# 题目

数组的每个索引作为一个阶梯，第i个阶梯对应着一个非负数的体力花费值cost[i](索引从0开始)。

每当你爬上一个阶梯你都要花费对应的体力花费值，然后你可以选择继续爬一个阶梯或者爬两个阶梯。

您需要找到达到楼层顶部的最低花费。在开始时，你可以选择从索引为 0 或 1 的元素作为初始阶梯。

**示例1:**

输入: cost = [10, 15, 20]

输出: 15

解释: 最低花费是从cost[1]开始，然后走两步即可到阶梯顶，一共花费15。

**示例 2:**

输入: cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]

输出: 6

解释: 最低花费方式是从cost[0]开始，逐个经过那些1，跳过cost[3]，一共花费6。

**注意：**

cost的长度将会在[2, 1000]。

每一个cost[i] 将会是一个Integer类型，范围为[0, 999]。

# 分析

## 方法一：动态规划

思路：

为了解决这个问题，我们可以使用动态规划（Dynamic Programming）的方法。我们可以定义一个数组dp，其中dp[i]表示到达第i个阶梯的最低花费。由于我们可以选择从索引为0或1的阶梯开始，我们初始化dp[0]和dp[1]为cost[0]和cost[1]，分别表示从第一个和第二个阶梯开始时的花费。

接下来，我们遍历cost数组从索引2开始，计算dp[i]的值。对于每个i，我们有两个选择：从i-1个阶梯爬上来，或者从i-2个阶梯爬上来（如果i-2不小于0）。因此，dp[i]的值应该是cost[i]加上从i-1和i-2个阶梯爬上来的最低花费中的较小值。即：

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

最后，dp数组中的最后一个元素dp[cost.size()-1]表示到达楼层顶部的最低花费，但是我们还需要考虑到可以选择从dp[cost.size()-2]结束，因为我们可以选择爬一个阶梯或者两个阶梯到达顶部。因此，最终的最低花费应该是dp[cost.size()-1]和dp[cost.size()-2]中的较小值。

说明：即最后一步可能是1步上来的，也可能是2步上来的，因此这里的dp[cost.size() -1]并不一定是最后结束的地方，还可能在dp[cost.size() -2]的地方结束，即最小值不一定是dp[cost.size() -1]。

下面是完整的代码实现：

class Solution {

public:

int minCostClimbingStairs(std::vector<int>& cost) {

int n = cost.size();

if (n <= 1) return 0; // 如果cost数组为空或只有一个元素，则没有花费

std::vector<int> dp(n);

dp[0] = cost[0];

dp[1] = cost[1];

for (int i = 2; i < n; ++i) {

dp[i] = cost[i] + std::min(dp[i-1], dp[i-2]);

}

return std::min(dp[n-1], dp[n-2]); // 返回最后两个阶梯中的较小花费

}

};

这个算法的时间复杂度是O(n)，其中n是cost数组的长度，因为我们只需要遍历一次cost数组。空间复杂度也是O(n)，因为我们需要一个额外的数组dp来存储到达每个阶梯的最低花费。然而，由于我们只需要访问dp数组中的最后两个元素，我们可以使用两个变量来代替整个数组，从而将空间复杂度优化到O(1)。下面是优化后的代码：

class Solution {

public:

int minCostClimbingStairs(std::vector<int>& cost) {

int n = cost.size();

if (n <= 1) return 0; // 如果cost数组为空或只有一个元素，则没有花费

int prev2 = cost[0];

int prev1 = cost[1];

int curr;

for (int i = 2; i < n; ++i) {

curr = cost[i] + std::min(prev1, prev2);

prev2 = prev1;

prev1 = curr;

}

return std::min(prev1, prev2); // 返回最后两个阶梯中的较小花费

}

};

在这个优化后的版本中，我们只使用了三个变量prev2、prev1和curr来分别表示到达前两个阶梯和当前阶梯的最低花费，从而减少了空间使用。

在上述代码中，其实dp[0]是无意义的，因此可以根据下面的方法修改：

class Solution {

public:

int minCostClimbingStairs(std::vector<int>& cost) {

int n = cost.size();

if (n == 0) return 0; // 如果cost数组为空，则没有花费

// dp[i]表示到达第i个台阶（从下标为1开始计数）的最低花费

std::vector<int> dp(n + 1, INT\_MAX); // 初始化为一个大值，表示初始时无法到达（dp[0]无意义，这里多申请一个内存）

// 初始化dp[1]和dp[2]为到达第一个和第二个台阶的最低花费

dp[1] = cost[0];

if (n > 1) {

dp[2] = std::min(cost[0] + cost[1], cost[1]);

}

// 从第三个台阶开始计算最低花费

for (int i = 3; i <= n; ++i) {

// 到达第i个台阶的最低花费是从第i-1个或第i-2个台阶爬上来所需的花费中的较小值

dp[i] = std::min(dp[i-1], dp[i-2]) + cost[i-1]; // 注意这里用i-1作为cost的索引

}

// 返回最后一个台阶的最低花费（因为可以从倒数第二个或倒数第一个台阶爬上来）

return std::min(dp[n], dp[n-1]);

}

};

**思路：**

计算花费f[i]有一个清楚的递归关系：f[i] = cost[i] + min(f[i+1], f[i+2])。我们可以使用动态规划来实现。

**算法：**

当我们要计算f[i]时，要先计算出f[i+1]和f[i+2]。所以我们应该从后往前计算f。

在第i步，让f1，f2为f[i+1]，f[i+2]的旧值，并将其更新为f[i]，f[i+1]的新值。当我们从后遍历i时，我们会保持这些更新。在最后答案是min(f1, f2)。

**代码：**

class Solution {

public int minCostClimbingStairs(vector<int>& cost) {

int f1 = 0, f2 = 0;

for (int i = cost.size() - 1; i >= 0; --i) {

int f0 = cost[i] + min(f1, f2);

f2 = f1;

f1 = f0;

}

return min(f1, f2);

}

}

复杂度分析：

时间复杂度：O(N)。N指的是cost的长度

空间复杂度：O(1)，只使用了f1, f2。

另一种写法：

class Solution {

public:

int minCostClimbingStairs(vector<int>& cost) {

int n = cost.size();

vector<int> dp(n + 1);

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

for (int i = 2; i <= n; i++) {

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

}

return dp[n];

}

};

**复杂度分析：**

时间复杂度和空间复杂度都是O(n)