

法律声明

- 本课件包括：演示文稿，示例，代码，题库，视频和声音等，小象学院拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意，我们将保留一切通过法律手段追究违反者的权利。



关注 小象学院

第八课 动态规划

林沐

内容概述

1.8道经典动态规划的相关题目

例1:爬楼梯(easy)

例2:打家劫舍(easy)

例3:最大字段和(easy)

例4:找零钱(medium)

例5:三角形(medium)

例6:最长上升子序列(medium,hard)

例7:最小路径和(medium)

例8:地牢游戏(hard)

2.详细讲解题目解题方法、代码实现

动态规划(Dynamic Programming)概述

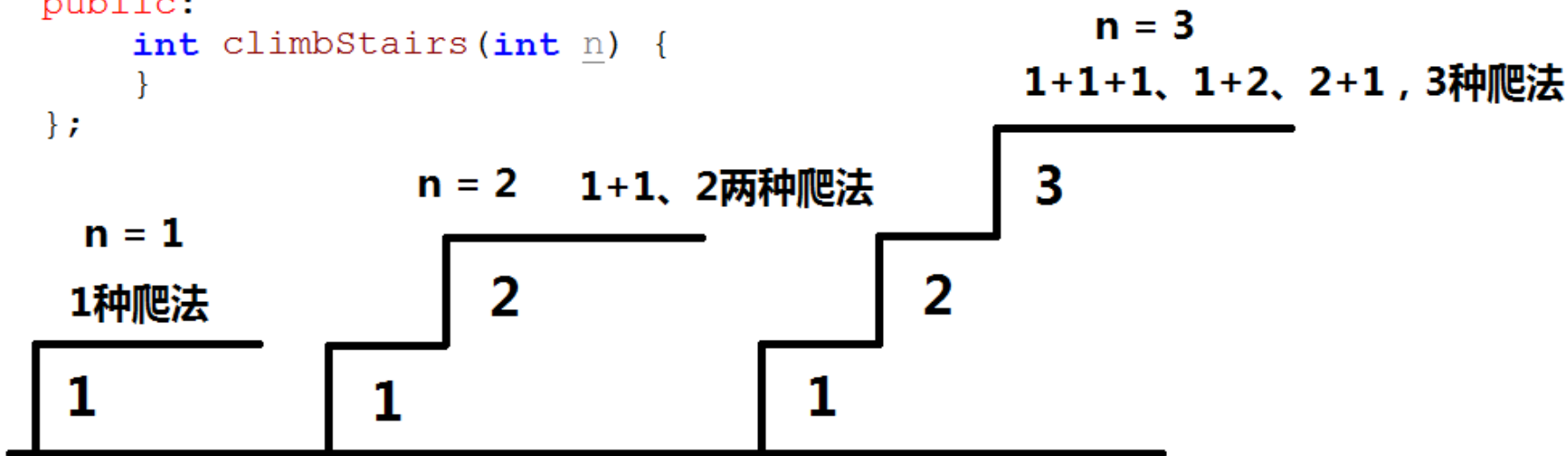
动态规划(dynamic programming)是**运筹学**的一个分支，是求解**决策过程最优化的**数学方法。它是20世纪50年代初美国数学家R.E.Bellman等人提出的**最优化原理**，它利用各阶段之间的关系，逐个求解，最终求得**全局最优解**。在设计动态规划算法时，需要确认原问题与子问题、动态规划状态、边界状态结值、状态转移方程等**关键要素**。

在算法面试中，动态规划是**最常考察**的题型之一，大多数面试官都以**否可较好的解决动态规划**相关问题来区分候选是否“聪明”。

例1: 爬楼梯

在爬楼梯时，每次可向上走**1阶台阶**或**2阶台阶**，问有n阶楼梯有**多少种**上楼的方式？

```
class Solution {  
public:  
    int climbStairs(int n) {  
    }  
};
```



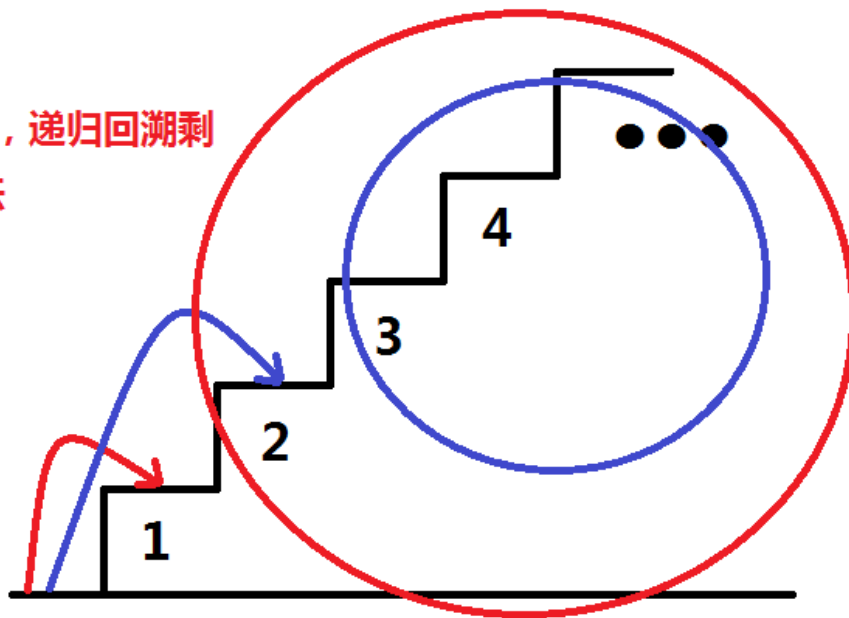
选自 **LeetCode 70. Climbing Stairs**

<https://leetcode.com/problems/climbing-stairs/description/>

难度:**Easy**

例1:暴力搜索，回溯法

当前可走1步，递归回溯剩下的台阶走法



当前走2步，递归回溯剩下的台阶走法

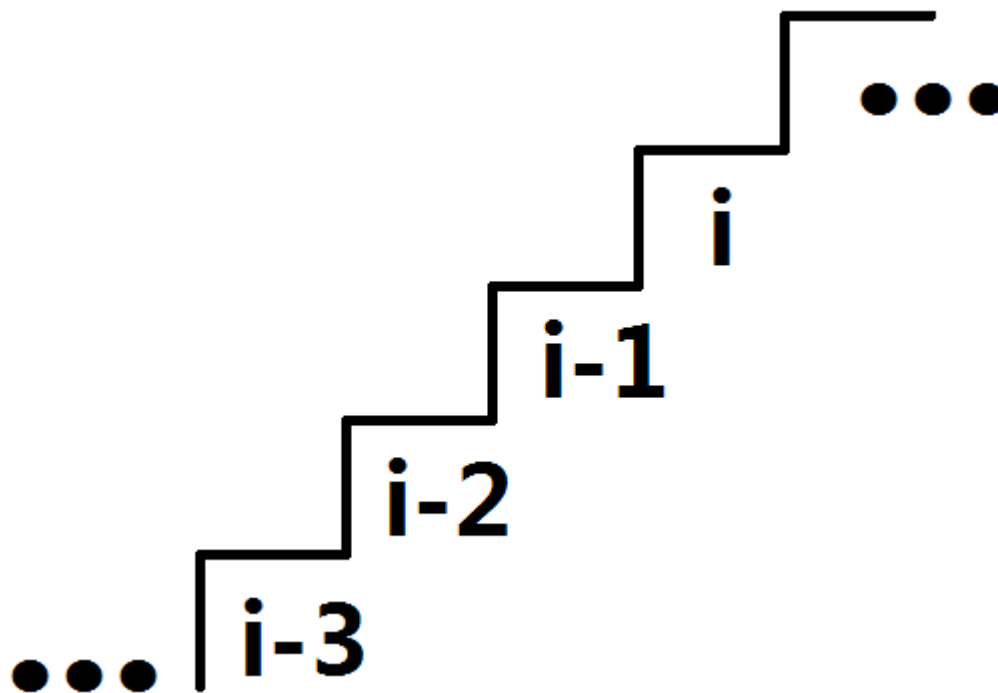
```
class Solution {
public:
    int climbStairs(int n) {
        if (n == 1 || n == 2) {
            return n;
        }
        return climbStairs(n-1) + climbStairs(n-2);
    }
};
```

Submission Result: Time Limit Exceeded ?

Last executed input: 45

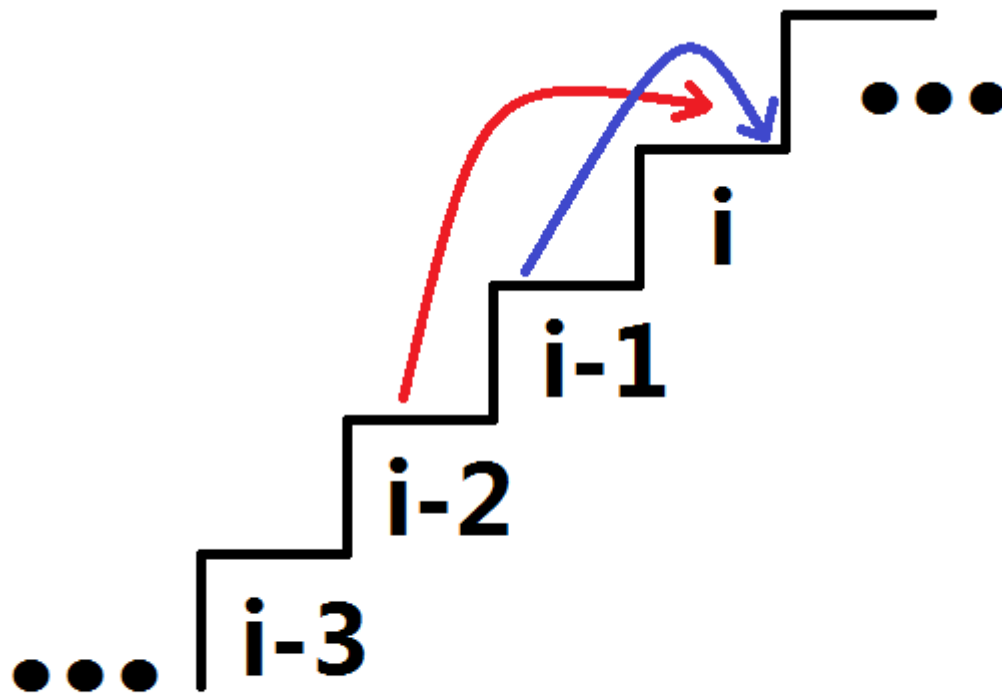
例1:思考

到达楼梯的**第 i 阶**有多少种爬法，与第几阶的爬法**直接相关**，
如何**递推**的求出第 i 阶爬法数量？



例1:分析

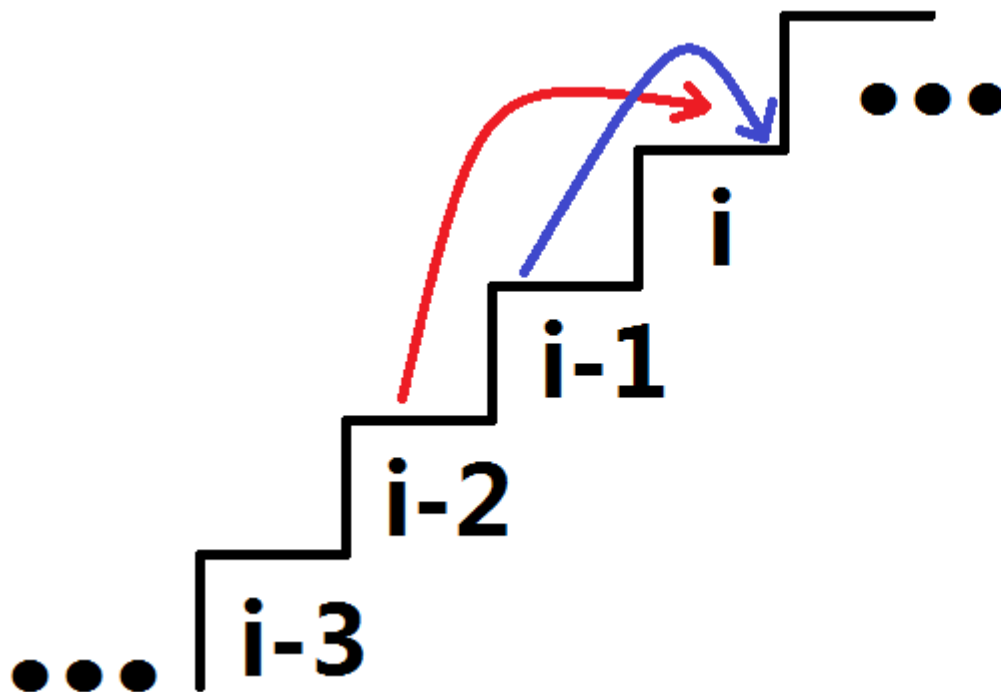
由于每次**最多**爬2阶，楼梯的**第i阶**，只可能从楼梯第i-1阶与第i-2阶**到达**。故到达**第i阶**有多少种爬法，只与**第i-1阶**、**第i-2阶**的爬法数量**直接相关**。



第i阶的爬法数量 = 第i-1阶的爬法数量 + 第i-2阶的爬法数量

例1:算法思路

1. 设置**递推数组** $dp[0...n]$, $dp[i]$ 代表到达第*i*阶, 有多少种走法, **初始化**数组元素为0。
2. 设置到达**第1阶**台阶, **有1种**走法; 到达**第2阶**台阶, **有2种**走法。
3. 利用**循环递推**从第3阶至第*n*阶结果:
到达**第*i*阶**的方式**数量** = 到达**第*i*-1阶**的方式**数量** + 到达**第*i*-2阶**的方式**数量**



$$dp[0] = 0$$

$$dp[1] = 1$$

$$dp[2] = 2$$

$$dp[3] = dp[1] + dp[2] = 3$$

...

$$dp[i] = dp[i-1] + dp[i-2]$$

...

$$dp[n] = dp[n-1] + dp[n-2]$$

例1:课堂练习

```
#include <vector>
class Solution {
public:
    int climbStairs(int n) {
        std::vector<int> dp(n + 3, 0);
        dp[1] = 1;
        

1


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

2


        }
        

3


    }
};
```

3分钟时间填写代码，
有问题随时提出！

例1:实现

```
#include <vector>
class Solution {
public:
    int climbStairs(int n) {
        std::vector<int> dp(n + 3, 0);
        dp[1] = 1;
        dp[2] = 2;
        for (int i = 3; i <= n; i++) {
            dp[i] = dp[i-1] + dp[i-2];
        }
        return dp[n];
    }
};
```

例1:测试与leetcode提交结果

```
int main() {  
    Solution solve;  
    printf("%d\n", solve.climbStairs(3));  
    return 0;  
}
```

Climbing Stairs


Submission Details

45 / 45 test cases passed.

Status: **Accepted**

Runtime: 3 ms

Submitted: 0 minutes ago



```
3  
请按任意键继续. . .
```

例1:动态规划原理

1.确认原问题与子问题:

原问题为求n阶台阶所有走法的数量，子问题是求1阶台阶、2阶台阶、...、n-1阶台阶的走法。

2.确认状态:

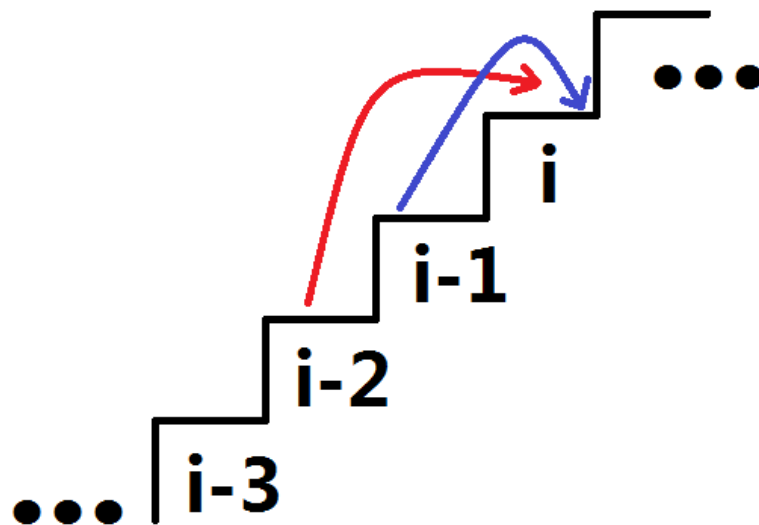
本题的动态规划**状态单一**，第i个状态即为i阶台阶的所有走法数量。

3.确认边界状态的值:

边界状态为1阶台阶与2阶台阶的走法，1阶台阶有1种走法，2阶台阶有2种走法，即 $dp[1] = 1$ ， $dp[2] = 2$ 。

4.确定状态转移方程:

将求第i个状态的值转移为求第i-1个状态值与第i-2个状态的值，动态规划转移方程， $dp[i] = dp[i-1] + dp[i-2]$; ($i \geq 3$)



第i阶的爬法数量 = 第i-1阶的爬法数量 + 第i-2阶的爬法数量

例2: 打家劫舍

在**一条直线**上，有n个房屋，每个房屋中有**数量不等**的财宝，有一个盗贼希望从房屋中盗取财宝，由于房屋中有**报警器**，如果同时从**相邻的**两个房屋中盗取财宝就会**触发**报警器。问在**不触发报警器**的前提下，**最多**可获取多少财宝？

0 1 2 3 4 5
[5, 2, 6, 3, 1, 7] 5+6+7 = 18

```
class Solution {  
public:  
    int rob(std::vector<int>& nums) {  
    }  
};
```

选自 **LeetCode 198. House Robber**

<https://leetcode.com/problems/house-robber/description/>

难度: **Easy**

例2:思考

1. n 个房屋，每个房间都有**盗取/不盗取**两种可能，类似求子集(暴力搜索)的方法，在**不触发警报**的情况下，选择**总和最大**的子集，最多有 2^n 种可能，时间复杂度 $O(2^n)$ ，是否有**更好的**方法？

2. **贪心算法**是否可行？

例如，在满足不触发警报的同时，每次选择**财宝最多**的房间。
如， $[5, 2, 6, 3, 1, 7]$ ，选择**最大的**7, 6, 5恰好是**最佳答案**。

3. 若考虑**动态规划**(dp)方法，如何确认dp原问题与子问题、状态、边界状态、状态转移方程？

例2:分析

由于**同时**从相邻的**两个房屋**中盗取财宝就会触发**报警器**，故：

- a. 若**选择**第*i*个房间盗取财宝，就一定**不能选择**第*i-1*个房间盗取财宝；
- b. 若**不选择**第*i*个房间盗取财宝，则相当于只考虑**前*i-1*个房间**盗取财宝。

只考虑第1个房间5:

[**5**, 2, 6, 3, 1, 7], 最佳解5

考虑第1,2个房间5, 2:

[**5**, **2**, 6, 3, 1, 7], 最佳解5

考虑第1,2,3个房间5, 2, 6:

[**5**, **2**, **6**, 3, 1, 7], 最佳解11

考虑第1,2,3,4个房间5, 2, 6, 3:

[**5**, **2**, **6**, **3**, 1, 7], 最佳解11

考虑第1,2,3,4,5个房间5, 2, 6, 3, 1:

[**5**, **2**, **6**, **3**, **1**, 7], 最佳解12

考虑全部房间:

[**5**, **2**, **6**, **3**, **1**, **7**], 最佳解18

例2:算法思路

[5, 2, 6, 3, 1, 7]

1.确认原问题与子问题:

原问题为求n个房间的最优解, **子问题**为求前1个房间、前2个房间、...、前n-1个房间的最优解。

2.确认状态:

第i个状态即为**前i个房间**能够获得的最大财宝(最优解)。

3.确认边界状态的值:

前1个房间的最优解, 第1个房间的财宝;

前2个房间的最优解, 第1、2个房间中较大财宝的。

4.确定状态转移方程:

a. **选择**第i个房间:第i个房间+前i-2个房间的最优解

b. **不选择**第i个房间:前i-1个房间的最优解

动态规划转移方程:

$dp[i] = \max(dp[i-1], dp[i-2] + nums[i]); (i \geq 3)$

设第i个房间的最优解为dp[i]

$dp[1] = 5$

$dp[2] = 5$

$dp[3] = \max(dp[1] + nums[3], dp[2])$
 $= \max(5 + 6, 5) = 11$

$dp[4] = \max(dp[2] + nums[4], dp[3])$
 $= \max(5 + 3, 11) = 11$

$dp[5] = \max(dp[3] + nums[5], dp[4])$
 $= \max(11 + 1, 11) = 12$

$dp[6] = \max(dp[4] + nums[6], dp[5])$
 $= \max(11 + 7, 12) = 18$

例2:课堂练习

```
class Solution {
public:
    int rob(std::vector<int>& nums) {
        if (nums.size() == 0) {
            return 0;
        }
        if (nums.size() == 1) {
            return nums[0];
        }
        std::vector<int> dp(nums.size(), 0);
        dp[0] = nums[0];
        dp[1] = std::max(nums[0], nums[1]);
        for (int i = 2; i < nums.size(); i++) {
            dp[i] = std::max(1, 2);
        }
        return 3;
    }
};
```

//设第i个房间的最优解为dp[i]

3分钟时间填写代码，
有问题随时提出！

例2:实现

```
class Solution {
public:
    int rob(std::vector<int>& nums) {
        if (nums.size() == 0) {
            return 0;
        }
        if (nums.size() == 1) {
            return nums[0];
        }
        std::vector<int> dp(nums.size(), 0);
        dp[0] = nums[0];
        dp[1] = std::max(nums[0], nums[1]);
        for (int i = 2; i < nums.size(); i++) {
            dp[i] = std::max(dp[i-1], dp[i-2] + nums[i]);
        }
        return dp[nums.size() - 1];
    }
};
```

//设第i个房间的最优解为dp[i]

例2:测试与leetcode提交结果

```
int main() {  
    Solution solve;  
    std::vector<int> nums;  
    nums.push_back(5);  
    nums.push_back(2);  
    nums.push_back(6);  
    nums.push_back(3);  
    nums.push_back(1);  
    nums.push_back(7);  
    printf("%d\n", solve.rob(nums));  
    return 0;  
}
```

18

请按任意键继续. . .

House Robber

Submission Details

69 / 69 test cases passed.

Status: **Accepted**

Runtime: 0 ms

Submitted: 0 minutes ago

例3:最大子段和

给定一个数组，求这个数组的**连续子数组**中，**最大的**那一段的和。

如数组[-2,1,-3,4,-1,2,1,-5,4]:

连续子数组如:

[-2,1]、 [1,-3,4,-1]、 **[4,-1,2,1]**、 ...、 [-2,1,-3,4,-1,2,1,-5,4]， 和最大的是[4,-1,2,1]， 为6。

```
class Solution {  
public:  
    int maxSubArray(std::vector<int>& nums) {  
    }  
};
```

选自 **LeetCode 53. Maximum Subarray**

<https://leetcode.com/problems/maximum-subarray/description/>

难度:**Easy**

例3:思考

数组 $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$ ，所有连续子段：

$[-2]$	$[1]$	$[1]$	$[-5]$	$[4]$
$[-2, 1]$	$[1, -3]$	$[1, -5]$	$[-5, 4]$	
$[-2, 1, -3]$	$[1, -3, 4]$...	$[1, -5, 4]$	
...	...			
$[-2, 1, -3, 4, -1, 2, 1, -5, 4]$	$[1, -3, 4, -1, 2, 1, -5, 4]$			

暴力枚举所有连续子段的和，**复杂度**是？

若尝试**动态规划**方法，最关键的是确认**动态规划状态**，若假设第 i 个状态($dp[i]$)代表**前 i 个数字**组成的连续的最大子段和，能否**推导出** $dp[i]$ 与 $dp[i-1]$ 之间的关系呢？

例3:分析

实际上, 如果设第 i 个状态($dp[i]$)代表**前 i 个数字**组成的**连续的最大子段和**, 并不能根据 $dp[i-1]$ 、 $dp[i-2]$ 、...、 $dp[0]$ 推导出 $dp[i]$ 。

例如:

$[-2, 1, 1, -3, 4]$ $dp[0] = -2; \quad dp[2] = 2;$
 $dp[1] = 1; \quad dp[3] = 2$
 $dp[4] = ??$

前4个数字连续的最大子段: $[-2, 1, 1, -3]$ 4]

前5个数字连续的最大子段: $[-2, 1, 1, -3, 4]$

两者**不相邻**, 故无法构成**连续的**子数组, 之间无内在联系, 故**无法**进行推导。

为了让**第 i 个状态**的最优解与**第 $i-1$ 个状态**的最优解产生**直接联系**, 思考:

如果第 i 个状态($dp[i]$)代表**以第 i 个数字结尾**的最大子段和, 那么 $dp[i]$ 与 $dp[i-1]$ 之间的**关系**是什么? 如何根据 $dp[i-1]$ **推导**出 $dp[i]$? 这样推导又如何求得最终结果?

例3:算法思路

将求**n个数**的数组的最大子段和，转换为**分别求出**以第1个、第2个、...、**第i个**、...、第n个**数字结尾**的最大子段和，再找出这**n个结果中最大的**，即为结果。

动态规划算法:

第i个状态($dp[i]$)即为**以第i个数字结尾**的最大子段和(最优解)。由于以第i-1个数字结尾的最大子段和($dp[i-1]$)与 $nums[i]$ **相邻**:

$dp[0]$ [-2]

$dp[1]$ [-2, 1]

$dp[2]$ [-2, 1, -3]

 [-2, 1, -3, 4]

... [-2, 1, -3, 4, -1]

 [-2, 1, -3, 4, -1, 2]

$dp[i]$ [-2, 1, -3, 4, -1, 2, 1]

 [-2, 1, -3, 4, -1, 2, 1, -5]

... [-2, 1, -3, 4, -1, 2, 1, -5, 4]

若 $dp[i-1] > 0$:

$dp[i] = dp[i-1] + nums[i]$

否则:

$dp[i] = nums[i]$

边界值:以第1个数字结尾的最大子段和 $dp[0] = nums[0]$ 。

$dp[0] = nums[0] = -2$

$dp[1] = \max(dp[0] + nums[1], nums[1]) = \max(-2, 1) = 1$

$dp[2] = \max(dp[1] + nums[2], nums[2]) = \max(1 - 3, -3) = -2$

...

$dp[i] = \max(dp[i-1] + nums[i], nums[i])$

...

例3:课堂练习

```
class Solution {
public:
    int maxSubArray(std::vector<int>& nums) {
        std::vector<int> dp(nums.size(), 0);
        dp[0] = 1;
        int max_res = dp[0];
        for (int i = 1; i < nums.size(); i++) {
            dp[i] = std::max(2, nums[i]);
            if (3) {
                max_res = dp[i];
            }
        }
        return max_res;
    }
};
```

3分钟时间填写代码，
有问题随时提出！

例3:实现

```
class Solution {
public:
    int maxSubArray(std::vector<int>& nums) {
        std::vector<int> dp(nums.size(), 0);
        dp[0] = nums[0];

        int max_res = dp[0];
        for (int i = 1; i < nums.size(); i++) {
            dp[i] = std::max(dp[i-1] + nums[i], nums[i]);

            if (max_res < dp[i]) {
                max_res = dp[i];
            }
        }
        return max_res;
    }
};
```

例3:测试与leetcode提交结果

```
int main() {  
    Solution solve;  
    std::vector<int> nums;  
    nums.push_back(-2);  
    nums.push_back(1);  
    nums.push_back(-3);  
    nums.push_back(4);  
    nums.push_back(-1);  
    nums.push_back(2);  
    nums.push_back(1);  
    nums.push_back(-5);  
    nums.push_back(4);  
    printf("%d\n", solve.maxSubArray(nums));  
    return 0;  
}
```

Maximum Subarray

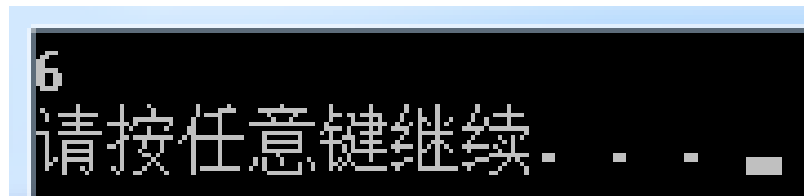
Submission Details

202 / 202 test cases passed.

Status: **Accepted**

Runtime: 9 ms

Submitted: 0 minutes ago



例4:找零钱

已知**不同面值**的钞票，求如何用**最少数量**的钞票组成**某个金额**，求可以使用的最少钞票数量。如果**任意数量**的已知面值钞票都**无法**组成该金额，则返回**-1**。

例如:

钞票面值: [1, 2, 5] ; 金额: $11 = 5 + 5 + 1$; 需要3张。

钞票面值: [2] ; 金额: 3 ; 无法组成, 返回-1。

钞票面值: [1, 2, 5, 7, 10] ; 金额: $14 = 7 + 7$; 需要2张。

```
class Solution {  
public:  
    int coinChange(std::vector<int>& coins, int amount) {  
    }  
};
```

选自 **LeetCode 322. Coin Change**

<https://leetcode.com/problems/coin-change/description/>

难度:**Medium**

例4:思考:贪心可否?

钞票面值: [1, 2, 5, 10] ; 金额: 14; **最优解**需要3张

贪心思想:每次**优先使用大面值**的金额, 如:

先选1张10块的, 剩下4元; 再选1张2元的, 剩下2元; 再选1张2元的, **搞定!**

钞票面值: [1, 2, 5, 7, 10] ; 金额: 14; **最优解**需要2张(两张7块的)。

仍然用贪心思想:

先选1张10块的, 剩下4元; 再选1张2元的, 剩下2元; 再选1张2元的, 这就**错了!**

结论:

贪心思想在**个别面值组合**时是可以的, 比如日常生活中的RMB面值[1, 2, 5, 10, 20, 50, 100], 但是本题面值不确定, 故贪心思想不可以。

如果使用**动态规划**求解该问题, 又需要如何设计**解决方案**呢?

例4:算法思路

钞票面值: $\text{coins} = [1, 2, 5, 7, 10]$; **金额**: 14

$\text{dp}[i]$, 代表金额 i 的**最优解**(即最小使用张数)

数组 $\text{dp}[]$ 中存储**金额1至金额14**的最优解(最少使用钞票的数量)。

在计算 $\text{dp}[i]$ 时, $\text{dp}[0]$ 、 $\text{dp}[1]$ 、 $\text{dp}[2]$ 、...、 $\text{dp}[i-1]$ 都是**已知**的:

而**金额 i** 可由:

金额 $i-1$ 与 $\text{coins}[0](1)$ 组合;

金额 $i-2$ 与 $\text{coins}[1](2)$ 组合;

金额 $i-5$ 与 $\text{coins}[2](5)$ 组合;

金额 $i-7$ 与 $\text{coins}[3](7)$ 组合;

金额 $i-10$ 与 $\text{coins}[4](10)$ 组合;

即**状态 i** 可由状态 $i-1$ 、 $i-2$ 、 $i-5$ 、 $i-7$ 、 $i-10$, 5个**状态所转移到**, 故,

$$\text{dp}[i] = \min(\text{dp}[i-1], \text{dp}[i-2], \text{dp}[i-5], \text{dp}[i-7], \text{dp}[i-10]) + 1$$

例4:算法思路

初始化:

$dp[1] = 1$	$dp[8] = -1$
$dp[2] = 1$	$dp[9] = -1$
$dp[3] = -1$	$dp[10] = 1$
$dp[4] = -1$	$dp[11] = -1$
$dp[5] = 1$	$dp[12] = -1$
$dp[6] = -1$	$dp[13] = -1$
$dp[7] = 1$	$dp[14] = -1$

$coins = \{1, 2, 5, 7, 10\}$

金额1最优解: $dp[1] = 1$

金额2最优解: $dp[2] = 1$

金额3最优解计算:

1) $3 = 1(coins[0]) + 2$

$dp[3] = 1 + dp[2]$

2) $3 = 2(coins[1]) + 1$

$dp[3] = 1 + dp[1]$

$dp[3] = \text{getmin}(dp[2], dp[1])$

金额4的计算:

1) $4 = 1(coins[0]) + 3$

$dp[4] = 1 + dp[3]$

2) $4 = 2(coins[1]) + 2$

$dp[4] = 1 + dp[2]$

$dp[4] = \text{getmin}(dp[3], dp[2]) + 1$

金额5最优解: $dp[5] = 1$

金额6的计算:

1) $6 = 1(coins[0]) + 5$

$dp[6] = 1 + dp[5]$

2) $6 = 2(coins[1]) + 4$

$dp[6] = 1 + dp[4]$

3) $6 = 5(coins[2]) + 1$

$dp[6] = 1 + dp[1]$

$dp[6] = \text{getmin}(dp[5], dp[4], dp[1]) + 1$

例4:算法思路

递推至dp[14]

● ● ●

coins={1, 2, 5, 7, 10}

- | | |
|-------------------------------------|------------------------------------|
| 1) $14 = 1(\text{coins}[0]) + 13$ | 4) $14 = 7(\text{coins}[3]) + 7$ |
| $\text{dp}[14] = 1 + \text{dp}[13]$ | $\text{dp}[14] = 1 + \text{dp}[7]$ |
| 2) $14 = 2(\text{coins}[1]) + 12$ | 5) $14 = 10(\text{coins}[4]) + 4$ |
| $\text{dp}[14] = 1 + \text{dp}[12]$ | $\text{dp}[14] = 1 + \text{dp}[4]$ |
| 3) $14 = 5(\text{coins}[2]) + 9$ | |
| $\text{dp}[14] = 1 + \text{dp}[9]$ | |
- $\text{dp}[14] = \text{getmin}(\text{dp}[13], \text{dp}[12], \text{dp}[9], \text{dp}[7], \text{dp}[4]) + 1$

另外: 设 $\text{dp}[0] = 0$

$\text{dp}[1] = 1 + \text{dp}[0]$

$\text{dp}[2] = 1 + \text{dp}[0]$

$\text{dp}[5] = 1 + \text{dp}[0]$

$\text{dp}[7] = 1 + \text{dp}[0]$

$\text{dp}[10] = 1 + \text{dp}[0]$

coins={1, 2, 5, 7, 10}

设i代表金额, coins[j]代表第j个面值的金额:

当 $i - \text{coins}[j] \geq 0$ 且 $\text{dp}[i - \text{coins}[j]] \neq -1$ 时:

$j = 0, 1, 2, 3, 4$; $\text{coins}[j] = 1, 2, 5, 7, 10$

$\text{dp}[i] = \text{getmin}(\text{dp}[i - \text{coins}[j]]) + 1$

例4:课堂练习

```
class Solution {
public:
    int coinChange(std::vector<int>& coins, int amount) {

        std::vector<int> dp; //初始化dp数组
        for (int i = 0; i <= amount; i++) {
            1
        }
        2 //递推
        for (int i = 1; i <= amount; i++) { //循环各个面值，找到dp[i]最优解
            for (int j = 0; j < 3; j++) {
                if (4) {
                    if (dp[i] == -1 || dp[i] > dp[i - coins[j]] + 1) {
                        5
                    }
                }
            }
        }
        return dp[amount];
    }
};
```

5分钟时间填写代码，
有问题随时提出！

例4:实现

```
class Solution {
public:
    int coinChange(std::vector<int>& coins, int amount) {

        std::vector<int> dp;           //初始化dp数组
        for (int i = 0; i <= amount; i++){
            dp.push_back(-1);          //最初所有金额的最优解均为-1(不可达到)
        }
        dp[0] = 0;                     //金额0最优解0           //递推
        for (int i = 1; i <= amount; i++){           //循环各个面值，找到dp[i]最优解
            for (int j = 0; j < coins.size(); j++){ //递推条件
                if (i - coins[j] >= 0 && dp[i - coins[j]] != -1) {
                    if (dp[i] == -1 || dp[i] > dp[i - coins[j]] + 1) {
                        dp[i] = dp[i - coins[j]] + 1;
                    }
                }
            }
        }
        return dp[amount];
    }
};
```

例4:测试与leetcode提交结果

```
int main() {
    Solution solve;
    std::vector<int> coins;
    coins.push_back(1);
    coins.push_back(2);
    coins.push_back(5);
    coins.push_back(7);
    coins.push_back(10);
    for (int i = 1; i <= 14; i++) {
        printf("dp[%d] = %d\n", i, solve.coinChange(coins, i));
    }
    return 0;
}
```

Coin Change

Submission Details

182 / 182 test cases passed.

Runtime: 29 ms

```
dp[1] = 1
dp[2] = 1
dp[3] = 2
dp[4] = 2
dp[5] = 1
dp[6] = 2
dp[7] = 1
dp[8] = 2
dp[9] = 2
dp[10] = 1
dp[11] = 2
dp[12] = 2
dp[13] = 3
dp[14] = 2
请按任意键继续. . .
```

Status: Accepted

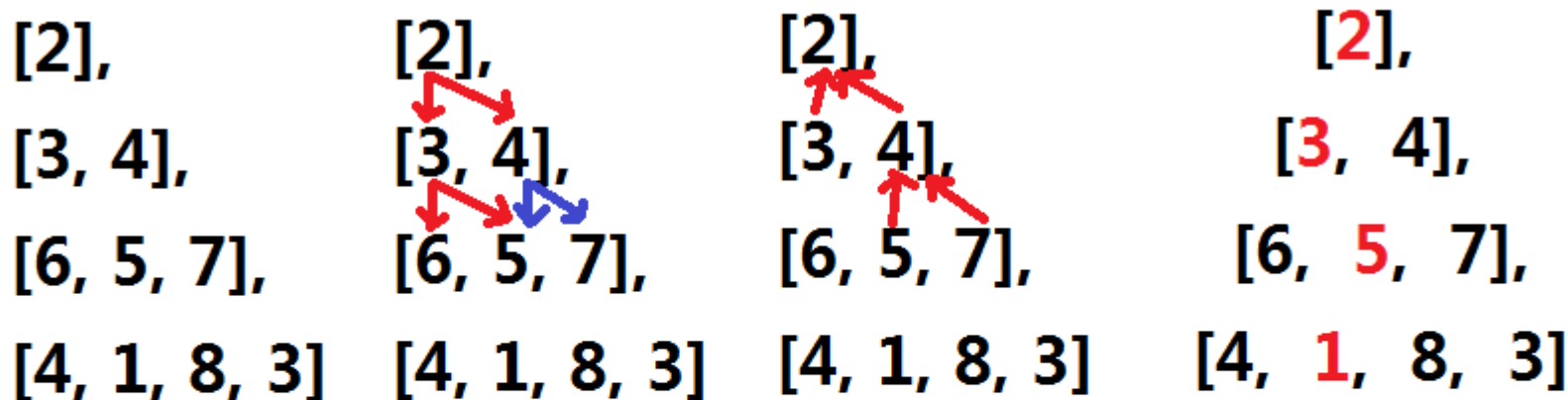
Submitted: 0 minutes ago

课间休息10分钟

有问题提出！

例5: 三角形

给定一个二维数组，其保存了一个数字三角形，求从数字三角形**顶端到底端**各数字**和最小**的路径之和，每次可以向下走**相邻**的两个位置。



```
class Solution {  
public:  
    int minimumTotal(std::vector<std::vector<int>> & triangle) {  
    }  
};
```

选自 **LeetCode 120. Triangle**

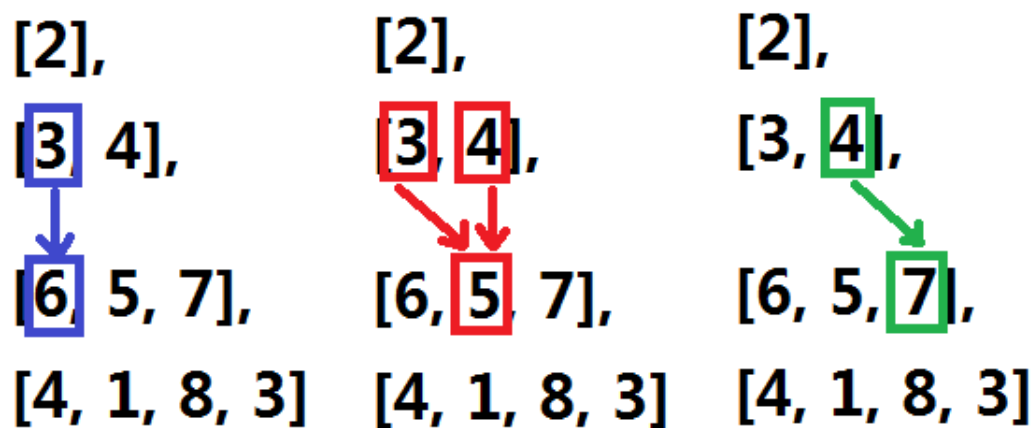
<https://leetcode.com/problems/maximum-subarray/description/>

难度:**Medium**

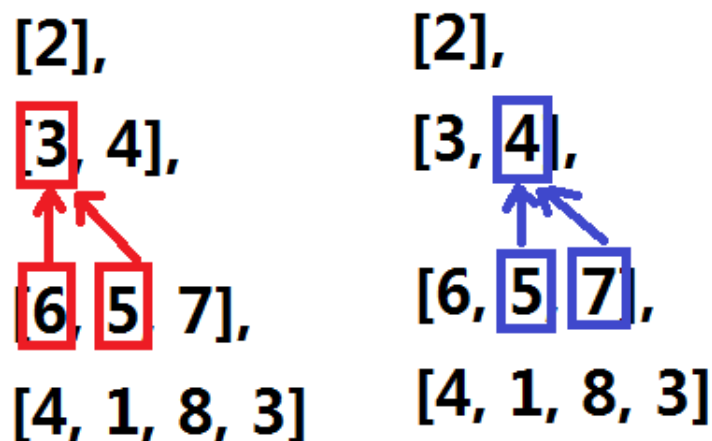
例5:思考

1. **从上到下**或者**从下到上**的寻找路径的**思考方式**本质是一样的吗?
2. 假设 $dp[i][j]$ 代表了数组三角形第 i 行、第 j 列的最优解, 从上到下与从下到上哪种方式**递推更容易**? (更少的考虑**边界条件**)

从上到下的思考:



从下到上的思考:



例5:分析1

从上到下的推导:

假设三角形只有1层:

[2]

路径各元素和的最小值:

[2]

若三角形有2层:

[2] [2]
↓ ↓
[3, 4] [3, 4],

到达第二层各个位置的最优值:

[2] [2]
[3] 4] [5] 6]

从2->3 = 5、2->4 = 6中, 选择较小的5

若三角形有3层:

[2], [2], [2],
[3] 4], [3] 4], [3] 4],
↓ ↓ ↓
[6] 5, 7], [6] 5, 7], [6] 5, 7],

到达第三层各个位置的最优值:

[2] [2] [2], [2],
[5, 6] [5, 6] [3] 4], [3] 4],
[?, ?, ?] [11, ?, 13] [6] 5, 7], [6] 5, 7],

故?的取值为10或11, 由于希望到达各个位置最小, 故选择红色路径

最终最优值结果数组: [2] 实际上?代表了
[5, 6] $\min(5, 6) + 5 = 10$
[11, 10, 13]

若三角形有4层: 最优值三角形: 最后一层:
[2] [4, 1, 8, 3]
[5, 6]
[11, 10, 13]

[11] 10, 13] [11, 10] 13] [11, 10, 13] [11, 10, 13]
[4, 1, 8, 3] [4, 1, 8, 3] [4, 1, 8, 3] [4, 1, 8, 3]
↓ ↓ ↓ ↓
[15, ?, ?, ?] [15, 11, ?, ?] [15, 11, 18, ?] [15, 11, 18, 16]
最终结果为11。

例5:分析2

设一个与数字三角形对应的最优值三角形:

[2],	[?],
[3, 4],	[?, ?],
[6, 5, 7],	[?, ?, ?],
[4, 1, 8, 3]	[?, ?, ?, ?]

从下到上的推导: 最优值三角形即为:

假设三角形只有1层:	[?],
[4, 1, 8, 3]	[?, ?],
	[?, ?, ?],
	[4, 1, 8, 3]

假设三角形有2层: [6, 5, 7],

推导: [4, 1, 8, 3]

[6, 5, 7],	[6, 5, 7],	最优值三角形:
[4, 1, 8, 3]	[4, 1, 8, 3]	[?],
		[?, ?],
[6, 5, 7],		[7, 6, 10],
[4, 1, 8, 3]		[4, 1, 8, 3]

假设三角形有3层:

[3, 4],
[6, 5, 7],
[4, 1, 8, 3]

推导:

[3, 4],	[3, 4],
[7, 6, 10],	[7, 6, 10],

假设三角形有4层:

[2],
[3, 4],
[6, 5, 7],
[4, 1, 8, 3]

推导:

[2]
[9, 10],

最优值三角形:

[?],
[9, 10],
[7, 6, 10],
[4, 1, 8, 3]

最优值三角形:

[11],
[9, 10],
[7, 6, 10],
[4, 1, 8, 3]

从上到下的思考:

[2],	[2],
[3, 4],	[3, 4],
[6, 5, 7],	[6, 5, 7],
[4, 1, 8, 3]	[4, 1, 8, 3]

从下到上的思考:

[2],	[2],
[3, 4],	[3, 4],
[6, 5, 7],	[6, 5, 7],
[4, 1, 8, 3]	[4, 1, 8, 3]

例5:算法思路

1. 设置一个二维数组，**最优值三角形** $dp[i][j]$ ，并初始化数组元素为0。 $dp[i][j]$ 代表了从底向上递推时，走到三角形第*i*行第*j*列的最优解。
2. 从三角形的**底面**向三角形**上方**进行动态规划：
 - a. 动态规划边界条件:底面上的最优值即为数字三角形的**最后一层**。
 - b. 利用*i*循环，从**倒数第二层**递推至**第一层**，对于每层的各列，进行动态规划递推：
第*i*行，第*j*列的**最优解**为 $dp[i][j]$ ，可到达(*i*,*j*)的两个位置的最优解 $dp[i+1][j]$ 、 $dp[i+1][j+1]$ ：
 $dp[i][j] = \min(dp[i+1][j], dp[i+1][j+1]) + triangle[i][j]$
3. 返回 $dp[0][0]$

$dp[i][j] = \min(dp[i+1][j], dp[i+1][j+1]) + triangle[i][j]$

[?]

[?, ?]

[?, ?, ?]

[...]

[..., $dp[i][j]$, ...]

[..., $dp[i+1][j]$, $dp[i+1][j+1]$, ...]

[...]

原三角形:

[2],

[3, 4],

[6, 5, 7],

[4, 1, 8, 3]

最优解三角形:

[0],

[0, 0],

[0, 0, 0],

[0, 0, 0, 0]

[0],

[0, 0],

[0, 0, 0],

[4, 1, 8, 3]

[0],

[0, 0],

[7, 6, 10],

[4, 1, 8, 3]

[0],

[9, 10],

[7, 6, 10],

[4, 1, 8, 3]

[11],

[9, 10],

[7, 6, 10],

[4, 1, 8, 3]

例5:课堂练习

```
class Solution {
public:
    int minimumTotal(std::vector<std::vector<int>> & triangle) {
        if (triangle.size() == 0) {
            return 0;
        }
        std::vector<std::vector<int>> dp;
        for (int i = 0; i < triangle.size(); i++) {
            dp.push_back(std::vector<int>());
            for (int j = 0; j < triangle.size(); j++) {
                dp[i].push_back(0);
            }
        }
        for (int i = 0; i < dp.size(); i++) {
            dp[dp.size()-1][i] = 1
        }
        for (int i = dp.size() - 2; i >= 0; i--) {
            for (int j = 0; j < dp[i].size(); j++)
                dp[i][j] = 2
        }
        return 3
    }
};
```

3分钟时间填写代码，
有问题随时提出！

例5:实现

```
class Solution {
public:
    int minimumTotal(std::vector<std::vector<int> >& triangle) {
        if (triangle.size() == 0) {
            return 0;
        }
        std::vector<std::vector<int> > dp;
        for (int i = 0; i < triangle.size(); i++) {
            dp.push_back(std::vector<int>());
            for (int j = 0; j < triangle.size(); j++) {
                dp[i].push_back(0);
            }
        }
        for (int i = 0; i < dp.size(); i++) {
            dp[dp.size()-1][i] = triangle[dp.size()-1][i];
        }
        for (int i = dp.size() - 2; i >= 0; i--) {
            for (int j = 0; j < dp[i].size(); j++)
                dp[i][j] = std::min(dp[i+1][j], dp[i+1][j+1]) + triangle[i][j];
        }
        return dp[0][0];
    }
};
```

例5:测试与leetcode提交结果

```
int main() {
    std::vector<std::vector<int> > triangle;
    int test[][10] = {{2}, {3, 4}, {6, 5, 7}, {4, 1, 8, 3}};
    for (int i = 0; i < 4; i++) {
        triangle.push_back(std::vector<int>());
        for (int j = 0; j < i + 1; j++) {
            triangle[i].push_back(test[i][j]);
        }
    }
    Solution solve;
    printf("%d\n", solve.minimumTotal(triangle));
    return 0;
}
```

Triangle

Submission Details

43 / 43 test cases passed.

Status: **Accepted**

Runtime: 6 ms

Submitted: 0 minutes ago

11

请按任意键继续 . . .

例6:最长上升子序列

已知一个**未排序**数组，求这个数组**最长上升子序列**的**长度**。

例如: [1, 3, 2, 3, 1, 4],

其中有很多上升子序列，如[1, 3]、[1, 2, 3]、 [1, 2, 3, 4]等，其中**最长的**上升子序列长度为4。分别考虑 $O(n^2)$ 与 $O(n \cdot \log n)$ 两种复杂度算法。

```
class Solution {  
public:  
    int lengthOfLIS(std::vector<int>& nums) {  
    }  
};
```

选自 **LeetCode 300. Longest Increasing Subsequence**

<https://leetcode.com/problems/longest-increasing-subsequence/description/>

难度:**Medium, Hard**

例6:思考

暴力枚举: [1, 3, 2, 3, 1, 4]

n个元素组成的数组，**枚举**数组的全部子序列，即数组中的任意某个元素都有**选择、不选择**两种可能，时间复杂度 $O(2^n)$ ，枚举时选择**最长的**子序列长度作为结果。

若采用**动态规划**，设第i个状态为dp[i]:

- 1.若第i个状态代表**前i个数字**中最长上升子序列的长度，是否可找出dp[i]与dp[i-1]的关系?
- 2.若第i个状态代表**以第i个数字**为结尾的最长上升子序列的长度，是否可找出dp[i]与dp[i-1]的关系?再如何求出n个数字的最长上升子序列?
- 3.思考该题与例3-最大子段和的**相似**之处。

例6:分析1

[1, 3, 2, 3, 1, 4]

若第*i*个状态 $dp[i]$ 代表前*i*个元素中最长上升子序列的长度:

$dp[i-1]$ 代表前*i-1*个元素中的最长上升子序列长度,

...

如:

$dp[0] = 1, [1]$

$dp[1] = 2, [1, 3]$

$dp[2] = 2, [1, 3], [1, 2]$

$dp[3] = 3, [1, 2, 3]$

$dp[4] = 3, [1, 2, 3]$

$dp[5] = ?$

实际 $dp[5]$ 与之前的结果无直接联系, 故无法递推。

例6:分析2

[1, 3, 2, 3, 1, 4]

若第*i*个状态 $dp[i]$ 代表以第*i*个元素结尾的最长上升子序列的长度:

$dp[i-1]$ 代表以第*i-1*个元素结尾的最长上升子序列长度,

..., $nums[i]$ 一定是 $dp[i]$ 所对应的最长上升子序列中最大的元素(因为在末尾)

如:

$dp[5]$ 对应的 $nums[5] = 4$:

$dp[0] = 1, [1]$

大于 $dp[0]$ 对应 $num[0]$, 则 $[1]+[4] = [1,4]$

$dp[1] = 2, [1,3]$

大于 $dp[1]$ 对应 $num[1]$, 则 $[1,3]+[4] = [1,3,4]$

$dp[2] = 2, [1,2]$

大于 $dp[2]$ 对应 $num[2]$, 则 $[1,2]+[4] = [1,2,4]$

$dp[3] = 3, [1,2,3]$

大于 $dp[3]$ 对应 $num[3]$, 则 $[1,2,3]+[4] = [1,2,3,4]$

$dp[4] = 1, [1]$

大于 $dp[4]$ 对应 $num[4]$, 则 $[1]+[4] = [1,4]$

$dp[5] = ?$

故最终 $dp[5] = 4$ 。

最终结果为 $dp[0], dp[1], ..., dp[i], ..., dp[n-1]$ 中的最大值。(与最大子段和相似之处)

例6:算法思路

设置**动态规划**数组 $dp[]$ ，第 i 个状态 $dp[i]$ 代表**以第 i 个元素结尾**的最长上升子序列的长度：

动态规划边界： $dp[0] = 1$ ；

初始化最长上升子序列的长度 $LIS = 1$ ；

从1到 $n-1$ ，循环 i ，计算 $dp[i]$ ：

从0至 $i-1$ ，循环 j ，若 $nums[i] > nums[j]$ ，说明 $nums[i]$ 可放置在 $nums[j]$ 的后面，组成最长上升子序列：

若 $dp[i] < dp[j] + 1$ ：

$dp[i] = dp[j] + 1$

LIS为 $dp[0], dp[1], \dots, dp[i], \dots, dp[n-1]$ 中**最大**的。

[1, 3, 2, 3, 1, 4]

$dp[0] = 1$

$dp[1]$:

$nums[1] = 3, > \text{nums}[0]$

$dp[1] = dp[0] + 1 = 2$ [1,3]

$dp[2]$:

$nums[2] = 2, > \text{nums}[0]$

$dp[2] = dp[0] + 1 = 2$ [1,2]

$dp[3]$:

$nums[3] = 3, > \text{nums}[0], \text{nums}[1], \text{nums}[2]$

$dp[3] = dp[2] + 1 = 3$ [1,2,3]

$dp[4]$:

$nums[4] = 1,$

$dp[4] = 1$ [1]

$dp[5]$:

$nums[5] = 4, > \text{nums}[0], \text{nums}[1], \text{nums}[2], \text{nums}[3], \text{nums}[4]$

$dp[5] = dp[3] + 1 = 4$ [1,2,3,4]

最终，选择 $dp[0], dp[1], dp[2], dp[3], dp[4], dp[5]$ 中最大的， $LIS = dp[5]$

例6:课堂练习

```
class Solution {
public:
    int lengthOfLIS(std::vector<int>& nums) {
        if (nums.size() == 0){
            return 0;
        }
        std::vector<int> dp(nums.size(), 0);
        dp[0] = 1;
        int LIS = 1;
        for (int i = 1; i < dp.size(); i++){
            1
            for (int j = 0; j < i; j++){
                if ( 2 ) {
                    dp[i] = dp[j] + 1;
                }
            }
            if (LIS < dp[i]){
                3
            }
        }
        return LIS;
    }
};
```

3分钟时间填写代码，
有问题随时提出！

例6:实现

```
class Solution {
public:
    int lengthOfLIS(std::vector<int>& nums) {
        if (nums.size() == 0) {
            return 0;
        }
        std::vector<int> dp(nums.size(), 0);
        dp[0] = 1;
        int LIS = 1;
        for (int i = 1; i < dp.size(); i++) {
            dp[i] = 1;
            for (int j = 0; j < i; j++) {
                if ( nums[i] > nums[j] && dp[i] < dp[j] + 1 ) {
                    dp[i] = dp[j] + 1;
                }
            }
            if (LIS < dp[i]) {
                LIS = dp[i];
            }
        }
        return LIS;
    }
};
```

例6:算法思路2

设置一个**栈**(使用vector实现)stack, $\text{stack}[i]$ 代表长度为 $i+1$ 的上升子序列最后一个元素的**最小可能取值**, 即若要组成长度为 $i+2$ 的上升子序列, 需要**一个大于 $\text{stack}[i]$** 的元素。最终栈的大小, 即为最长上升子序列长度。

[1, 3, 2, 3, 1, 4]

长度为1的上升子序列:[1]、[2]、[3]、[4]

长度为2的上升子序列:

[1,2]、[1,3]、[1,4]、[2,3]、[2,4]、[3,4]

长度为3的上升子序列: [1,2,3]、[1,2,4]、[2,3,4]

长度为4的上升子序列: [1,2,3,4]



例6:算法思路2

nums = [1, 3, 2, 3, 1, 4]

1. 设置一个**栈**(使用vector实现), 将nums[0] push栈中。

2. 从1至n-1**遍历**nums数组:

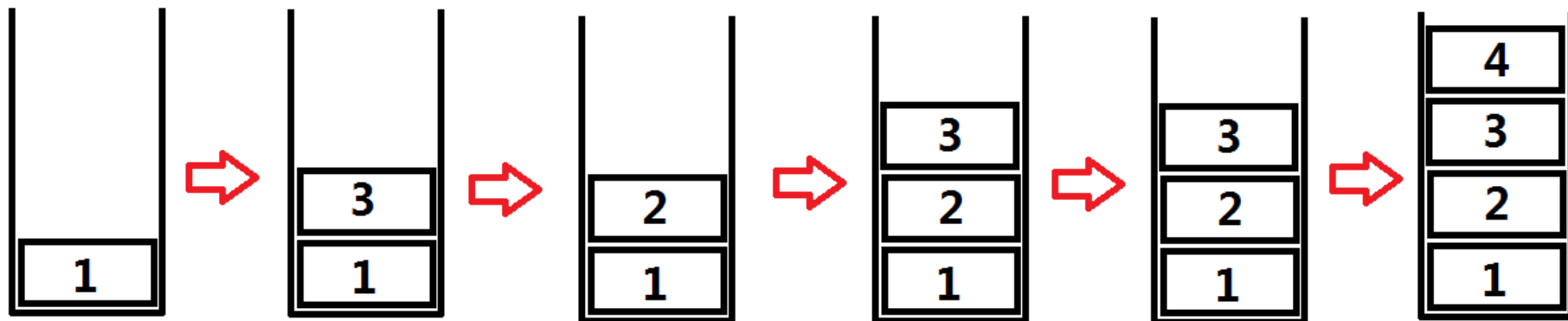
若nums[i] > 栈顶:将nums[i] push至栈中。

否则:

从**栈底遍历至栈顶**, 若遍历时, 栈中元素大于等于nums[i],

使用nums[i]**替换**该元素, 并**跳出**循环。

3. 返回栈的大小。



```

class Solution {
public:
    int lengthOfLIS(std::vector<int>& nums) {
        if (nums.size() == 0) {
            return 0;
        }
        std::vector<int> stack;
        stack.push_back(nums[0]);
        for (int i = 1; i < nums.size(); i++) {
            if ( 1 > stack.back()) {
                stack.push_back(nums[i]);
            }
            else {
                for (int j = 0; j < stack.size(); j++) {
                    if ( 2 ) {
                        3
                        break;
                    }
                }
            }
        }
        return stack.size();
    }
};

```

例6:课堂练习2

3分钟时间填写代码，
有问题随时提出！

例6:实现2

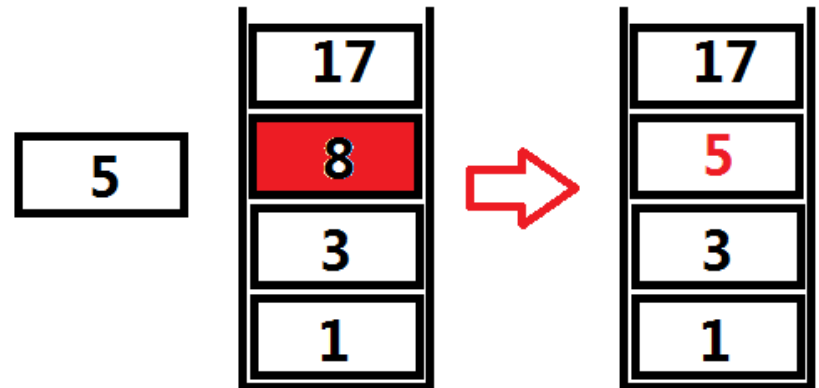
```
class Solution {
public:
    int lengthOfLIS(std::vector<int>& nums) {
        if (nums.size() == 0) {
            return 0;
        }
        std::vector<int> stack;
        stack.push_back(nums[0]);
        for (int i = 1; i < nums.size(); i++) {
            if (nums[i] > stack.back()) {
                stack.push_back(nums[i]);
            }
            else {
                for (int j = 0; j < stack.size(); j++) {
                    if (stack[j] >= nums[i]) {
                        stack[j] = nums[i];
                        break;
                    }
                }
            }
        }
        return stack.size();
    }
};
```

例6:实现2优化($n \cdot \log n$)

```
int binary_search(std::vector<int> nums, int target){
    int index = -1;
    int begin = 0;
    int end = nums.size() - 1;
    while (index == -1){
        int mid = (begin + end) / 2;
        if (target == nums[mid]){
            index = mid;
        }
        else if (target < nums[mid]){
            if (mid == 0 || target > nums[mid - 1]){
                index = mid;
            }
            end = mid - 1;
        }
        else if (target > nums[mid]){
            if (mid == nums.size() - 1 || target < nums[mid + 1]){
                index = mid + 1;
            }
            begin = mid + 1;
        }
    }
    return index;
}

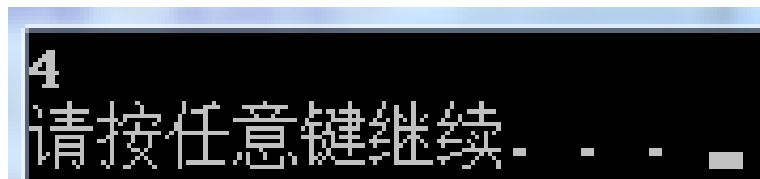
class Solution {
public:
    int lengthOfLIS(std::vector<int>& nums) {
        if (nums.size() == 0){
            return 0;
        }
        std::vector<int> stack;
        stack.push_back(nums[0]);
        for (int i = 1; i < nums.size(); i++){
            if (nums[i] > stack.back()){
                stack.push_back(nums[i]);
            }
            else{
                int pos = binary_search(stack, nums[i]);
                stack[pos] = nums[i];
            }
        }
        return stack.size();
    }
};
```

二分查找 5 该插入的位置！



例6:测试与leetcode提交结果

```
int main() {  
    int test[] = {1, 3, 2, 3, 1, 4};  
    std::vector<int> nums;  
    for (int i = 0; i < 6; i++) {  
        nums.push_back(test[i]);  
    }  
    Solution solve;  
    printf("%d\n", solve.lengthOfLIS(nums));  
    return 0;  
}
```



Longest Increasing Subsequence

Submission Details

24 / 24 test cases passed.

Status: **Accepted**

Runtime: 26 ms

Submitted: 0 minutes ago

Longest Increasing Subsequence

Submission Details

24 / 24 test cases passed.

Status: **Accepted**

Runtime: 3 ms

Submitted: 0 minutes ago

例7:最小路径和

已知一个**二维数组**，其中存储了**非负整数**，找到从**左上角**到**右下角**的一条路径，使得路径上的**和最小**。(移动过程中只能向下或向右)

```
class Solution {  
public:  
    int minPathSum(std::vector<std::vector<int>> & grid) {  
    }  
};
```

1	3	1
1	5	1
4	2	1

1	3	1
1	5	1
4	2	1

最小路径和为7

选自 **LeetCode 64. Minimum Path Sum**

<https://leetcode.com/problems/minimum-path-sum/description/>

难度:**Medium**

例7:思考

思考该题与**例-5三角形**的**相似**之处，如何设计动态规划算法，使得求得从**左上角**到**右下角**使得路径上的值最小的**最优解**？

设 $dp[i][j]$ 为到达位置 (i,j) 时的最优解(最小值)：

$dp[i][j]$ 与 $dp[i-1][j]$ 、 $dp[i][j-1]$ 、 $grid[i][j]$ 之间的关系是什么？

动态规划的边界条件是什么？

grid原始数组:

1	3	1
1	5	1
4	2	1

dp最优值(最小值数组):

1	4	?
2	?	?
?	?	?

递推方法:

	$(i-1,j)$?
$(i,j-1)$	(i,j)	?
?	?	?

例7:课堂练习

```
class Solution {
public:
    int minPathSum(std::vector<std::vector<int> >& grid) {
        if (grid.size() == 0) {
            return 0;
        }
        int row = grid.size();
        int column = grid[0].size();
        std::vector<std::vector<int> >
            dp(row, std::vector<int>(column, 0));

        dp[0][0] = 1

        for (int i = 1; i < column; i++) {
            dp[0][i] = 2
        }
        for (int i = 1; i < row; i++) {
            dp[i][0] = 3

            for (int j = 1; j < column; j++) {
                dp[i][j] = 4
            }
        }
        return 5
    }
};
```

5分钟时间填写代码，
有问题随时提出！

例7:实现

```
class Solution {
public:
    int minPathSum(std::vector<std::vector<int>> & grid) {
        if (grid.size() == 0) {
            return 0;
        }
        int row = grid.size();
        int column = grid[0].size();
        std::vector<std::vector<int>> >
            dp(row, std::vector<int>(column, 0));

        dp[0][0] = grid[0][0];

        for (int i = 1; i < column; i++) {
            dp[0][i] = dp[0][i-1] + grid[0][i];
        }
        for (int i = 1; i < row; i++) {
            dp[i][0] = dp[i-1][0] + grid[i][0];

            for (int j = 1; j < column; j++) {
                dp[i][j] = std::min(dp[i-1][j], dp[i][j-1]) + grid[i][j];
            }
        }
        return dp[row-1][column-1];
    };
};
```

grid原始数组:

1	3	1
1	5	1
4	2	1

$\min(2,4)+5$

1	4	5
2	7	?
3	?	?

dp最优值(最小值数组):

1	4	?
2	?	?
?	?	?

$\min(5,7)+1$

$\min(6,7)+2$

1	4	5
2	7	6
6	8	?

$\min(6,8)+1$

1	4	5
2	?	?
6	?	?

1	4	5
2	7	6
3	8	7

例7:测试与leetcode提交结果

```
int main() {
    int test[][3] = {{1,3,1}, {1,5,1}, {4,2,1}};
    std::vector<std::vector<int> > grid;
    for (int i = 0; i < 3; i++){
        grid.push_back(std::vector<int>());
        for (int j = 0; j < 3; j++){
            grid[i].push_back(test[i][j]);
        }
    }
    Solution solve;
    printf("%d\n", solve.minPathSum(grid));
    return 0;
}
```

Minimum Path Sum

Submission Details

61 / 61 test cases passed.

Status: Accepted

Runtime: 9 ms

Submitted: 0 minutes ago

?

请按任意键继续 . . .

例8:地牢游戏

已知一个**二维数组**，**左上角**代表**骑士**的位置，**右下角**代表**公主**的位置，二维数组中存储**整数**，**正数**可以给骑士**增加**生命值，**负数**会**减少**骑士的生命值，问骑士**初始时**至少是多少生命值，才可保证骑士在行走的过程中至少保持生命值为1。(骑士只能向下或向右行走)

```
class Solution {  
public:  
    int calculateMinimumHP(std::vector<std::vector<int> >& dungeon) {  
    }  
};
```

-2	-3	3
-5	-10	1
10	30	-5

K

-2	-3	3
-5	-10	1
10	30	-5

P

初始血量至少为7:

$$7 - 2 - 3 + 3 + 1 - 5 = 1$$

选自 **LeetCode 174. Dungeon Game**

<https://leetcode.com/problems/dungeon-game/description/>

难度:**Hard**

例8:思考

直接思考**动态规划**:

我们应该**从左上向右下**递推, 还是**从右下向左上**递推? 若用一个二维数组代表每个格子的状态, $dp[i][j]$ 具体代表什么?

若左上向右下递推:

可以递推出在每个格子**最多能获得**多少血量, 即 $dp[i][j]$ 代表骑士在位置 (i, j) 可以**积累**的最大血量, 能否转换成**初始时**至少是多少血量?

若右下向左上递推:

应如何设计动态规划状态, 二维数组 $dp[i][j]$ 代表了什么?

额外获得了28的血量:

$$-2-5+10+30-5=28$$

K

-2	-3	3
-5	-10	1
10	30	-5

P

初始血量至少为7:

$$7-2-3+3+1-5 = 1$$

K

-2	-3	3
-5	-10	1
10	30	-5

P

例8:算法思路

从左上向右下递推:

没办法将"每个格子最多能获得多少血量", **转换**成"初始时至少是多少血量"。

$$\begin{aligned} dp[0][0] &= \max(1, 1-dungeon[0][0]) \\ &= \max(1, 6) \\ &= 6 \end{aligned}$$

-5

从右下向左上递推:

$dp[i][j]$ 即代表若要达到右下角,至少有多少血量,能在行走的过程中至少保持生命值为1。

$$\begin{aligned} dp[0][0] &= \max(1, 1-dungeon[0][0]) \\ &= \max(1, 0) \\ &= 1 \end{aligned}$$

0

例如:

若代表地牢的二维数组为1*1的:

$$dp[0][0] = \max(1, 1-dungeon[0][0])$$

$$\begin{aligned} dp[0][0] &= \max(1, 1-dungeon[0][0]) \\ &= \max(1, -2) \\ &= 1 \end{aligned}$$

3

例8:算法思路

若代表地牢的二维数组为**1*n或n*1**的:

1*n, i从n-2至0:

$dp[0][i] = \max(1, dp[0][i+1] - \text{dungeon}[0][i])$

n*1, i从n-2至0:

$dp[i][0] = \max(1, dp[i+1][0] - \text{dungeon}[i][0])$

3	2	-5
---	---	----

$$dp[0][2] = \max(1, 1 - (-5)) = 6$$

$$dp[0][1] = \max(1, dp[0][2] - \text{dungeon}[0][1])$$

$$= \max(1, 6 - 2) = 4$$

$$dp[0][0] = \max(1, dp[0][1] - \text{dungeon}[0][0])$$

$$= \max(1, 4 - 3) = 1$$

?	?	?
?	?	6
?	4	6
1	4	6

例8:算法思路

若代表地牢的二维数组为 $n*m$ 的:

i 代表行, 从 $n-2$ 至 0 :

j 代表列, 从 $m-2$ 至 0 :

设 $dp_min = \min(dp[i+1][j], dp[i][j+1]);$

$dp[i][j] = \max(1, dp_min - \text{dungeon}[i][j]);$

-2	-3	3
-5	-10	1
10	30	-5

?	?	2
?	?	5
1	1	6

(1, 1)

$dp_min = \min(dp[2][1], dp[1][2])$

$= \min(1, 5)$

$= 1$

$dp[1][1] = \max(1, 1 - (-10))$

$= 11$

?	?	2
?	11	5
1	1	6

(0, 1)

$dp_min = \min(dp[0][2], dp[1][1])$

$= \min(2, 11)$

$= 2$

$dp[0][1] = \max(1, 2 - (-3))$

$= 5$

?	5	2
6	11	5
1	1	6

(1, 0)

$dp_min = \min(dp[2][0], dp[1][1])$

$= \min(1, 11)$

$= 1$

$dp[1][0] = \max(1, 1 - (-5))$

$= 6$

?	?	2
6	11	5
1	1	6

(0, 0)

$dp_min = \min(dp[0][1], dp[1][0])$

$= \min(5, 6)$

$= 5$

$dp[0][0] = \max(1, 5 - (-2))$

$= 7$

7	5	2
6	11	5
1	1	6

例8:课堂练习

```
class Solution {
public:
    int calculateMinimumHP(std::vector<std::vector<int> >& dungeon) {
        if (dungeon.size() == 0) {
            return 0;
        }
        std::vector<std::vector<int> >
            dp(dungeon.size(), std::vector<int>(dungeon[0].size(), 0));
        int row = dungeon.size();
        int column = dungeon[0].size();
        dp[row-1][column-1] = std::max(1, 1);
        for (int i = column-2; i>=0; i--) {
            dp[row-1][i] = std::max(1, 2);
        }
        for (int i = row-2; i>=0; i--) {
            dp[i][column-1] = std::max(1, 3);
        }
        for (int i = row-2; i>=0; i--) {
            for (int j = column-2; j>=0; j--) {
                int dp_min = std::min(4);
                dp[i][j] = std::max(5);
            }
        }
        return dp[0][0];
    }
};
```

5分钟时间填写代码，
有问题随时提出！

例8:实现

```
class Solution {
public:
    int calculateMinimumHP(std::vector<std::vector<int>> & dungeon) {
        if (dungeon.size() == 0) {
            return 0;
        }
        std::vector<std::vector<int>> dp(dungeon.size(), std::vector<int>(dungeon[0].size(), 0));
        int row = dungeon.size();
        int column = dungeon[0].size();
        dp[row-1][column-1] = std::max(1, 1-dungeon[row-1][column-1]);
        for (int i = column-2; i>=0; i--) {
            dp[row-1][i] = std::max(1, dp[row-1][i+1] - dungeon[row-1][i]);
        }
        for (int i = row-2; i>=0; i--) {
            dp[i][column-1] = std::max(1, dp[i+1][column-1] - dungeon[i][column-1]);
        }
        for (int i = row-2; i>=0; i--) {
            for (int j = column-2; j>=0; j--) {
                int dp_min = std::min(dp[i+1][j], dp[i][j+1]);
                dp[i][j] = std::max(1, dp_min - dungeon[i][j]);
            }
        }
        return dp[0][0];
    }
};
```

-2	-3	3
-5	-10	1
10	30	-5

?	?	2
?	?	5
1	1	6

?	?	2
?	11	5
1	1	6

例8:测试与leetcode提交结果

```
int main() {
    int test[][3] = {{-2, -3, 3}, {-5, -10, 1}, {10, 30, -5}};
    std::vector<std::vector<int> > dungeon;
    for (int i = 0; i < 3; i++) {
        dungeon.push_back(std::vector<int>());
        for (int j = 0; j < 3; j++) {
            dungeon[i].push_back(test[i][j]);
        }
    }
    Solution solve;
    printf("%d\n", solve.calculateMinimumHP(dungeon));
    return 0;
}
```

Dungeon Game

Submission Details

44 / 44 test cases passed.

Status: **Accepted**

Runtime: 6 ms

Submitted: 0 minutes ago

?

请按任意键继续. . .

结束

非常感谢大家！

林沐

联系我们

小象学院：互联网新技术在线教育领航者

— 微信公众号：**小象学院**

