<https://leetcode.ca/all/265.html>

There are a row of *n* houses, each house can be painted with one of the *k* colors. 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 *k* cost matrix. For example, costs[0][0] is the cost of painting house 0 with color 0; costs[1][2] is the cost of painting house 1 with color 2, and so on... Find the minimum cost to paint all houses.

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

Example:

Input: [[1,5,3],[2,9,4]]

Output: 5

Explanation: Paint house 0 into color 0, paint house 1 into color 2. Minimum cost: 1 + 4 = 5;

Or paint house 0 into color 2, paint house 1 into color 0. Minimum cost: 3 + 2 = 5.

**Follow up:** Could you solve it in *O*(*nk*) runtime?

**Attempt 1: 2023-11-25**

**Solution 1: Native DFS (10 min)**

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

class Solution {

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

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

return 0;

}

int k = costs[0].length;

int minCost = Integer.MAX\_VALUE;

for(int i = 0; i < k; i++) {

minCost = Math.min(minCost, helper(costs, 0, i, k));

}

return minCost;

}

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

if(i == costs.length) {

return 0;

}

int minVal = Integer.MAX\_VALUE;

for(int j = 1; j < k; j++) {

minVal = Math.min(minVal, helper(costs, i + 1, (color + j) % k, k));

}

return minVal + costs[i][color];

}

}

Time complexity : O(nk^2)

Space complexity : O(nk)

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

class Solution {

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

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

return 0;

}

int n = costs.length;

int k = costs[0].length;

int minCost = Integer.MAX\_VALUE;

for(int i = 0; i < k; i++) {

minCost = Math.min(minCost, helper(costs, n - 1, i, k));

}

return minCost;

}

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

if(i < 0) {

return 0;

}

int minVal = Integer.MAX\_VALUE;

for(int j = 1; j < k; j++) {

minVal = Math.min(minVal, helper(costs, i - 1, (color + j) % k, k));

}

return minVal + costs[i][color];

}

}

Time complexity : O(nk^2)

Space complexity : O(nk)

**Solution 2: DFS + Memoization (10 min)**

**Style 1: 基于Solution 1 Style 1**

class Solution {

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

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

return 0;

}

int k = costs[0].length;

int minCost = Integer.MAX\_VALUE;

Integer[][] memo = new Integer[costs.length][k];

for(int i = 0; i < k; i++) {

minCost = Math.min(minCost, helper(costs, 0, i, k, memo));

}

return minCost;

}

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

if(i == costs.length) {

return 0;

}

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

return memo[i][color];

}

int minVal = Integer.MAX\_VALUE;

for(int j = 1; j < k; j++) {

minVal = Math.min(minVal, helper(costs, i + 1, (color + j) % k, k, memo));

}

return memo[i][color] = minVal + costs[i][color];

}

}

Time complexity : O(nk^2)

Space complexity : O(nk)

**Style 2: 基于Solution 1 Style 2**

class Solution {

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

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

return 0;

}

int n = costs.length;

int k = costs[0].length;

int minCost = Integer.MAX\_VALUE;

Integer[][] memo = new Integer[n][k];

for(int i = 0; i < k; i++) {

minCost = Math.min(minCost, helper(costs, n - 1, i, k, memo));

}

return minCost;

}

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

if(i < 0) {

return 0;

}

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

return memo[i][color];

}

int minVal = Integer.MAX\_VALUE;

for(int j = 1; j < k; j++) {

minVal = Math.min(minVal, helper(costs, i - 1, (color + j) % k, k, memo));

}

return memo[i][color] = minVal + costs[i][color];

}

}

Time complexity : O(nk^2)

Space complexity : O(nk)

**Refer to**

<https://wentao-shao.gitbook.io/leetcode/dynamic-programming/1.position/265.paint-house-ii>

### Approach #1 DFS + Memoization

**Time complexity : O*(*nk^2)**

**Space complexity : O*(*nk)**

class Solution {

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

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

int n = costs.length;

int k = costs[0].length;

int[][] memo = new int[n][k];

int minCost = Integer.MAX\_VALUE;

for (int color = 0; color < k; color++) {

minCost = Math.min(minCost, dfs(costs, memo, 0, color));

}

return minCost;

}

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

// 1. end

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

return costs[houseNumber][color];

}

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

return memo[houseNumber][color];

}

int minCost = Integer.MAX\_VALUE;

for (int nextColor = 0; nextColor < costs[0].length; nextColor++) {

if (color == nextColor) continue;

int curCost = dfs(costs, memo, houseNumber + 1, nextColor);

minCost = Math.min(curCost, minCost);

}

int totalCost = costs[houseNumber][color] + minCost;

memo[houseNumber][color] = totalCost;

return totalCost;

}

}

**Solution 3: DP + Fibonacci (60 min，属于Fibonacci的一种是因为DP开局多个基础状态，而不是决定于单一基础状态)**

**Style 1: 基于Solution 1 Style 1 -> 顶[0,0],[0,1]...[0,k - 1]，底[n - 1,0],[n - 1,1]...[n - 1,k - 1]**

class Solution {

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

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

return 0;

}

int n = costs.length;

int k = costs[0].length;

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

// Initialize dp with house n - 1

for(int color = 0; color < k; color++) {

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

}

// Build dp with house n - 2 to 0

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

for(int color = 0; color < k; color++) {

int minVal = Integer.MAX\_VALUE;

for(int delta = 1; delta < k; delta++) {

minVal = Math.min(minVal, dp[house + 1][(color + delta) % k]);

}

dp[house][color] += costs[house][color] + minVal;

}

}

// Find minimum val among all potential final status

int result = Integer.MAX\_VALUE;

for(int color = 0; color < k; color++) {

result = Math.min(result, dp[0][color]);

}

return result;

}

}

Time complexity : O(nk^2)

Space complexity : O(1)

**Style 2: 基于Solution 1 Style 2 -> 顶[n - 1,0],[n - 1,1]...[n - 1,k - 1]，底[0,0],[0,1]...[0,k - 1]**

class Solution {

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

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

return 0;

}

int n = costs.length;

int k = costs[0].length;

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

// Initialize dp with house 0

for(int color = 0; color < k; color++) {

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

}

// Build dp with house 1 to n - 1

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

for(int color = 0; color < k; color++) {

int minVal = Integer.MAX\_VALUE;

for(int delta = 1; delta < k; delta++) {

minVal = Math.min(minVal, dp[house - 1][(color + delta) % k]);

}

dp[house][color] += costs[house][color] + minVal;

}

}

// Find minimum val among all potential final status

int result = Integer.MAX\_VALUE;

for(int color = 0; color < k; color++) {

result = Math.min(result, dp[n - 1][color]);

}

return result;

}

}

Time complexity : O(nk^2)

Space complexity : O(nk)

**Refer to**

<https://wentao-shao.gitbook.io/leetcode/dynamic-programming/1.position/265.paint-house-ii>

### Approach #2 Dynamic Programming

**Time complexity : O(nk^2)**

**Space complexity : O(1) -> 做到O(1)是因为在原有输入2D array costs上直接改动**

class Solution {

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

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

int n = costs.length;

int k = costs[0].length;

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

for (int color = 0; color < k; color++) {

int min = Integer.MAX\_VALUE;

for (int prevColor = 0; prevColor < k; prevColor++) {

if (color == prevColor) continue;

min = Math.min(min, costs[house - 1][prevColor]);

}

costs[house][color] += min;

}

}

int minCost = Integer.MAX\_VALUE;

for (int cost : costs[n - 1]) {

minCost = Math.min(minCost, cost);

}

return minCost;

}

}

**Solution 4: DP + Fibonacci  Time Optimization (60 min)**

**Style 1: 基于Solution 1 Style 1 -> 顶[0,0],[0,1]...[0,k - 1]，底[n - 1,0],[n - 1,1]...[n - 1,k - 1]**

class Solution {

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

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

return 0;

}

int n = costs.length;

int k = costs[0].length;

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

// Initialize dp with house n - 1

for(int color = 0; color < k; color++) {

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

}

// Build dp with house n - 2 to 0

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

// For each new house(current house), we have to find previous

// house cost, which color get its minimum & second minimum cost

int minColor = -1;

int secMinColor = -1;

for(int color = 0; color < k; color++) {

//int minVal = Integer.MAX\_VALUE;

//for(int delta = 1; delta < k; delta++) {

// minVal = Math.min(minVal, dp[house - 1][(color + delta) % k]);

//}

//dp[house][color] += costs[house][color] + minVal;

int cost = costs[house + 1][color];

if(minColor == -1 || cost < costs[house + 1][minColor]) {

secMinColor = minColor;

minColor = color;

} else if(secMinColor == -1 || cost < costs[house + 1][secMinColor]) {

secMinColor = color;

}

}

for(int color = 0; color < k; color++) {

// Cannot pick the same color as previous house even make the minimum cost,

// we can take the second best

if(color == minColor) {

dp[house][color] += costs[house][color] + dp[house + 1][secMinColor];

} else {

dp[house][color] += costs[house][color] + dp[house + 1][minColor];

}

}

}

// Find minimum val among all potential final status

int result = Integer.MAX\_VALUE;

for(int color = 0; color < k; color++) {

result = Math.min(result, dp[0][color]);

}

return result;

}

}

Time complexity:O(nk)

Space complexity:O(nk)

**Style 2: 基于Solution 1 Style 2 -> 顶[n - 1,0],[n - 1,1]...[n - 1,k - 1]，底[0,0],[0,1]...[0,k - 1]**

class Solution {

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

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

return 0;

}

int n = costs.length;

int k = costs[0].length;

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

// Initialize dp with house 0

for(int color = 0; color < k; color++) {

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

}

// Build dp with house 1 to n - 1

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

// For each new house(current house), we have to find previous

// house cost, which color get its minimum & second minimum cost

int minColor = -1;

int secMinColor = -1;

for(int color = 0; color < k; color++) {

//int minVal = Integer.MAX\_VALUE;

//for(int delta = 1; delta < k; delta++) {

// minVal = Math.min(minVal, dp[house - 1][(color + delta) % k]);

//}

//dp[house][color] += costs[house][color] + minVal;

int cost = costs[house - 1][color];

if(minColor == -1 || cost < costs[house - 1][minColor]) {

secMinColor = minColor;

minColor = color;

} else if(secMinColor == -1 || cost < costs[house - 1][secMinColor]) {

secMinColor = color;

}

}

for(int color = 0; color < k; color++) {

// Cannot pick the same color as previous house even make the minimum cost,

// we can take the second best

if(color == minColor) {

dp[house][color] += costs[house][color] + dp[house - 1][secMinColor];

} else {

dp[house][color] += costs[house][color] + dp[house - 1][minColor];

}

}

}

// Find minimum val among all potential final status

int result = Integer.MAX\_VALUE;

for(int color = 0; color < k; color++) {

result = Math.min(result, dp[n - 1][color]);

}

return result;

}

}

Time complexity:O(nk)

Space complexity:O(nk)

**Refer to**

<https://wentao-shao.gitbook.io/leetcode/dynamic-programming/1.position/265.paint-house-ii>

### Approach #3 Dynamic Programming with Optimized Time

如果最小值是第i个元素，次小值是第j个元素

如果除掉的元素不是第i个，剩下的最小值就是第i个元素

如果除掉的是第i个，剩下的最小值就是第j个元素

Time complexity : O(nk)

Space complexity : O(1) **-> 做到O(1)是因为在原有输入2D array costs上直接改动**

class Solution {

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

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

int k = costs[0].length;

int n = costs.length;

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

int minColor = -1;

int secondMinColor = -1;

for (int color = 0; color < k; color++) {

int cost = costs[house - 1][color];

if (minColor == -1 || cost < costs[house - 1][minColor]) {

secondMinColor = minColor;

minColor = color;

} else if (secondMinColor == -1 || cost < costs[house - 1][secondMinColor]) {

secondMinColor = color;

}

}

for (int color = 0; color < k; color++) {

if (color == minColor) { // 不能相邻

costs[house][color] += costs[house - 1][secondMinColor];

} else {

costs[house][color] += costs[house - 1][minColor];

}

}

}

int minCost = Integer.MAX\_VALUE;

for (int cost: costs[n - 1]) {

minCost = Math.min(minCost, cost);

}

return minCost;

}

}

**Refer to**

<https://grandyang.com/leetcode/265/>

这道题是之前那道 [Paint House](http://www.cnblogs.com/grandyang/p/5319384.html) 的拓展，那道题只让用红绿蓝三种颜色来粉刷房子，而这道题让用k种颜色，这道题不能用之前那题的解法，会 TLE。这题的解法的思路还是用 DP，但是在找不同颜色的最小值不是遍历所有不同颜色，而是用 min1 和 min2 来记录之前房子的最小和第二小的花费的颜色，如果当前房子颜色和 min1 相同，那么用 min2 对应的值计算，反之用 min1 对应的值，这种解法实际上也包含了求次小值的方法，感觉也是一种很棒的解题思路，参见代码如下：

class Solution {

public:

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

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

vector<vector<int>> dp = costs;

int min1 = -1, min2 = -1;

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

int last1 = min1, last2 = min2;

min1 = -1; min2 = -1;

for (int j = 0; j < dp[i].size(); ++j) {

if (j != last1) {

dp[i][j] += last1 < 0 ? 0 : dp[i - 1][last1];

} else {

dp[i][j] += last2 < 0 ? 0 : dp[i - 1][last2];

}

if (min1 < 0 || dp[i][j] < dp[i][min1]) {

min2 = min1; min1 = j;

} else if (min2 < 0 || dp[i][j] < dp[i][min2]) {

min2 = j;

}

}

}

return dp.back()[min1];

}

};

下面这种解法不需要建立二维 dp 数组，直接用三个变量就可以保存需要的信息即可，参见代码如下：

class Solution {

public:

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

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

int min1 = 0, min2 = 0, idx1 = -1;

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

int m1 = INT\_MAX, m2 = m1, id1 = -1;

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

int cost = costs[i][j] + (j == idx1 ? min2 : min1);

if (cost < m1) {

m2 = m1; m1 = cost; id1 = j;

} else if (cost < m2) {

m2 = cost;

}

}

min1 = m1; min2 = m2; idx1 = id1;

}

return min1;

}

};