ways where the last two posts are the same.

The recurrence relations based on the defined scenarios are as follows:

- `dp[i][0]` (different colors) = (`dp[i - 1][0]` + `dp[i - 1][1]`) * (k - 1)
- `dp[i][1]` (same colors) = `dp[i - 1][0]`

The reason behind these relations is that, for `dp[i][0]`, you have both previous scenarios possible and you cannot use the last color used, hence `k - 1` options. For `dp[i][1]`, you can only come from the scenario where previous colors are different, and you must use the same color as the last one, hence only one option available.

The initial conditions are:

- `dp[0][0]` = `k` (because the first post can be any of the `k` colors)
- `dp[0][1]` is not applicable because there is no previous post

We iterate through the posts, calculating the number of ways based on the above recurrence relations. Finally, the answer is the sum of ways for the last post being either the same or different colors as the one before it, i.e., `sum(dp[-1])`.

Solution Approach

The given solution implements the intuition using a dynamic programming (DP) approach. The essential components of this solution are:

1. **Data Structure:** A 2D list `dp` with `n` rows and 2 columns, used to store the number of ways to paint up to the current post. Here, `n` is the number of fence posts. The first column (`dp[i][0]`) stores the number of ways if the last two posts have different colors, and the second column (`dp[i][1]`) stores the number if they are the same.

2. **Initialization:** The DP table is initialized with `dp[0][0] = k`, representing the `k` ways to paint the first fence. Since there is no post before the first one, we don't need to initialize `dp[0][1]` as it is not applicable.

3. **Iteration:** The code iterates over each post from 1 to `n-1`. For each post, the following calculations are performed:

   - `dp[i][0] = (dp[i - 1][0] + dp[i - 1][1]) * (k - 1)`: The last two posts can be either different or the same, but the current post must be a different color than the last one, hence `(k - 1)` choices.
   - `dp[i][1] = dp[i - 1][0]`: The last two posts must be of different colors for the current post to have the same color as the last one. There is exactly one way to paint it, which is the same color as the previous post.

4. **Final Calculation:** After filling the DP table up to `n` posts, the total ways to paint the fence can be found by adding the values of the last row of the DP table: `sum(dp[-1])`. This is because the final post can either be painted the same or a different color from the one before it, and we are interested in all possible valid combinations.

The core of this algorithm lies in understanding the constraints and how they impact the subsequences and the transitions between states in the dynamic programming table. This solution maintains constant space complexity for each post with respect to the number of colors, making the overall space complexity `O(n)`. The time complexity is `O(n)` as well because we compute the entries of the DP table with constant time operations for each of the `n` posts.

The final result is returned by summing the two scenarios in the last row, which provides the count of all the ways to paint the fence according to the rules.

Example Walkthrough

Let's walk through an example to illustrate the solution approach using the given problem and intuition. Suppose we have `n = 3` posts to paint and `k = 2` different colors. We want to find out how many ways we can paint the fence.

We initialize a DP table `dp` with dimensions `[n][2]` where `n` is the number of fence posts. Each entry `dp[i][0]` will store ways we can paint up to the `i-th` post with the last two posts having different colors, and `dp[i][1]` will store ways with the last two posts having the same color.

Step 1: Initialization

For the first fence post (`i = 0`), we have `k` options, assuming `k` is not zero. So, `dp[0][0] = k` since there is no previous post, and the condition for `dp[0][1]` is not applicable.

For `k = 2`, the initialization would be `dp[0][0] = 2` (we have 2 ways to paint the first post as there are 2 colors).

Step 2: Iteration for the second post

Now, let's move to the second post `i = 1`. We have two scenarios:

- `dp[1][0] = (dp[0][0] + dp[0][1]) * (k - 1)` since we can paint the second post with a different color than the first in `k - 1` ways. Here, `dp[0][1]` can be considered 0 because it's not applicable. So, `dp[1][0] = (2 + 0) * (2 - 1) = 2`.
- `dp[1][1] = dp[0][0]` since the last two posts can be the same if the first post was unique, which is already counted in `dp[0][0]`. Therefore, `dp[1][1] = 2`.

At this point, our DP table for `i = 1` looks like this:

```
dp = [
  [2, X],  // X denotes non-applicable
  [2, 2],
]
```

Step 3: Iteration for the third post

For the third post `i = 2`, we follow a similar procedure:

- `dp[2][0] = (dp[1][0] + dp[1][1]) * (k - 1)` which translates to `(2 + 2) * (2 - 1) = 4`. We have 4 ways to paint the third post with a different color than the second post.
- `dp[2][1] = dp[1][0]` since again, the last two can be the same only if the previous two were different. So we take the value from `dp[1][0]` which is 2, giving us `dp[2][1] = 2`.

Now, our DP table for `i = 2` looks like this:

```
dp = [
  [2, X],
  [2, 2],
  [4, 2],
]
```

Step 4: Final Calculation

After computing all the values, we want the sum of the last row to get the total number of ways to paint the fence according to the problem's rules. `sum(dp[2]) = dp[2][0] + dp[2][1] = 4 + 2 = 6`.

Thus, there are 6 different ways to paint the 3 posts using 2 colors while following the given rules. This completes the example walk-through using the dynamic programming solution approach described in the content.

Python Solution

```python
class Solution:
    def numWays(self, n: int, k: int) -> int:
        # Initialize the dynamic programming table
        # dp[i][0] stores the number of ways to paint i+1 fences such that the last two have different colors
        # dp[i][1] stores the number of ways to paint i+1 fences such that the last two have the same color
        dp = [[0] * 2 for _ in range(n)]

        # Base case: The first fence can be painted in k ways
        dp[0][0] = k

        # Populate the dynamic programming table
        for i in range(1, n):
            # The number of ways to paint i+1 fences with the last two being of different colors
            # is the sum of:
            # 1. The number of ways to paint i fences in either same or different colors
            # and then paint the (i+1)th fence in k-1 different colors.
            dp[i][0] = (dp[i - 1][0] + dp[i - 1][1]) * (k - 1)

            # The number of ways to paint i+1 fences with the last two being of the same color
            # is simply the number of ways to paint i fences with different colors
            # (because we can't have more than 2 fences in a row with the same color).
            dp[i][1] = dp[i - 1][0]

        # The answer is the sum of ways to paint n fences with the last two being of the same or different colors
        return sum(dp[n-1])

```

Java Solution

```java
class Solution {
    public int numWays(int n, int k) {
        // Create a 2D array to use as a dynamic programming table
        // dp[i][0] represents the number of ways to paint the fence up to post i
        // without repeating colors on the last two posts
        // dp[i][1] represents the number of ways to paint the fence up to post i
        // with the same color on the last two posts
        int[][] dp = new int[n][2];

        // Base case initialization:
        // There are k ways to paint the first post (since it has no previous post to consider)
        dp[0][0] = k;

        // Iterate over the fence posts starting from the second post
        for (int i = 1; i < n; ++i) {
            // Calculate the number of ways to paint the current post without repeating colors
            // This is done by multiplying the total number of ways to paint the previous post
            // by (k - 1), since we can choose any color except the one used on the last post
            dp[i][0] = (dp[i - 1][0] + dp[i - 1][1]) * (k - 1);

            // Calculate the number of ways to paint the current post using the same color
            // as the last post. This can only be done if the last two posts have different colors,
            // so we use the value from dp[i - 1][0].
            dp[i][1] = dp[i - 1][0];
        }

        // Return the total number of ways to paint the entire fence with n posts by summing
        // the ways to paint with the same color and with different colors on the last two posts
        return dp[n - 1][0] + dp[n - 1][1];
    }
}

```

C++ Solution

```cpp
#include <vector> // Include vector header for using std::vector

class Solution {
public:
    int numWays(int n, int k) {
        // Check for the base case where no fence posts are to be painted
        if (n == 0) {
            return 0;
        }

        // Initialize dynamic programming table with two columns:
        // dp[post][0] is the number of ways to paint the post such that it's a different color than the previous one
        // dp[post][1] is the number of ways to paint the post the same color as the previous one
        std::vector<std::vector<int>> dp(n, std::vector<int>(2));

        // Base case: Only one way to paint the first post with k different colors
        dp[0][0] = k;

        // Fill the dp table
        for (int post = 1; post < n; ++post) {
            // The number of ways to paint the current post a different color than the previous post
            // is the sum of the ways to paint the previous post (either same or different color)
            // times the number of colors left (k - 1)
            dp[post][0] = (dp[post - 1][0] + dp[post - 1][1]) * (k - 1);

            // The number of ways to paint the current post the same color as the previous post
            // is the number of ways the previous post was painted with a different color than its previous one
            // Important: This is limited to one choice to ensure we don't break the rule of not having more than two
            // adjacent posts with the same color.
            dp[post][1] = dp[post - 1][0];
        }

        // Return the sum of the two possibilities for the last fence post
        return dp[n - 1][0] + dp[n - 1][1];
    }
};

```

Typescript Solution

```typescript
// Import the Array type from TypeScript standard library for creating arrays
// (The actual import statement is not needed in TypeScript for Array as it's globally available)

// Function to calculate the number of ways to paint the fence
function numWays(n: number, k: number): number {
    // Handle the base case where there are no fence posts to paint
    if (n === 0) {
        return 0;
    }

    // Initialize a dynamic programming array with two values for each fence post:
    // dp[post][0] represents the ways to paint the post a different color than the previous one
    // dp[post][1] represents the ways to paint the post the same color as the previous one
    let dp: number[][] = new Array(n).fill(0).map(() => [0, 0]);

    // Base case: There is only one way to paint the first post with any of k colors
    dp[0][0] = k;

    // Iterate over the fence posts to fill the dynamic programming array
    for (let post = 1; post < n; post++) {
        // The ways to paint the current post a different color than the previous
        // are the sum of the ways to paint the previous post (either same or different color)
        // multiplied by (k - 1) to account for the remaining color choices
        dp[post][0] = (dp[post - 1][0] + dp[post - 1][1]) * (k - 1);

        // The ways to paint the current post the same color as the previous
        // are equal to the ways to paint the previous post a different color
        // to avoid having more than two consecutive posts with the same color
        dp[post][1] = dp[post - 1][0];
    }

    // Return the sum of the two scenarios for the last post
    return dp[n - 1][0] + dp[n - 1][1];
}

```

Time and Space Complexity

The given Python code is a dynamic programming solution for a problem where we want to calculate the number of ways to paint a fence with `n` posts using `k` colors, such that no more than two adjacent fence posts have the same color.

The time complexity of the code is `O(n)`, where `n` is the number of fence posts. This is because there is a single for-loop that iterates from `1` to `n-1`, with each iteration performing a constant number of operations.

The space complexity of the code is also `O(n)`, since it uses a 2D list `dp` with size `n x 2` to store the number of ways to paint the fence for each post, considering whether the current post has the same color as the previous post or not (`0` for different, `1` for the same).