256. Paint House

Medium Array Dynamic Programming

Problem Description

In this problem, we are given a scenario where we have n houses in a row, and we need to paint each house in such a way that no two adjacent houses have the same color. There are three colors available: red, blue, and green. The cost of painting each house with each color is different and is provided to us in the form of a $n \times 3$ cost matrix - `costs`.

Each row in the `costs` matrix corresponds to a house, and each column to a color (where the first column is red, the second is blue, and the third is green). For instance, `costs[0][0]` represents the cost of painting the first house red, while `costs[1][2]` represents the cost of painting the second house green.

Our goal is to find the minimum total cost to paint all the houses while making sure that no two adjacent houses are painted the same color.

Intuition

The key to solving this problem lies in understanding that the color choice for each house depends on the color choices made for the previous house. To tackle this, we can use dynamic programming to keep track of the minimum cost up to each house while adhering to the color restriction.

We initialize three variables: a, b, and c, representing the minimum cost of painting the last house with one of the three colors. As we iterate through each house, we update the costs for each color:

- a will be updated to the cost of painting the current house red plus the minimum cost of painting the previous house either blue or green (whichever is lesser).
- Similarly, b will be updated to the cost of painting the current house blue plus the minimum cost of painting the previous house either red or green.
- c will be updated to the cost of painting the current house green plus the minimum cost of painting the previous house either red or blue.

In this way, each choice takes into consideration the previous choices and ensures that no two adjacent houses will have the same color, as we always pick a different color for the current house than the one picked for the previous house (captured in a, b, or c).

Finally, after going through all the houses, we will have the minimum cost accumulated in a, b, and c for ending with each color. The answer will be the minimum of these three values as it represents the least cost of painting all houses while satisfying the constraint.

Solution Approach

The implementation of the solution follows the dynamic programming strategy since it involves breaking the problem down into subproblems, solving each subproblem just once, and storing their solutions – typically using a memory-based data structure (array, map, etc.).

Here is a step-by-step breakdown of the algorithm used:

1. **Initialization**: We start by initializing three variables, a, b, and c to 0. These variables will hold the minimum painting cost for the last painted house for each color (red, blue, and green respectively).
2. **Iterating Over Houses**: We then iterate over each house. For each house, we get `ca`, `cb`, and `cc` from the `costs` matrix, which represent the painting costs for red, blue, and green, respectively.
3. **Updating Costs for Each Color**: The core of the solution is updating the cost for each color for the current house. We do this by calculating the minimum cost for painting the current house with a particular color based on the costs of painting the previous house.
   - To paint the current house red (`ca`), we look for the minimum cost between painting the previous house blue or green, which is `min(b, c)`, and add that to the cost of painting the current house red, `ca`. This updated cost is stored back in a.
   - Similarly, to paint the current house blue (`cb`), we find the minimum cost of a and c, and add `cb` to it, and store this value back in b.
   - To paint the current house green (`cc`), we take the minimum cost from a and b and add `cc` to it, and store this value in c.

   This update ensures that for any house, we choose the color that was not chosen for the last house and had the minimum cost among the remaining colors.

   The code snippet for this step is:

   ```
   a, b, c = min(b, c) + ca, min(a, c) + cb, min(a, b) + cc
   ```

4. **Finding the Minimum Total Cost**: After iterating through all the houses, we have three values a, b, and c, each representing the total minimum cost of painting all houses up to this point, with the last house being red, blue, or green respectively. The final step is to return the smallest value among a, b, and c which is `min(a, b, c)`.

This approach uses a constant amount of extra space (for the variables a, b, c) and thus has a space complexity of O(1). The time complexity is O(n) since we iterate through the list of costs once.

Example Walkthrough

Let's go through an example to illustrate the solution approach in action. Suppose we have the following cost matrix for 3 houses and 3 colors:

```
costs = [[17, 2, 17],
         [16, 16, 5],
         [14, 3, 19]]
```

- House 1: red = 17, blue = 2, green = 17
- House 2: red = 16, blue = 16, green = 5
- House 3: red = 14, blue = 3, green = 19

Following the steps of the described solution:

Step 1: Initialization
We initialize a, b, c to 0.

Step 2: Iterating Over Houses
Start with the first house ($i = 0$):

- `ca = costs[0][0] = 17`
- `cb = costs[0][1] = 2`
- `cc = costs[0][2] = 17`

Step 3: Updating Costs for Each Color Since this is the first house, we simply assign each cost to a, b, and c respectively.

- $a = 17$
- $b = 2$
- $c = 17$

Move to the second house ($i = 1$):

- `ca = costs[1][0] = 16`
- `cb = costs[1][1] = 16`
- `cc = costs[1][2] = 5`

We now update the costs based on the previous house's costs:

- $a = \min(\text{previous } b, c) + \text{current } ca = \min(2, 17) + 16 = 2 + 16 = 18$
- $b = \min(\text{previous } a, c) + \text{current } cb = \min(17, 17) + 16 = 17 + 16 = 33$
- $c = \min(\text{previous } a, b) + \text{current } cc = \min(17, 2) + 5 = 2 + 5 = 7$

Now, $a = 18$, $b = 33$, $c = 7$.

Move to the third house ($i = 2$):

- `ca = costs[2][0] = 14`
- `cb = costs[2][1] = 3`
- `cc = costs[2][2] = 19`

Finally, update the costs for the third house:

- $a = \min(\text{previous } b, c) + \text{current } ca = \min(33, 7) + 14 = 7 + 14 = 21$
- $b = \min(\text{previous } a, c) + \text{current } cb = \min(18, 7) + 3 = 7 + 3 = 10$
- $c = \min(\text{previous } a, b) + \text{current } cc = \min(18, 33) + 19 = 18 + 19 = 37$

After considering the third house, $a = 21$, $b = 10$, $c = 37$.

Step 4: Finding the Minimum Total Cost
We look for the smallest value among the final a, b, and c. Which will be the minimum cost to paint all the houses while no two adjacent houses have the same color:

- `min(a, b, c) = min(21, 10, 37) = 10`

Therefore, the minimum total cost to paint all the houses is $10.

This example neatly demonstrates how the dynamic programming approach keeps track of the minimum cost up to each house and updates the cost during each iteration to ensure the final solution adheres to the problem's constraints.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def minCost(self, costs: List[List[int]]) -> int:
        # Initialize the minimum costs for the three colors
        cost_red = cost_blue = cost_green = 0

        # Iterate over each house and calculate the minimum cost
        # of painting each house, not repeating the color of the adjacent house
        for cost of red, cost of blue, cost of green in costs:
            # Update the cost of painting each house with red, blue, or green
            # by adding the current painting cost to the minimum cost of the
            # other two colors from the previous house
            new_cost_red = min(cost_blue, cost_green) + cost_of_red
            new_cost_blue = min(cost_red, cost_green) + cost_of_blue
            new_cost_green = min(cost_red, cost_blue) + cost_of_green

            # Update the current minimum costs for each color
            cost_red, cost_blue, cost_green = new_cost_red, new_cost_blue, new_cost_green

        # Return the minimum cost among the three colors after
        # completing the calculation for all houses
        return min(cost_red, cost_blue, cost_green)

# Example usage:
# solution = Solution()
# print(solution.minCost([[17,2,17],[16,16,5],[14,3,19]]))
```

Java

```java
class Solution {

    // Function to calculate the minimum cost of painting houses
    // with no two adjacent houses having the same color.
    public int minCost(int[][] costs) {
        // Initialize variables to store the minimum cost of painting up to the current house
        int costRed = 0, costGreen = 0, costBlue = 0;

        // Loop through each house and calculate the minimum cost by considering previous house colors
        for (int[] cost : costs) {
            // Temporary variables to store previous costs before updating
            int prevCostRed = costRed, prevCostGreen = costGreen, prevCostBlue = costBlue;

            // Cost of painting current house red is the minimum cost of previous house painted green or blue plus current cost for r
            costRed = Math.min(prevCostGreen, prevCostBlue) + cost[0];

            // Cost of painting current house green is the minimum cost of previous house painted red or blue plus current cost for g
            costGreen = Math.min(prevCostRed, prevCostBlue) + cost[1];

            // Cost of painting current house blue is the minimum cost of previous house painted red or green plus current cost for b
            costBlue = Math.min(prevCostRed, prevCostGreen) + cost[2];
        }

        // Return the overall minimum cost of painting all houses, which is the minimum of the three colors.
        return Math.min(costRed, Math.min(costGreen, costBlue));
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to calculate the minimum cost of painting houses.
    // No two adjacent houses can have the same color.
    int minCost(vector<vector<int>>& costs) {
        // Initialize the minimum costs of the last house for
        // each color to be zero.
        int lastRed = 0, lastGreen = 0, lastBlue = 0;

        // Iterate over each house.
        for (auto& cost : costs) {
            // Store the previous costs before updating.
            int prevRed = lastRed, prevGreen = lastGreen, prevBlue = lastBlue;

            // Update the minimum cost for the current house when painted red.
            // It is the cost of painting the current house red plus the
            // minimum cost of the previous house when it was not red.
            lastRed = min(prevGreen, prevBlue) + cost[0];

            // Update the minimum cost for the current house when painted green.
            // It is the cost of painting the current house green plus the
            // minimum cost of the previous house when it was not green.
            lastGreen = min(prevRed, prevBlue) + cost[1];

            // Update the minimum cost for the current house when painted blue.
            // It is the cost of painting the current house blue plus the
            // minimum cost of the previous house when it was not blue.
            lastBlue = min(prevRed, prevGreen) + cost[2];
        }

        // After considering all houses, return the minimum of the costs
        // of painting the last house with any of the three colors.
        return min(lastRed, min(lastGreen, lastBlue));
    }
};
```

TypeScript

```typescript
/**
 * Calculates the minimum cost to paint all houses such that no two adjacent houses have the same color.
 * The cost of painting each house with each color is given in an input array.
 *
 * @param {number[][]} paintingCosts - A 2D array where paintingCosts[i][j] represents the cost to paint the i-th house with color j.
 * @return {number} - The minimum cost to paint all houses.
 */
function minCost(paintingCosts: number[][]): number {
    // Initialize variables to store the cumulative costs of painting houses with three different colors.
    let costA: number = 0;
    let costB: number = 0;
    let costC: number = 0;

    // Iterate over each house's painting cost array.
    for (let [costAColor0, costAColor1, costAColor2] of paintingCosts) {
        // Calculate the cumulative cost of painting the current house with each color.
        // For each color, we choose the minimum of the other two colors' previous cumulative costs.
        const newCostA: number = Math.min(costB, costC) + costAColor0;
        const newCostB: number = Math.min(costA, costC) + costAColor1;
        const newCostC: number = Math.min(costA, costB) + costAColor2;

        // Update the cumulative costs for the next iteration.
        costA = newCostA;
        costB = newCostB;
        costC = newCostC;
    }

    // The answer is the smallest of the cumulative costs after painting all houses.
    return Math.min(costA, costB, costC);
}
```

```python
from typing import List

class Solution:
    def minCost(self, costs: List[List[int]]) -> int:
        # Initialize the minimum costs for the three colors
        cost_red = cost_blue = cost_green = 0

        # Iterate over each house and calculate the minimum cost
        # of painting each house, not repeating the color of the adjacent house
        for cost of red, cost of blue, cost of green in costs:
            # Update the cost of painting each house with red, blue, or green
            # by adding the current painting cost to the minimum cost of the
            # other two colors from the previous step
            new_cost_red = min(cost_blue, cost_green) + cost_of_red
            new_cost_blue = min(cost_red, cost_green) + cost_of_blue
            new_cost_green = min(cost_red, cost_blue) + cost_of_green

            # Update the current minimum costs for each color
            cost_red, cost_blue, cost_green = new_cost_red, new_cost_blue, new_cost_green

        # Return the minimum cost among the three colors after
        # completing the calculation for all houses
        return min(cost_red, cost_blue, cost_green)

# Example usage:
# solution = Solution()
# print(solution.minCost([[17,2,17],[16,16,5],[14,3,19]]))
```

Time and Space Complexity

The provided code aims to solve a problem where we have a list of `costs` that represents the cost of painting each house with one of three colors, such that no two adjacent houses have the same color, and we want to find the minimum cost to paint all houses.

Time Complexity

The time complexity of the code is determined by a single loop that iterates over each house exactly once. There are n houses (where n is the length of the `costs` list), and for each house, it performs a constant amount of work:

- Accessing the costs of painting the current house with each of the three colors (`ca`, `cb`, `cc`).
- Calculating the new costs for painting the next house with each color by taking the minimum of the previous two not equal colors (the calculation for a, b, c variables).

Since the loop runs n times and does a constant amount of work each time, the time complexity is $O(n)$.

Space Complexity

The space complexity of the code is determined by the additional memory used by the algorithm, not including the input costs themselves. The code only uses a fixed number of variables a, b, and c to keep track of the minimum costs up to the current house for each color, and these are updated in place.

No additional structures that grow with the size of the input are used. Therefore, the space complexity is constant, or $O(1)$.