256. Paint House

Dynamic Programming

Problem Description

Medium Array

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.

with each color is different and is provided to us in the form of a $n \times 3$ cost matrix - costs.

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

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 <u>dynamic programming</u> 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).

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.

• 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

- 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.

Solution Approach

The implementation of the solution follows the <u>dynamic programming</u> 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

The answer will be the minimum of these three values as it represents the least cost of painting all houses while satisfying the

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).

(array, map, etc.).

constraint.

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

Iterating Over Houses: We then iterate over each house. For each house, we get ca, cb, and cc from the costs matrix, which

previous house.

To paint the current house red (ca), we look for:

cost among the remaining colors.

The code snippet for this step is:

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

represent the painting costs for red, blue, and green, respectively.

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

complexity is O(n) since we iterate through the list of costs once.

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.

This update ensures that for any house, we choose the color that was not chosen for the last house and had the minimum

• 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.

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

and 3 colors: costs = [[17, 2, 17], [16, 16, 5],

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

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

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

• ca = costs[1][0] = 16

Move to the second house (i = 1):

Step 1: Initialization

We initialize a, b, c to 0.

Example Walkthrough

[14, 3, 19]]

• House 1: red = 17, blue = 2, green = 17

Following the steps of the described solution:

b = 2
c = 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.

cb = costs[1][1] = 16
 cc = costs[1][2] = 5
 We now update the

• a = 17

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

 a = min(previous b, c) + current ca = min(2, 17) + 16 = 2 + 16 = 18
- c = min(previous a, b) + current cc = min(17, 2) + 5 = 2 + 5 = 7

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

• ca = costs[2][0] = 14

• cc = costs[2][2] = 19

• cb = costs[2][1] = 3

Finally, update the costs for the third house:

• a = min(previous b, c) + current ca = min(33, 7) + 14 = 7 + 14 = 21

• b = min(previous a, c) + current cb = min(17, 17) + 16 = 17 + 16 = 33

c = min(previous a, b) + current cc = min(18, 33) + 19 = 18 + 19 = 37
 After considering the third house, a = 21, b = 10, c = 37.

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

• b = min(previous a, c) + current cb = min(18, 7) + 3 = 7 + 3 = 10

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

Move to the third house (i = 2):

Solution Implementation

def minCost(self, costs: List[List[int]]) -> int:

cost_red = cost_blue = cost_green = 0

Initialize the minimum costs for the three colors

Iterate over each house and calculate the minimum cost

Update the current minimum costs for each color

Return the minimum cost among the three colors after

completing the calculation for all houses

return min(cost_red, cost_blue, cost_green)

print(solution.minCost([[17,2,17], [16,16,5], [14,3,19]]))

// with no two adjacent houses having the same color.

int costRed = 0, costGreen = 0, costBlue = 0;

public int minCost(int[][] costs) {

for (auto& cost : costs) {

// Store the previous costs before updating.

lastRed = min(prevGreen, prevBlue) + cost[0];

lastGreen = min(prevRed, prevBlue) + cost[1];

lastBlue = min(prevRed, prevGreen) + cost[2];

return min(lastRed, min(lastGreen, lastBlue));

// Iterate over each house's painting cost array.

def minCost(self, costs: List[List[int]]) -> int:

cost_red = cost_blue = cost_green = 0

Initialize the minimum costs for the three colors

Iterate over each house and calculate the minimum cost

for cost_of_red, cost_of_blue, cost_of_green in costs:

other two colors from the previous step

of painting each house, not repeating the color of the adjacent house

Update the cost of painting each house with red, blue, or green

• Accessing the costs of painting the current house with each of the three colors (ca, cb, cc).

cost_red, cost_blue, cost_green = new_cost_red, new_cost_blue, new_cost_green

by adding the current painting cost to the minimum cost of the

new_cost_red = min(cost_blue, cost_green) + cost_of_red

Update the current minimum costs for each color

Return the minimum cost among the three colors after

completing the calculation for all houses

return min(cost_red, cost_blue, cost_green)

new_cost_blue = min(cost_red, cost_green) + cost_of_blue

new_cost_green = min(cost_red, cost_blue) + cost_of_green

for (let [costColorA, costColorB, costColorC] of paintingCosts) {

int prevRed = lastRed, prevGreen = lastGreen, prevBlue = lastBlue;

// Update the minimum cost for the current house when painted red.

// Update the minimum cost for the current house when painted green.

// Update the minimum cost for the current house when painted blue.

* Calculates the minimum cost to paint all houses such that no two adjacent houses have the same color.

// Initialize variables to store the cumulative costs of painting houses with three different colors.

// It is the cost of painting the current house red plus the

// It is the cost of painting the current house green plus the

// minimum cost of the previous house when it was not green.

// It is the cost of painting the current house blue plus the

// minimum cost of the previous house when it was not blue.

// After considering all houses, return the minimum of the costs

// of painting the last house with any of the three colors.

// minimum cost of the previous house when it was not red.

of painting each house, not repeating the color of the adjacent house

cost_red, cost_blue, cost_green = new_cost_red, new_cost_blue, new_cost_green

// Initialize variables to store the minimum cost of painting up to the current house

new_cost_blue = min(cost_red, cost_green) + cost_of_blue

new_cost_green = min(cost_red, cost_blue) + cost_of_green

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

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.

```
class Solution {
    // Function to calculate the minimum cost of painting houses
```

Java

Example usage:

solution = Solution()

Python

from typing import List

class Solution:

```
// 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 1
            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 1
            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 1
            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++
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.
```

* 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 colors are described by the cost to paint all houses. */ function minCost(paintingCosts: number[][]): number {

class Solution:

Example usage:

houses.

solution = Solution()

let costA: number = 0;

let costB: number = 0;

let costC: number = 0;

TypeScript

};

/**

```
// 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 newCostB: number = Math.min(costB, costC) + costColorA;
const newCostB: number = Math.min(costA, costC) + costColorB;
const newCostC: number = Math.min(costA, costB) + costColorC;

// 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);
}

from typing import List
```

```
# print(solution.minCost([[17,2,17], [16,16,5], [14,3,19]]))

Time and Space Complexity
```

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:

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

• 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 0(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 0(1).