# 题目

假如有一排房子，共 n 个，每个房子可以被粉刷成红色、蓝色或者绿色这三种颜色中的一种，你需要粉刷所有的房子并且使其相邻的两个房子颜色不能相同。

当然，因为市场上不同颜色油漆的价格不同，所以房子粉刷成不同颜色的花费成本也是不同的。每个房子粉刷成不同颜色的花费是以一个 n x 3 的正整数矩阵 costs 来表示的。

例如，costs[0][0] 表示第 0 号房子粉刷成红色的成本花费；costs[1][2] 表示第 1 号房子粉刷成绿色的花费，以此类推。

请计算出粉刷完所有房子最少的花费成本。

示例 1：

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

输出: 10

解释: 将 0 号房子粉刷成蓝色，1 号房子粉刷成绿色，2 号房子粉刷成蓝色。

  最少花费: 2 + 5 + 3 = 10。

示例 2：

输入: costs = [[7,6,2]]

输出: 2

提示:

costs.length == n

costs[i].length == 3

1 <= n <= 100

1 <= costs[i][j] <= 20

# 分析

## 方法一：动态规划

**思路：**

第一步，先提出自底部向上的方案———无论题目怎么变换，动态规划都是要老老实实一步步遍历数据。

这里最容易想到的思路就是

cur[n] = min(cur[n - 1]) + cur\_min\_color

然后，发现肯定不行。

第二步，优化当前方案——\*\*存储的上1/N步的重要信息是否找对了\*\*

为什么上一种方案会错误。

万一上个颜色的最小值相同的当前颜色的值，特别小。其他颜色的值特别大[[2,1,3][1,111111,2]] 这个反例肯定就跪了

所以，我们存储的上1/N步的信息需要补充那些呢？（注意有些动态规划的题目，是需要存储不止前一步的信息的）

把上一步的三种颜色的最小值都给存储就OK了。

代码：

class Solution {

public:

int minCost(vector<vector<int>>& costs) {

if (costs.size() == 0) {

return 0;

}

int pre\_red\_min = 0;

int pre\_blue\_min = 0;

int pre\_green\_min = 0;

int min\_result;

for (int i = 0; i < costs.size(); ++i) {

min\_result = INT\_MAX;

int temp\_red\_min = min(pre\_blue\_min, pre\_green\_min) + costs[i][0];

int temp\_blue\_min = min(pre\_red\_min, pre\_green\_min) + costs[i][1];

int temp\_green\_min = min(pre\_red\_min, pre\_blue\_min) + costs[i][2];

min\_result = min(min(min(temp\_green\_min, temp\_red\_min), temp\_blue\_min), min\_result);

pre\_red\_min = temp\_red\_min;

pre\_blue\_min = temp\_blue\_min;

pre\_green\_min = temp\_green\_min;

}

return min\_result;

}

}

另一种写法：

状态转移方程

R[0] = cost[0][0]

B[0] = cost[0][1]

G[0] = cost[0][2]

R[1] = min(B[0], G[0]) + cost[1][0]

B[1] = min(R[0], G[0]) + cost[1][1]

G[1] = min(R[0], B[0]) + cost[1][2]

R[k] = min(B[k-1], G[k-1]) + cost[k][0]

B[k] = min(R[k-1], G[k-1]) + cost[k][1]

G[k] = min(R[k-1], B[k-1]) + cost[k][2]

result = min(R[n-1], B[n-1], G[n-1])

代码

class Solution {

public:

int minCost(vector<vector<int>>& costs) {

int preR = 0; int curR = 0;

int preB = 0; int curB = 0;

int preG = 0; int curG = 0;

for (int i = 0; i < costs.size(); i++) {

curR = min(preB, preG) + costs[i][0];

curB = min(preR, preG) + costs[i][1];

curG = min(preR, preB) + costs[i][2];

preR = curR;

preB = curB;

preG = curG;

}

return min(min(curR, curB), curG);

}

};

总结

这道题关键在于记录三个维度的数据的迭代, 每轮迭代得到的值符合相邻颜色不能相同的要求，稍加修改还可以保存路径信息。

解题思路

（1）状态定义：dp[i][k]: 第i个房子，用第k种颜色, 花费的最小成本

（2）状态转移：相邻的不同颜色的房子才能状态转移;

dp[i][0] = min(dp[i-1][1], dp[i-1][2]) + costs[i-1][0];

dp[i][1] = min(dp[i-1][0], dp[i-1][2]) + costs[i-1][1];

dp[i][2] = min(dp[i-1][0], dp[i-1][1]) + costs[i-1][2];

（3）结果定义：第n个房子，用任意颜色粉刷所需要的最小成本：

min(min(dp[n][0], dp[n][1]), dp[n][2]);

代码

class Solution {

public:

// dp[i][k]: 第i个房子，用第k种颜色, 花费的最小成本

// 状态转移：相邻的不同颜色的房子才能状态转移:

int minCost(vector<vector<int>>& costs) {

int n = costs.size();

vector<vector<int>> dp(n+1, vector<int>(3, 0));

for(int i = 1; i <= n; i++){

dp[i][0] = min(dp[i-1][1], dp[i-1][2]) + costs[i-1][0];

dp[i][1] = min(dp[i-1][0], dp[i-1][2]) + costs[i-1][1];

dp[i][2] = min(dp[i-1][0], dp[i-1][1]) + costs[i-1][2];

}

// n个房子

return min(min(dp[n][0], dp[n][1]), dp[n][2]);

}

};

贴个265.粉刷房子II的题解：

https://leetcode-cn.com/problems/paint-house-ii/solution/dp-dpik-fen-shua-i-houses-shi-yong-k-col-en9y/