<https://leetcode.ca/all/256.html>

There are a row of *n* houses, each house can be painted with one of the three colors: red, blue or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a *n* x *3* cost matrix. For example, costs[0][0] is the cost of painting house 0 with color red; costs[1][2] is the cost of painting house 1 with color green, and so on... Find the minimum cost to paint all houses.

**Note:** All costs are positive integers.

Example:

Input: [[17,2,17],

[16,16,5],

[14,3,19]]

Output: 10

Explanation: Paint house 0 into blue, paint house 1 into green, paint house 2 into blue.

Minimum cost: 2 + 5 + 3 = 10.

**Attempt 1: 2023-11-24**

**Solution 1: Native DFS (30 min)**

**Style 1: 顶[0,0],[0,1],[0,2]，底[n - 1,0],[n - 1,1],[n - 1,2](实际上是[n,0],[n,1],[n,2]，因为界条件退出递归的是i == n)**

class Solution {

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

if(costs.length == 0 || costs[0].length == 0) {

return 0;

}

return Math.min(Math.min(helper(costs, 0, 0), helper(costs, 0, 1)), helper(costs, 0, 2));

}

private int helper(int[][] costs, int i, int color) {

if(i == costs.length) {

return 0;

}

int minCost = 0;

if(color == 0) {

minCost = costs[i][color] + Math.min(helper(costs, i + 1, 1), helper(costs, i + 1, 2));

} else if(color == 1) {

minCost = costs[i][color] + Math.min(helper(costs, i + 1, 0), helper(costs, i + 1, 2));

} else if(color == 2) {

minCost = costs[i][color] + Math.min(helper(costs, i + 1, 0), helper(costs, i + 1, 1));

}

return minCost;

}

}

**Style 2: 顶[n - 1,0],[n - 1,1],[n - 1,2]，底[0,0],[0,1],[0,2](实际上是[-1,0],[-1,1],[-1,2]，因为边界条件退出递归的是i < 0)**

class Solution {

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

if(costs.length == 0 || costs[0].length == 0) {

return 0;

}

int n = costs.length;

return Math.min(Math.min(helper(costs, n - 1, 0), helper(costs, n - 1, 1)), helper(costs, n - 1, 2));

}

private int helper(int[][] costs, int i, int color) {

if(i < 0) {

return 0;

}

int minCost = 0;

if(color == 0) {

minCost = costs[i][color] + Math.min(helper(costs, i - 1, 1), helper(costs, i - 1, 2));

} else if(color == 1) {

minCost = costs[i][color] + Math.min(helper(costs, i - 1, 0), helper(costs, i - 1, 2));

} else if(color == 2) {

minCost = costs[i][color] + Math.min(helper(costs, i - 1, 0), helper(costs, i - 1, 1));

}

return minCost;

}

}

**Style 3: 基于Style 1的more general的写法**

class Solution {

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

if(costs.length == 0 || costs[0].length == 0) {

return 0;

}

return Math.min(Math.min(helper(costs, 0, 0), helper(costs, 0, 1)), helper(costs, 0, 2));

}

private int helper(int[][] costs, int i, int color) {

if(i == costs.length) {

return 0;

}

return costs[i][color] + Math.min(helper(costs, i + 1, (color + 1) % 3), helper(costs, i + 1, (color + 2) % 3));

}

}

**Solution 2: DFS + Memoization (30 min)**

**Style 1:**

class Solution {

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

if(costs.length == 0 || costs[0].length == 0) {

return 0;

}

Integer[][] memo = new Integer[costs.length + 1][3];

return Math.min(Math.min(helper(costs, 0, 0, memo), helper(costs, 0, 1, memo)), helper(costs, 0, 2, memo));

}

private int helper(int[][] costs, int i, int color, Integer[][] memo) {

if(i == costs.length) {

return 0;

}

if(memo[i][color] != null) {

return memo[i][color];

}

int minCost = 0;

if(color == 0) {

minCost = costs[i][color] + Math.min(helper(costs, i + 1, 1, memo), helper(costs, i + 1, 2, memo));

} else if(color == 1) {

minCost = costs[i][color] + Math.min(helper(costs, i + 1, 0, memo), helper(costs, i + 1, 2, memo));

} else if(color == 2) {

minCost = costs[i][color] + Math.min(helper(costs, i + 1, 0, memo), helper(costs, i + 1, 1, memo));

}

return memo[i][color] = minCost;

}

}

**Style 2:**

class Solution {

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

if(costs.length == 0 || costs[0].length == 0) {

return 0;

}

Integer[][] memo = new Integer[costs.length + 1][3];

int n = costs.length;

return Math.min(Math.min(helper(costs, n - 1, 0, memo), helper(costs, n - 1, 1, memo)), helper(costs, n - 1, 2, memo));

}

private int helper(int[][] costs, int i, int color, Integer[][] memo) {

if(i < 0) {

return 0;

}

if(memo[i][color] != null) {

return memo[i][color];

}

int minCost = 0;

if(color == 0) {

minCost = costs[i][color] + Math.min(helper(costs, i - 1, 1, memo), helper(costs, i - 1, 2, memo));

} else if(color == 1) {

minCost = costs[i][color] + Math.min(helper(costs, i - 1, 0, memo), helper(costs, i - 1, 2, memo));

} else if(color == 2) {

minCost = costs[i][color] + Math.min(helper(costs, i - 1, 0, memo), helper(costs, i - 1, 1, memo));

}

return memo[i][color] = minCost;

}

}

**Style 3: 基于Style 1的more general的写法**

class Solution {

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

if(costs.length == 0 || costs[0].length == 0) {

return 0;

}

Integer[][] memo = new Integer[costs.length + 1][3];

return Math.min(Math.min(helper(costs, 0, 0, memo), helper(costs, 0, 1, memo)), helper(costs, 0, 2, memo));

}

private int helper(int[][] costs, int i, int color, Integer[][] memo) {

if(i == costs.length) {

return 0;

}

if(memo[i][color] != null) {

return memo[i][color];

}

return memo[i][color] = costs[i][color] + Math.min(helper(costs, i + 1, (color + 1) % 3, memo), helper(costs, i + 1, (color + 2) % 3, memo));

}

}

**Refer to**

<https://wentao-shao.gitbook.io/leetcode/dynamic-programming/1.position/256.paint-house>

**Approach #1 DFS With Memoization**

class Solution {

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

if (costs == null || costs.length == 0) return 0;

int[][] memo = new int[costs.length][costs[0].length];

return Math.min(dfs(0, 0, costs, memo),

Math.min(dfs(0, 1, costs, memo), dfs(0, 2, costs, memo)));

}

private int dfs(int n, int color, int[][] costs, int[][] memo) {

if (memo[n][color] > 0) {

return memo[n][color];

}

int totalCosts = costs[n][color];

if (n == costs.length - 1) {

} else if (color == 0) {

totalCosts += Math.min(dfs(n + 1, 1, costs, memo), dfs(n + 1, 2, costs, memo));

} else if (color == 1) {

totalCosts += Math.min(dfs(n + 1, 0, costs, memo), dfs(n + 1, 2, costs, memo));

} else if (color == 2) {

totalCosts += Math.min(dfs(n + 1, 0, costs, memo), dfs(n + 1, 1, costs, memo));

}

memo[n][color] = totalCosts;

return totalCosts;

}

}

**Solution 2: DP + Fibonacci (30 min)**

**Style 1: 匹配顶[0,0],[0,1],[0,2]，底[n-1,0],[n-1,1],[n-1,2]的Native DFS**

**Since only 3 colors means 2nd dimension of dp array as 3, start with 3 different colors make 3 different branches**

class Solution {

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

if(costs.length == 0 || costs[0].length == 0) {

return 0;

}

int n = costs.length;

int[][] dp = new int[n][3];

dp[n - 1][0] = costs[n - 1][0];

dp[n - 1][1] = costs[n - 1][1];

dp[n - 1][2] = costs[n - 1][2];

for(int i = n - 2; i >= 0; i--) {

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

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

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

}

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

}

}

**Style 2: 匹配顶[n - 1,0],[n - 1,1],[n - 1,2]，底[-1,0],[-1,1],[-1,2]的Native DFS**

**Since only 3 colors means 2nd dimension of dp array as 3, start with 3 different colors make 3 different branches**

class Solution {

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

if(costs.length == 0 || costs[0].length == 0) {

return 0;

}

int n = costs.length;

int[][] dp = new int[n + 1][3];

dp[0][0] = costs[0][0];

dp[0][1] = costs[0][1];

dp[0][2] = costs[0][2];

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

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

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

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

}

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

}

}

Time Complexity: O(N)

Space Complexity: O(N\*3)

**Style 3: 基于Style 1的more general的写法**

class Solution {

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

if(costs.length == 0 || costs[0].length == 0) {

return 0;

}

int n = costs.length;

int[][] dp = new int[n + 1][3];

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

for(int j = 0; j < 3; j++) {

dp[i][j] += Math.min(dp[i - 1][(j + 1) % 3], dp[i - 1][(j + 2) % 3]);

}

}

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

}

}

Time Complexity: O(N)

Space Complexity: O(N\*3)

**Refer to**

<https://wentao-shao.gitbook.io/leetcode/dynamic-programming/1.position/256.paint-house>

**Approach #2 Dynamic Programming**

class Solution {

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

if (costs == null || costs.length == 0) return 0;

int n = costs.length;

int[][] dp = new int[n][3];

dp[0][0] = costs[0][0];

dp[0][1] = costs[0][1];

dp[0][2] = costs[0][2];

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

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

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

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

}

return Math.min(dp[n-1][0],

Math.min(dp[n-1][1], dp[n-1][2]));

}

}

**Refer to**

<https://grandyang.com/leetcode/256/>

这道题说有n个房子，每个房子可以用红绿蓝三种颜色刷，每个房子的用每种颜色刷的花费都不同，限制条件是相邻的房子不能用相同的颜色来刷，现在让求刷完所有的房子的最低花费是多少。这题跟 [House Robber II](http://www.cnblogs.com/grandyang/p/4518674.html) 和 [House Robber](http://www.cnblogs.com/grandyang/p/4383632.html) 很类似，不过那题不是每个房子都抢，相邻的房子不抢，而这道题是每个房子都刷，相邻的房子不能刷同一种颜色，而 [Paint Fence](http://www.cnblogs.com/grandyang/p/5231220.html) 那道题主要考察有多少种刷法。这几道题很类似，但都不一样，需要分别区分。但是它们的解题思想都一样，需要用动态规划 Dynamic Programming 来做，这道题需要维护一个二维的动态数组 dp，**其中 dp[i][j] 表示刷到第 i+1 房子用颜色j的最小花费**，状态转移方程为:

dp[i][j] = dp[i][j] + min(dp[i - 1][(j + 1) % 3], dp[i - 1][(j + 2) % 3])；

这个也比较好理解，如果当前的房子要用红色刷，则上一个房子只能用绿色或蓝色来刷，那么要求刷到当前房子，且当前房子用红色刷的最小花费就等于当前房子用红色刷的钱加上刷到上一个房子用绿色和刷到上一个房子用蓝色中的较小值，这样当算到最后一个房子时，只要取出三个累计花费的最小值即可，参见代码如下：

class Solution {

public:

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

if (costs.empty() || costs[0].empty()) return 0;

vector<vector<int>> dp = costs;

for (int i = 1; i < dp.size(); ++i) {

for (int j = 0; j < 3; ++j) {

dp[i][j] += min(dp[i - 1][(j + 1) % 3], dp[i - 1][(j + 2) % 3]);

}

}

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

}

};

由于只有红绿蓝三张颜色，所以就可以分别写出各种情况，这样写可能比上面的写法更加一目了然一些，更容易理解一点吧：

class Solution {

public:

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

if (costs.empty() || costs[0].empty()) return 0;

vector<vector<int>> dp = costs;

for (int i = 1; i < dp.size(); ++i) {

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

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

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

}

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

}

};

**Refer to**

<https://segmentfault.com/a/1190000003903965>

直到房子i，其最小的涂色开销是直到房子i-1的最小涂色开销，加上房子i本身的涂色开销。但是房子i的涂色方式需要根据房子i-1的涂色方式来确定，所以我们对房子i-1要记录涂三种颜色分别不同的开销，这样房子i在涂色的时候，我们就知道三种颜色各自的最小开销是多少了。我们在原数组上修改，可以做到不用空间。

public class Solution {

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

if(costs != null && costs.length == 0) return 0;

for(int i = 1; i < costs.length; i++){

// 涂第一种颜色的话，上一个房子就不能涂第一种颜色，这样我们要在上一个房子的第二和第三个颜色的最小开销中找最小的那个加上

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

// 涂第二或者第三种颜色同理

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

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

}

// 返回涂三种颜色中开销最小的那个

return Math.min(costs[costs.length - 1][0], Math.min(costs[costs.length - 1][1], costs[costs.length - 1][2]));

}

}