1411. Number of Ways to Paint N × 3 Grid

Hard Dynamic Programming Leetcode Link

Problem Description

The problem presents a painting task where one has a grid of size `n x 3`, meaning there are `n` rows and `3` columns. The objective is to paint each cell with one of three colors: Red, Yellow, or Green. The constraint is that no two adjacent cells (those that are directly beside or above/below each other) can be the same color. You are asked to calculate the number of different ways to paint the entire grid that satisfies this constraint.

Given the value of `n`, the number of rows in the grid, your task is to find the total number of valid painting configurations, modulo `10^9 + 7`. This modulo operation is a common requirement in programming challenges to avoid large integer overflow issues.

Calculating the number of possible configurations manually is not feasible as `n` can be very large, so we have to come up with an efficient algorithm to compute the result.

Intuition

To arrive at the solution approach, let's understand that we only need to focus on a row-by-row basis since each row's coloring only depends on the row immediately above it. For a given row, two scenarios need to be considered: one where all three cells have different colors from their corresponding cells in the row above, and one where exactly one cell has the same color as the cell above it (forming a 'pattern'). These scenarios are seen as follows:

- Pattern A: No cell has the same color as the one above it. There are three different ways the first cell could be painted. For each choice of the color in the first cell, there are two choices for the second cell, and finally, two choices for the third cell (since it has to be different from both the first cell and the cell above it in the previous row). Hence, there are `3 * 2 * 2` ways to paint row `n` if row `n-1` is of Pattern A.
- Pattern B: Exactly one cell has the same color as the one above it. This again allows for two choices for one of the other cells and two more choices for the remaining cells, resulting in a total of `2 * 2 * 1` ways to paint row `n` if row `n-1` is of Pattern B.

We then calculate the number of ways to paint based on the above observations using dynamic programming. Instead of storing the whole grid, we maintain two variables `f0` and `f1`, representing the count of configurations following Pattern A and Pattern B for the current row being processed. These are being recalculated for every row based on the previous row's counts. For the base case (first row), there are `6` ways to paint following both patterns. The variables `g0` and `g1` are the new counts for the next row.

Each iteration updates `f0` and `f1`, simulating painting the next row until all rows have been 'processed'. The modulo operation ensures the number stays within the integer range specified by the problem.

Finally, the answer is the sum of `f0` and `f1` after processing all rows, as this represents all possible configurations after painting the last row. This sum is also taken modulo `10^9 + 7` to conform to the problem's requirements.

Solution Approach

The implementation uses a dynamic programming approach to solve the problem efficiently. Here's a step-by-step explanation of how the code operates:

1. Define a modulo constant `mod` that equals `10^9 + 7`. This value will be used to perform all calculations under the modulo to prevent integer overflow.
2. Initialize two variables, `f0` and `f1`, with the value `6`. `f0` corresponds to the count of ways to paint a row following Pattern A (no cell has the same color as the one above it) and `f1` corresponds to the count of ways to paint following Pattern B (exactly one cell has the same color as the one above it). This initialization is for the base case, the first row, which can be freely painted in `3 * 2 * 2 = 6` ways for both patterns.
3. Iterate over the range `n - 1` which represents the remaining rows that need to be painted after the first one.

   a. Compute `g0`, which is the new count for Pattern A for the next row. The formula used is `g0 = (3 * f0 + 2 * f1) % mod`. It derives from the two ways a Pattern A configuration can be followed by another Pattern A configuration (3 possibilities), and the two ways a Pattern B configuration can lead to a Pattern A configuration in the subsequent row (2 possibilities).

   b. Compute `g1`, which is the new count for Pattern B. The formula is `g1 = (2 * f0 + 2 * f1) % mod`. This considers that a Pattern A configuration can lead to a Pattern B in the next row (2 possibilities), and that a Pattern B can lead to another Pattern B in the next row (2 possibilities).

   c. Update `f0` and `f1` to be the new counts `g0` and `g1`, respectively. This prepares the algorithm to compute the counts for the next row.
4. After the loop, the total number of valid configurations is the sum of both `f0` and `f1`, which we also take modulo `mod`. This represents all the valid ways to paint the grid after considering all rows.

Here's an explanation of the mathematical formulas used in the code:

- For Pattern A (`g0`), each new row can come from either a Pattern A or Pattern B in the previous row and you have, respectively, 3 or 2 ways to continue the pattern without violating the color constraints.
- For Pattern B (`g1`), each new row can also come from a previous Pattern A or Pattern B, and there are two ways to paint the row for each case.

The use of dynamic programming in this problem allows us to solve it efficiently by breaking it down into subproblems (row-wise computation) and building up the solution from there.

Finally, the code returns `f0 + f1` modulo `mod` as the final answer.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose `n = 2`, which means we have a grid with 2 rows and 3 columns, and we want to find the number of ways to paint this grid according to the given rules.

For the first row, we don't have any restrictions since there's no row above it. The first cell can be painted with any of the three colors: Red, Yellow, or Green. For simplicity, let's denote these colors by R, Y, and G. After choosing a color for the first cell, we have two options for the second cell, and again two options for the third cell, because adjacent cells cannot be the same color.

Here are the possible ways to paint the first row following Pattern A:

1. RYG
2. RGY
3. YRG
4. YGR
5. GRG
6. GRY

As the problem statement outlined, there are 6 ways to paint the first row. For Pattern A (`f0`) and Pattern B (`f1`), the initial count is `6` because both patterns will have the same number of ways to paint the first row.

Now let's move on to the second row. We must consider the two patterns for calculating the possibilities for the second row:

For Pattern A (`g0`, meaning no cell shares a color with the one above it), each Pattern A configuration in the first row can lead to 3 configurations in the second row, while each Pattern B configuration can lead to 2 configurations in the second row because one color is already fixed for one cell.

For Pattern B (`g1`, meaning exactly one cell shares the color with the one above it), each Pattern A or B configuration in the first row can result in 2 configurations in the second row.

Using the formulae from the solution:

For the second row:

- New count for Pattern A (`g0`) = (3 * f0 + 2 * f1) % mod
- New count for Pattern B (`g1`) = (2 * f0 + 2 * f1) % mod

Since `f0` and `f1` are both initialized as `6`:

- `g0` = (3 * 6 + 2 * 6) % mod = (18 + 12) % mod = 30 % mod
- `g1` = (2 * 6 + 2 * 6) % mod = (12 + 12) % mod = 24 % mod

(Note: For this example, we are not applying the modulo operation because the numbers are small, but in the code, we would take modulo `10^9 + 7` to prevent overflow.)

After updating `f0` to `g0` and `f1` to `g1`, we proceed to calculate the total number of configurations for `n = 2`:

- Total configurations = (f0 + f1) % mod
- Total configurations = (30 + 24) % mod
- Total configurations = 54 % mod (which is simply 54 since it's less than `10^9 + 7`)

Thus, for a grid of size `2 x 3`, there are 54 different ways to paint it according to the rules.

This example clearly demonstrates how the dynamic programming approach leads to an efficient solution for larger values of `n` by calculating the number of ways for each row based on the number of ways for the previous row, updating the counts in the process, and ultimately summing up the configurations for the last row.

Python Solution

```python
class Solution:
    def numOfWays(self, n: int) -> int:
        # Define the modulo value to ensure the result is within the range.
        modulus = 10**9 + 7

        # Initialize the number of ways to paint a single row when the first two columns are of the same color (type 0)
        # and when the first two columns are of different colors (type 1). Both are 6 for n = 1.
        num_ways_type_0 = num_ways_type_1 = 6

        # Iteratively calculate the number of ways to paint each row based on the number of ways to paint the previous row.
        for _ in range(n - 1):
            # Calculate the number of ways to paint the current row given the last row is of type 0.
            new_ways_type_0 = (3 * num_ways_type_0 + 2 * num_ways_type_1) % modulus

            # Calculate the number of ways to paint the current row given the last row is of type 1.
            new_ways_type_1 = (2 * num_ways_type_0 + 2 * num_ways_type_1) % modulus

            # Update the number of ways for the next iteration.
            num_ways_type_0, num_ways_type_1 = new_ways_type_0, new_ways_type_1

        # Return the total number of ways to paint the n rows, sum of both types 0 and 1.
        return (num_ways_type_0 + num_ways_type_1) % modulus

```

Java Solution

```java
class Solution {

    // Method to calculate the number of ways to paint a grid
    public int numOfWays(int n) {
        final int MOD = (int) 1e9 + 7; // Define the modulo value for avoiding integer overflow

        // Initial counts of color combinations for a single row, f0 for one pattern type, f1 for the other pattern type
        long countPatternTypeA = 6, countPatternTypeB = 6;

        // Iterate through the grid rows, starting from the second row
        for (int i = 0; i < n - 1; ++i) {
            // Calculate the new count for pattern type A using the previous counts
            long newCountPatternTypeA = (3 * countPatternTypeA + 2 * countPatternTypeB) % MOD;
            // Calculate the new count for pattern type B using the previous counts
            long newCountPatternTypeB = (2 * countPatternTypeA + 2 * countPatternTypeB) % MOD;

            // Update the counts for the next iteration
            countPatternTypeA = newCountPatternTypeA;
            countPatternTypeB = newCountPatternTypeB;
        }

        // Return the total count (sum of both pattern types), ensuring the result is modulo-ed
        return (int) (countPatternTypeA + countPatternTypeB) % MOD;
    }
}

```

C++ Solution

```cpp
using ll = long long;  // Define long long as 'll' for convenience.

class Solution {
public:
    int numOfWays(int n) {
        const int MOD = 1e9 + 7;  // Define the modulus for calculating the result.

        ll patternsWithThreeColors = 6,     // Initialize the number of ways to paint using three colors.
           patternsWithTwoColors = 6;      // Initialize the number of ways to paint using two colors.

        // Loop through the number of tiles left after the first tile.
        while (--n) {
            // Calculate new ways to paint using three colors.
            ll newPatternsWithThreeColors = (patternsWithThreeColors * 3 + patternsWithTwoColors * 2) % MOD;

            // Calculate new ways to paint using two colors.
            ll newPatternsWithTwoColors = (patternsWithThreeColors * 2 + patternsWithTwoColors * 2) % MOD;

            // Update the variable to the latest calculation.
            patternsWithThreeColors = newPatternsWithThreeColors;
            patternsWithTwoColors = newPatternsWithTwoColors;
        }
        // Return the total number of ways to paint the tiles, within the modulus.
        return (int) (patternsWithThreeColors + patternsWithTwoColors) % MOD;
    }
};

```

Typescript Solution

```typescript
// Define modulus for calculating the result.
const MOD: number = 1e9 + 7;

// Define initial count of ways to paint using three colors.
let patternsWithThreeColors: number = 6;

// Define initial count of ways to paint using two colors.
let patternsWithTwoColors: number = 6;

function numOfWays(n: number): number {
    // Iterate through the number of tiles, excluding the first one.
    while (--n) {
        // Calculate new count of ways to paint using three colors.
        let newPatternsWithThreeColors: number = (patternsWithThreeColors * 3 + patternsWithTwoColors * 2) % MOD;

        // Calculate new count of ways to paint using two colors.
        let newPatternsWithTwoColors: number = (patternsWithThreeColors * 2 + patternsWithTwoColors * 2) % MOD;

        // Update counts to the latest calculations.
        patternsWithThreeColors = newPatternsWithThreeColors;
        patternsWithTwoColors = newPatternsWithTwoColors;
    }
    // Return the total count of ways to paint the tiles modulo `MOD`.
    return (patternsWithThreeColors + patternsWithTwoColors) % MOD;
}

```

Time and Space Complexity

The given Python code represents a dynamic programming solution where `n` is the number of rows of a 3x`n` grid to be painted in two colors with certain constraints.

Time Complexity:

The time complexity of the code can be determined by analyzing the for loop:

- The loop runs exactly `n-1` times.
- Inside the loop, it performs a constant number of operations (simple arithmetic and modulo).
- Since the number of operations inside the loop does not depend on `n` and is constant, the total number of operations is proportional to `n`.

Therefore, the time complexity is `O(n)`.

Space Complexity:

The space complexity of the code can be analyzed as follows:

- The variables `f0`, `f1`, `g0`, `g1`, and `mod` use a fixed amount of space regardless of the input size.
- No additional data structures (such as arrays or matrices) that grow with input size are being used.

Thus, the space complexity is `O(1)`.