265. Paint House II

Hard | Array | Dynamic Programming

Leetcode Link

Problem Description

In this problem, we are given `n` houses in a row and each house can be painted with one of `k` colors. The costs of painting each house with a certain color are provided in a two-dimensional array (matrix) called `costs`. The `costs` matrix has `n` rows (representing each house) and `k` columns (representing each color's cost for that house). For example, `costs[i][j]` gives the cost of painting the `i`-th house with the `j`-th color. The main objective is to determine the minimum total cost to paint all the houses while ensuring that no two adjacent houses are painted the same color.

Intuition

The intuition behind solving this problem lies in breaking it down into a sequence of decisions, where for each house we choose a color. However, we must make this decision considering the cost of the current house as well as ensuring that it doesn't have the same color as the adjacent house. We approach the solution dynamically, by building up the answer as we go along.

- We initialize our answer with the cost of painting the first house, which is simply the costs of painting that house with each color (the first row of our `costs` matrix).
- Then for each subsequent house, we update the cost of painting it with each color by adding the minimum cost of painting the previous house with a different color. This ensures we meet the requirement that no two adjacent houses share the same color.
- To efficiently perform this operation, we maintain a temporary array `g` which stores the new costs calculated for painting the current house. For each color `j`, we find the minimum cost from the array `f` (which stores the costs for the previous house) excluding the `j`-th color.
- We then add the current cost (`costs[i][j]`) to this minimum cost and store it in `g[j]`. At the end of each iteration, we assign `g` to `f`, so that `f` now represents the costs of painting up to the current house.
- After processing all houses, `f` will contain the total costs of painting all houses where the last house is painted with each of the `k` colors. Our answer is the minimum of these costs.

Through dynamic programming, the code optimizes the painting cost by cleverly tracking and updating the costs while satisfying the problem's constraints.

Solution Approach

The implementation of the solution follows a dynamic programming pattern, which is often used to break down a complex problem into smaller, easier-to-manage subproblems.

1. We start with an initialization where we assign the costs of painting the first house with the `k` different colors directly to our first version of the `f` array. This sets up our base case for the dynamic programming solution.

   ```
   f = costs[0][:]
   ```

2. We then iterate over each of the remaining houses `(i)` from the second house (`1`) to the last house `(n - 1)`. For each house, we need to calculate the new costs of painting it with each possible color `(j)`.

   ```
   for i in range(1, n):
       g = costs[i][:]
       ...
   ```

3. For each color `(j)`, we determine the minimum cost of painting the previous house with any color other than `j`. This is done to ensure that the same color is not used for adjacent houses.

   To find that minimum, we iterate through all the possible colors (`h`), ignoring the current color `j`. This inner loop effectively finds the lowest cost to paint the previous house with a different color.

   ```
   for j in range(k):
       t = min(f[h] for h in range(k) if h != j)
       ...
   ```

4. The found minimum cost `t` is then added to the current cost of painting the house `i` with color `j` (`costs[i][j]`). The result is the total cost to get to house `i`, having it painted with color `j`, and is stored in `g[j]`.

   ```
   g[j] += t
   ```

5. Once all colors for the current house have been evaluated, we update the `f` array to be our newly calculated `g`.

   ```
   f = g
   ```

6. After we have finished iterating through all the houses, the `f` array holds the minimum cost to paint all the houses up until the last, with the index representing the color used for the last house.

7. The final step is to find the minimum value within the `f` array since this represents the lowest possible cost for painting all the houses while following the rules. This is the value that is returned as the answer.

   ```
   return min(f)
   ```

The pattern used here leverages dynamic programming's key principle: solve the problem for a small section (the first house, in this case) and then build upon that solution incrementally, tackling a new subproblem in each iteration (each subsequent house). While the problem itself has potentially a very large number of combinations, dynamic programming allows us to consider only the relevant ones and efficiently find the solution.

Example Walkthrough

Let's consider a small example to illustrate the solution approach using the dynamic programming pattern described.

- Imagine we have `n = 3` houses and `k = 3` colors.
- The `costs` matrix provided to us is: `[[1,5,3], [2,9,4], [3,1,5]]`. This means:
  - To paint house 0, it costs 1 for color 0, 5 for color 1, and 3 for color 2.
  - To paint house 1, it costs 2 for color 0, 9 for color 1, and 4 for color 2.
  - To paint house 2, it costs 3 for color 0, 1 for color 1, and 5 for color 2.

Applying the approach:

1. We initialize `f` with the costs to paint the first house, so `f` will be `[1, 5, 3]`.

2. We move on to the second house and create a new temporary array `g`, which initially contains the costs for the second house: `g = costs[1][:]` or `g = [2, 9, 4]`.

3. Next, we find the minimum cost to paint the previous house with a color different from the color we want to use for the current house.

   For `g[0]` (house 1, color 0), we look for the minimum cost of painting house 0 with colors 1 or 2. The minimum of `[5, 3]` is `3`, so `g[0] += 3`, resulting in `g[0] = 5`.

   For `g[1]` (house 1, color 1), we look for the minimum cost of painting house 0 with colors 0 or 2. The minimum of `[1, 3]` is `1`, so `g[1] += 1`, resulting in `g[1] = 10`.

   For `g[2]` (house 1, color 2), we look for the minimum cost of painting house 0 with colors 0 or 1. The minimum of `[1, 5]` is `1`, so `g[2] += 1`, resulting in `g[2] = 5`.

4. Now `g` is updated to `[5, 10, 5]` and we update `f` with `g`. So now `f = g`.

5. We repeat the process for the third house.

   We start with `g = costs[2][:]` or `g = [3, 1, 5]`.

   For `g[0]`, we look for the minimum cost of `f[1]` or `f[2]`, which is the minimum of `[10, 5]` and that is `5`, so `g[0] += 5` resulting in `g[0] = 8`.

   For `g[1]`, we look for the minimum cost of `f[0]` or `f[2]`, which is the minimum of `[5, 5]` and that is `5`, so `g[1] += 5` resulting in `g[1] = 6`.

   For `g[2]`, we look for the minimum cost of `f[0]` or `f[1]`, which is the minimum of `[5, 10]` and that is `5`, so `g[2] += 5` resulting in `g[2] = 10`.

6. Updated `g` for the last house is `[8, 6, 10]`, this becomes new `f`, so `f = [8, 6, 10]`.

7. Finally, we find the minimum cost to paint all houses by taking the minimum of `f`, which is `6`. Therefore, the minimum cost to paint all the houses while ensuring no two adjacent houses have the same color is `6`.

Python Solution

```python
class Solution:
    def minCostII(self, costs: List[List[int]]) -> int:
        # Get the number of houses (n) and the number of colors (k)
        num_houses, num_colors = len(costs), len(costs[0])

        # Initialize the previous row with the costs of the first house
        prev_row_costs = costs[0][:]

        # Iterate through the rest of the houses
        for house_index in range(1, num_houses):
            # Initialize costs for the current row
            curr_row_costs = costs[house_index][:]

            # Iterate through each color for the current house
            for color_index in range(num_colors):
                # Find the minimum cost for the previous house, excluding the current color index
                min_cost_except_current = min(
                    prev_row_costs[color] for color in range(num_colors) if color != color_index
                )

                # Add the minimum cost found to the current color cost
                curr_row_costs[color_index] += min_cost_except_current

            # Update the previous row costs for the next iteration
            prev_row_costs = curr_row_costs

        # Return the minimum cost for painting all houses with the last row of costs
        return min(prev_row_costs)

```

Java Solution

```java
class Solution {
    public int minCostII(int[][] costs) {
        int numHouses = costs.length; // Number of houses is the length of the costs array
        int numColors = costs[0].length; // Number of colors available for painting is the length of the first item in costs array
        int[] previousCosts = costs[0].clone(); // Clone the first house's cost as the starting point

        // Iterate over each house starting from the second house
        for (int houseIndex = 1; houseIndex < numHouses; ++houseIndex) {
            int[] currentCosts = costs[houseIndex].clone(); // Clone the current house's costs
            // Iterate through each color for the current house
            for (int colorIndex = 0; colorIndex < numColors; ++colorIndex) {
                int minCost = Integer.MAX_VALUE;
                // Find the minimum cost of painting the previous house with a different color
                for (int prevColorIndex = 0; prevColorIndex < numColors; ++prevColorIndex) {
                    // Skip if it is the same color as the current one
                    if (prevColorIndex != colorIndex) {
                        minCost = Math.min(minCost, previousCosts[prevColorIndex]);
                    }
                }
                // Add the minimum found cost to paint the previous house to the current house's color cost
                currentCosts[colorIndex] += minCost;
            }
            // Update previousCosts to currentCosts for the next iteration
            previousCosts = currentCosts;
        }

        // Find and return the minimum cost from the last house's painting cost array
        return Arrays.stream(previousCosts).min().getAsInt();
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
#include <climits>
using namespace std;

class Solution {
public:
    int minCostII(vector<vector<int>>& costs) {
        int numHouses = costs.size(); // Number of houses
        int numColors = costs[0].size(); // Number of colors

        // Initialize the first row of the DP table with costs of painting the first house
        vector<int> dpTable = costs[0];

        // Loop over each house starting from the second
        for (int houseIndex = 1; houseIndex < numHouses; ++houseIndex) {
            // Temporary vector to hold the new costs for the current house
            vector<int> newCosts = costs[houseIndex];

            // Loop over each color for the current house
            for (int currentColor = 0; currentColor < numColors; ++currentColor) {
                int minCost = INT_MAX; // Initialize minCost to the largest possible value

                // Find the minimum cost to paint the previous house with a color different from currentColor
                for (int previousColor = 0; previousColor < numColors; ++previousColor) {
                    if (previousColor != currentColor) {
                        minCost = min(minCost, dpTable[previousColor]);
                    }
                }

                // Add the minimum cost found to paint the previous house with the cost to paint the current house
                newCosts[currentColor] += minCost;
            }

            // Move the values in newCosts to dpTable for the next iteration
            dpTable = move(newCosts);
        }

        // Return the minimum element in dpTable (which now contains the minimum cost to paint all houses)
        return *min_element(dpTable.begin(), dpTable.end());
    }
};

```

Typescript Solution

```typescript
function minCostII(costs: number[][]): number {
    const numHouses: number = costs.length; // Number of houses
    const numColors: number = costs[0].length; // Number of colors

    // Initialize the first row of the DP table with costs of painting the first house
    let dpTable: number[] = [...costs[0]];

    // Loop over each house starting from the second
    for (let houseIndex = 1; houseIndex < numHouses; houseIndex++) {
        // Temporary array to hold the new costs for the current house
        const newCosts: number[] = [...costs[houseIndex]];

        // Loop over each color for the current house
        for (let currentColor = 0; currentColor < numColors; currentColor++) {
            let minCost: number = Number.MAX_SAFE_INTEGER; // Initialize minCost to the largest possible safe integer value

            // Find the minimum cost to paint the previous house with a color different from currentColor
            for (let previousColor = 0; previousColor < numColors; previousColor++) {
                if (previousColor !== currentColor) {
                    minCost = Math.min(minCost, dpTable[previousColor]);
                }
            }

            // Add the minimum cost found to paint the previous house with the cost to paint the current house
            newCosts[currentColor] += minCost;
        }

        // Move the values in newCosts to dpTable for the next iteration
        dpTable = newCosts;
    }

    // Return the minimum element in dpTable (which now contains the minimum cost to paint all houses)
    return Math.min(...dpTable);
}

```

Time and Space Complexity

Time Complexity:

The given code iterates over `n` rows, and for each row, it iterates over `k` colors to calculate the minimum cost. Inside the inner loop, there is another loop that again iterates over `k` colors to find the minimum cost of painting the previous house with a different color, avoiding the current one. Therefore, for each of the `n` rows, the code performs `O(k)` operations for each color, and within that, it performs another `O(k)` to find the minimum. This results in a time complexity of `O(n * k * k)`.

Space Complexity:

The space complexity of the code is determined by the additional space used for the `f` and `g` arrays, both of size `k`. No other significant space is being used that scales with the input size. Thus the space complexity is `O(k)`.