

# 代码随想录 Chapter9 Dynamic Programming 动态规划（一）

## 动态规划理论基础

<https://programmerng.com/%E5%8A%A8%E6%80%81%E8%A7%84%E5%88%92%E7%90%86%E8%AE%BA%E5%9F%BA%E7%A1%>

### 1. 动态规划五部曲

- 1) 确定dp数组 (dp table) 和下标的含义
- 2) 确定递推公式 (状态转移方程)
- 3) 初始化dp数组
- 4) 确定遍历顺序
- 5) 举例推导dp数组检查

一些同学可能想为什么要先确定递推公式，然后在考虑初始化呢？因为一些情况是递推公式决定了dp数组要如何初始化！

可能刷过动态规划题目的同学可能都知道递推公式的重要性，感觉确定了递推公式这道题目就解出来了。

其实 确定递推公式 仅仅是解题里的一步而已！

一些同学知道递推公式，但搞不清楚dp数组应该如何初始化，或者正确的遍历顺序，以至于记下来公式，但写的程序怎么改都通过不了。

后序的讲解的大家就会慢慢感受到这五步的重要性了。

如果面试说英文：

1. Define the purpose of the dp array (dp table) and the meaning of its indices.
2. Determine the recurrence relation (state transition equation).
3. Initialize the dp array.
4. Decide on the traversal order.
5. Use an example to derive the dp array and verify correctness.

### 2. Code呈现的大致顺序

1. 特殊情况判断
2. 建立 dp array 或 table  
注意长度是 $n$ 还是 $n + 1$ ，都要从定义出发
3. Initialization
4. Recurrence step

包含三件事：

- (1) 遍历顺序（先遍历  $i$  还是先遍历  $j$ ？从后往前？还是从前往后？）
- (2) 每个遍历元素的取值范围（ $i$  什么范围？ $j$ ？）
- (3) recurrence relation

5. 最终返回什么

### 3. 动态规划题目总结

![Screenshot 2024-10-31 at 12.03.34 AM.png](attachment:Screenshot 2024-10-31 at 12.03.34 AM.png)

## 4. 动态规划应该如何debug

找问题的最好方式就是把dp数组打印出来，看看究竟是不是按照自己思路推导的！

做动规的题目，写代码之前一定要把状态转移在dp数组的上具体情况模拟一遍，心中有数，确定最后推出的是想要的结果。

发出这样的问题之前，其实可以自己先思考这三个问题：

- 这道题目我举例推导状态转移公式了么？
- 我打印dp数组的日志了么？
- 打印出来了dp数组和我想的一样么？

```
In [24]: from typing import List, Tuple, Dict
```

## Part 1 基础题目

### LC509 Fibonacci number (Easy)

## 509. Fibonacci Number

已解答

简单

相关标签

相关企业

文档

The **Fibonacci numbers**, commonly denoted  $F(n)$  form a sequence, called the **Fibonacci sequence**, such that each number is the sum of the two preceding ones, starting from  $0$  and  $1$ . That is,

$$\begin{aligned} F(0) &= 0, \quad F(1) = 1 \\ F(n) &= F(n - 1) + F(n - 2), \quad \text{for } n > 1. \end{aligned}$$

Given  $n$ , calculate  $F(n)$ .

#### Example 1:

**Input:**  $n = 2$

**Output:** 1

**Explanation:**  $F(2) = F(1) + F(0) = 1 + 0 = 1.$

#### Example 2:

**Input:**  $n = 3$

**Output:** 2

**Explanation:**  $F(3) = F(2) + F(1) = 1 + 1 = 2.$

#### Example 3:

**Input:**  $n = 4$

**Output:** 3

**Explanation:**  $F(4) = F(3) + F(2) = 2 + 1 = 3.$

简单题目是用来加深对解题方法论的理解的。这道题有很多种方法，同时也可以加深对于时间复杂度的理解。

注：本题面试常考，解析里也总结了quant fin prep的内容。

## 法1: recursion 递归

```
In [2]: def Fib_recursion(n:int) -> int:
    if n == 0:
        return 0
    if n == 1:
        return 1
    if n > 1:
        return Fib_recursion(n-1) + Fib_recursion(n-2)

# Given n, return F(n)
n = 10
print(Fib_recursion(n))
```

55

时间复杂度:  $\text{red} \mathcal{O}(2^N)$ 空间复杂度:  $O(N)$ 

The implementation above makes the calls to intermediate steps repeatedly and is inefficient.

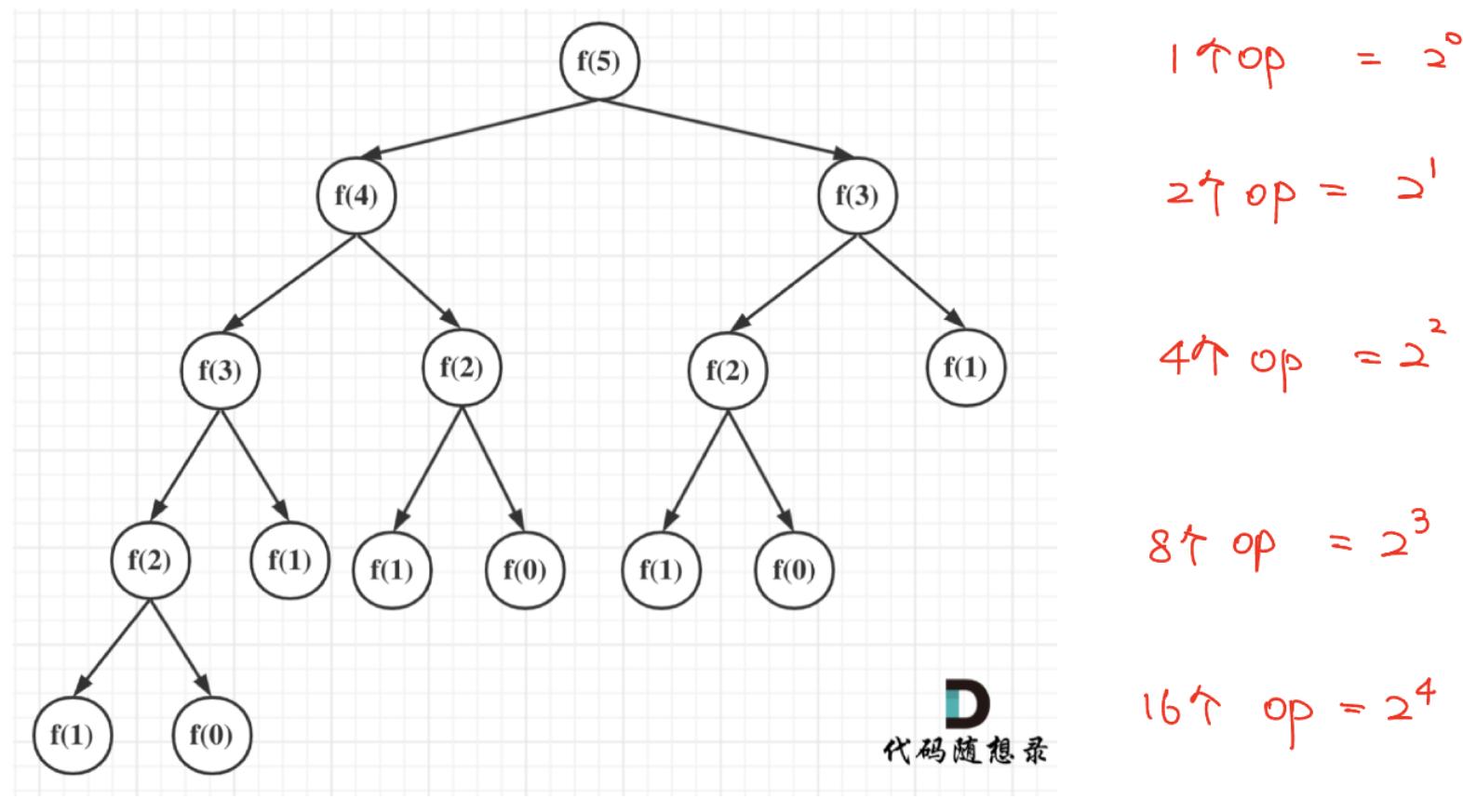
关于时间复杂度和空间复杂度的分析:

<https://programmercarl.com/%E5%89%8D%E5%BA%8F/%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6.html>

我们这里只讲解递归算法的时间复杂度是多少，空间复杂度请看网站。

在讲解递归时间复杂度的时候，我们提到了递归算法的时间复杂度本质上是要看: 递归的次数 \* 每次递归的时间复杂度。

可以看出上面的代码每次递归都是 $O(1)$ 的操作。再来看递归了多少次，这里将 $i=5$ 作为输入的递归过程抽象成一棵递归树，如图：



在这棵二叉树中每一个节点都是一次递归，那么这棵树有多少个节点呢？

我们之前也有说到，一棵深度为 $k$ 的二叉树最多可以有 $2^k - 1$ 个节点。

所以该递归算法的时间复杂度为 $O(2^N)$ ，这个复杂度是非常大的，随着 $n$ 的增大，耗时是指数上升的。

```
In [3]: # 如果让你print出前n个Fibonacci数列
def Fibseq(n):
```

```

res = []
for i in range(1,n+1): #默认range(start=0, end=end-1, step=1)
    res.append(Fib_recursion(i))
return res

# test
n = 10
print(Fibseq(n))

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

```

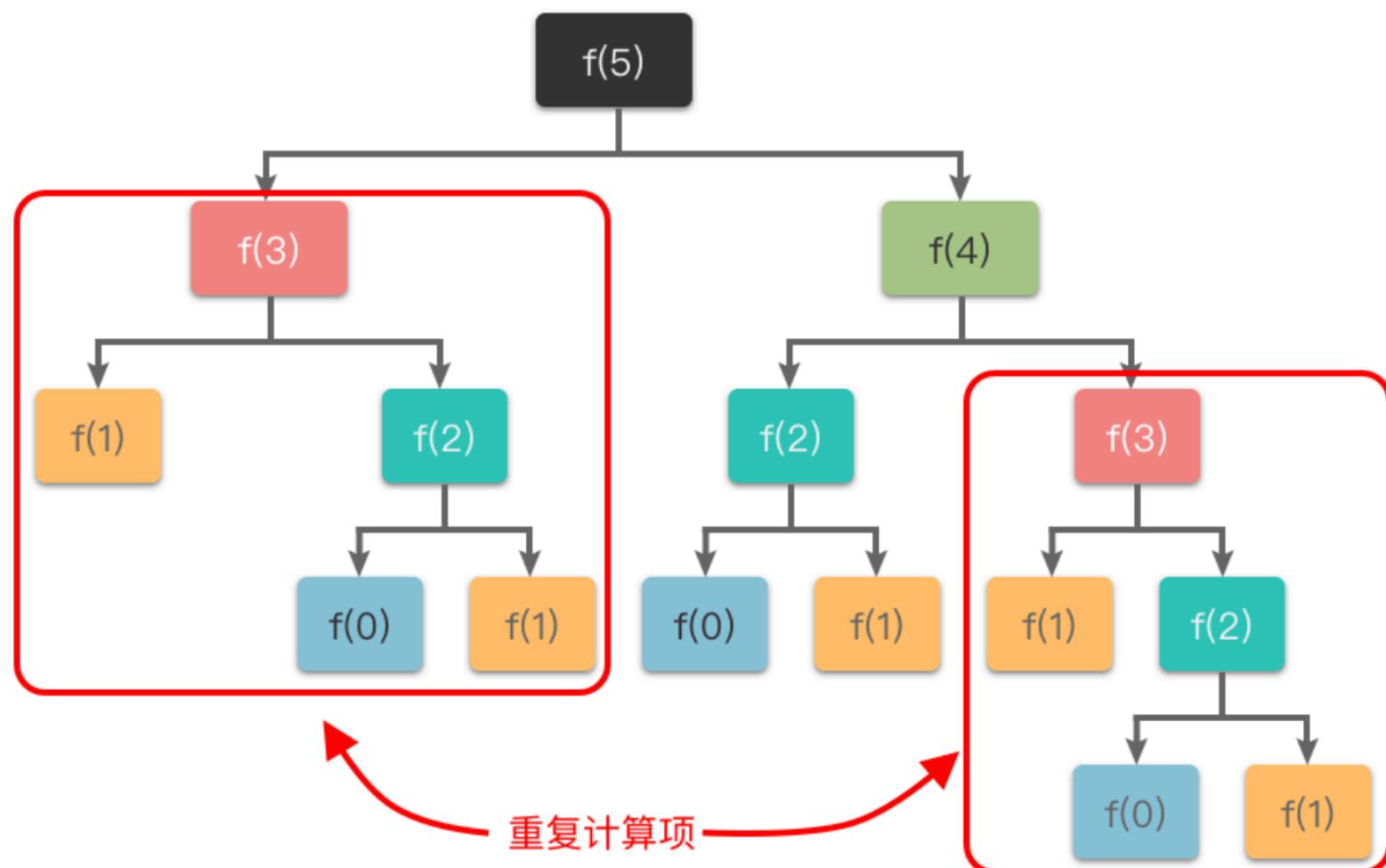
所以这种求斐波那契数的算法看似简洁，其实时间复杂度非常高，一般不推荐这样来实现斐波那契。

可不可以优化一下这个递归算法呢。主要是减少递归的调用次数。

## 法2: Memoization search

### 记忆化搜索 Memoization search

举个例子，比如「斐波那契数列」的定义是： $f(0) = 0, f(1) = 1, f(n) = f(n - 1) + f(n - 2)$ 。如果我们使用递归算法求解第  $n$  个斐波那契数，则对应的递推过程如下：



i	0	1	2	3	4	5	6	7	8	9	.....
$f(i)$	1	1	2	3	5	8	13	21	34	55	.....

从图中可以看出：如果使用普通递归算法，想要计算  $f(5)$ ，需要先计算  $f(3)$  和  $f(4)$ ，而在计算  $f(4)$  时还需要计算  $f(3)$ 。这样  $f(3)$  就进行了多次计算，同理  $f(0)、f(1)、f(2)$  都进行了多次计算，从而导致了重复计算问题。

为了避免重复计算，在递归的同时，我们可以使用一个缓存（数组或哈希表）来保存已经求解过的  $f(k)$  的结果。如上图所示，当递归调用到  $f(k)$  时，先查看一下之前是否已经计算过结果，如果已经计算过，则直接从缓存中取值返回，而不用再递推下去，这样就避免了重复计算问题。

There are multiple ways to implement **memoization search**.

### 1. A general framework

```
In [ ]: # 版本一
class Solution:
```

```

def fib(self, n: int) -> int:
    # Use array to store the intermediate results
    memo = [0 for _ in range(n + 1)]
    return self.my_fib(n, memo)

def my_fib(self, n: int, memo: List[int]) -> int:
    if n == 0:
        return 0
    if n == 1:
        return 1

    # 已经计算过结果
    if memo[n] != 0:
        return memo[n]

    # 没有计算过结果
    memo[n] = self.my_fib(n - 1, memo) + self.my_fib(n - 2, memo)
    return memo[n]

```

2. Since this question is fairly simple, we have the following simplified version that stores the intermediate results in dictionary

```

In [ ]: # 版本二
def fib_growing_dict(n: int) -> int:
    if n < 0:
        raise ValueError("n must be non-negative integer")

    result = {0: 0, 1: 1} # base case

    for i in range(2, n + 1):
        result[i] = result[i - 1] + result[i - 2]

    return result[n]

```

Another way can be to store the intermediate results in a list:

```

In [ ]: # 版本三
def fib_growing_list(n: int) -> int:
    if n < 0:
        raise ValueError("n must be non-negative integer")

    result = [0, 1] # base case

    for i in range(2, n + 1):
        result.append(result[i - 1] + result[i - 2])

    return result[n]

```

3. Memoization using built-in decorators from `functools`

`functools` library has `@lru_cache()` decorator, which implements memoization:

```

In [ ]: # 版本四

from functools import lru_cache
@lru_cache(128)
# default is to cache 128 most recently used calls,
# can set maxsize=None to indicate that cache never expires

def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 2) + fibonacci(n - 1)

# test
n = 10
print(fibonacci(50))

```

12586269025

- In the above example, a call to the function `fibonacci(n)` will be checked against a cache maintained by the decorator
- If the argument `n` is in cache, stored result is returned else it is computed and stored in the cache
- For large numbers, above calculation can be over 1000 times faster

时间复杂度:  $\text{O}(n)$

- 使用 `@lru_cache` 装饰器后，这个递归函数 `fibonacci` 的时间复杂度降低到了  $O(n)$ 。
- 每个 `fibonacci(n)` 只计算一次，然后结果被缓存起来。因此，每个值从 `fibonacci(0)` 到 `fibonacci(n)` 只会计算一次，避免了重复计算。

空间复杂度： $O(n)$

### 法3: 优化递归

```
In [7]: def fib2(n:int) -> int:
    if n == 0:
        return 0

    if n == 1:
        return 1

    elif n > 1:
        prev, curr = 0, 1
        for i in range(2, n+1):
            prev, curr = curr, prev + curr
    return curr

# test
n = 10
print(fib2(n))
```

55

时间复杂度： $\textcolor{red}{O}(n)$

空间复杂度： $O(1)$

Explanation:

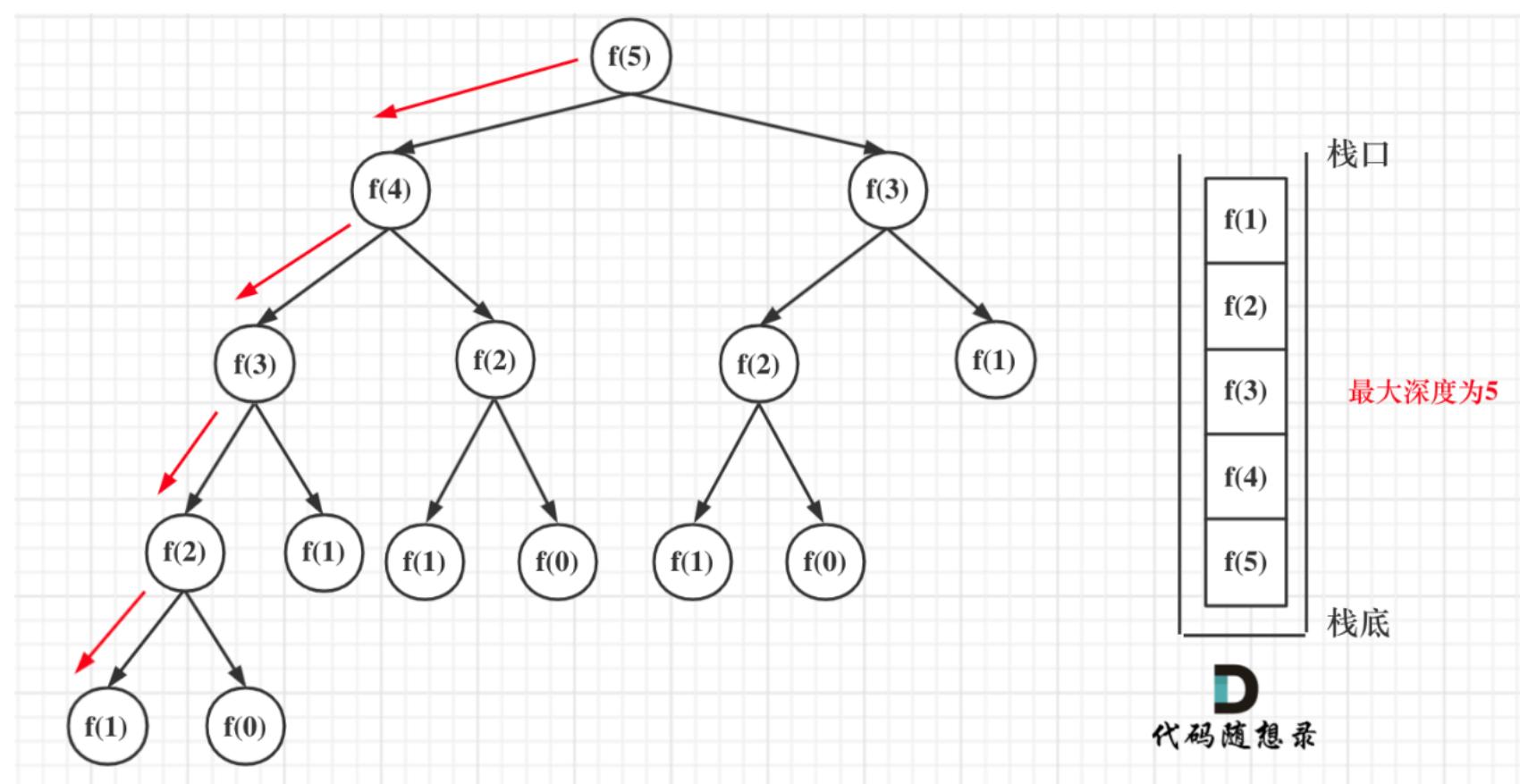
- The function computes the  $n$ -th Fibonacci number iteratively by updating two variables (`prev` and `curr`) in a loop.
- The loop runs from `2` to `n`, meaning it iterates  $n - 1$  times.
- Each iteration performs a constant amount of work (updating the values of `prev` and `curr`), so the time complexity is proportional to  $n$ .

说完了这段递归代码的时间复杂度，再看看如何求其空间复杂度呢，这里给大家提供一个公式：

$$\text{递归算法的空间复杂度} = \text{每次递归的空间复杂度} * \text{递归深度}$$

此时可以分析这段递归的空间复杂度，从代码中可以看出每次递归所需要的空间大小都是一样的，所以每次递归中需要的空间是一个常量，并不会随着n的变化而变化，每次递归的空间复杂度就是 $O(1)$ 。

在看递归的深度是多少呢？如图所示：



递归第n个斐波那契数的话，递归调用栈的深度就是n。

那么每次递归的空间复杂度是 $O(1)$ ，调用栈深度为n，所以这段递归代码的空间复杂度就是 $O(n)$ 。

### 法3：动态规划

动规五部曲：

- 确定dp数组以及下标的含义

`dp[i]` 的定义为：第i个数的斐波那契数值是 `dp[i]`

- 确定递推公式

状态转移方程 `dp[i] = dp[i - 1] + dp[i - 2]`

- dp数组如何初始化

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

- 确定遍历顺序

从递归公式`dp[i] = dp[i - 1] + dp[i - 2]`中可以看出，`dp[i]`是依赖 `dp[i - 1]` 和 `dp[i - 2]`，那么遍历的顺序一定是从前到后遍历

- 举例推导dp数组

按照这个递推公式`dp[i] = dp[i - 1] + dp[i - 2]`，我们来推导一下，当N为10的时候，dp数组应该是如下的数列： 0 1 1 2 3 5 8 13 21 34 55

```
In [8]: # 动态规划 (版本1)
class Solution:
    def fib(self, n: int) -> int:

        # 排除 Corner Case
        if n == 0:
            return 0

        # 创建 dp table
        dp = [0] * (n + 1)

        # 初始化 dp 数组
        dp[0] = 0
```

```

dp[1] = 1

# 遍历顺序: 由前向后。因为后面要用到前面的状态。
for i in range(2, n + 1):

    # 确定递归公式/状态转移公式
    dp[i] = dp[i - 1] + dp[i - 2]

    # 返回答案
    return dp[n]

# test
n = 10
object = Solution()
print(object.fib(n))

```

55

时间复杂度:  $\mathcal{O}(n)$ 空间复杂度:  $O(n)$ 

```

In [9]: # 动态规划 (版本2)
class Solution:
    def fib(self, n: int) -> int:
        if n <= 1:
            return n

        dp = [0, 1]

        for i in range(2, n + 1):
            total = dp[0] + dp[1]
            dp[0] = dp[1]
            dp[1] = total

        return dp[1]

# test
n = 10
object = Solution()
print(object.fib(n))

```

55

时间复杂度:  $\mathcal{O}(n)$ 空间复杂度:  $O(1)$ 

这个算法只使用了两个变量  $dp[0]$  和  $dp[1]$  来存储前两个斐波那契数，因此所需的额外空间是常数，空间复杂度为  $O(1)$ 。不需要使用一个长度为  $n+1$  的数组来存储所有中间结果，这里只需要存储两个变量。

## LC70 Climbing stairs (easy)

# 70. Climbing Stairs

已解答

**简单**

相关标签

相关企业

提示

文档

You are climbing a staircase. It takes  $n$  steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

## Example 1:

**Input:**  $n = 2$

**Output:** 2

**Explanation:** There are two ways to climb to the top.

1. 1 step + 1 step
2. 2 steps

## Example 2:

**Input:**  $n = 3$

**Output:** 3

**Explanation:** There are three ways to climb to the top.

1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step

思路: <https://programmercarl.com/0070.%E7%88%AC%E6%A5%BC%E6%A2%AF.html#%E6%80%9D%E8%B7%AF>

1. 确定dp数组以及下标的含义

$dp[i]$ : 爬到第*i*层楼梯，有 $dp[i]$ 种方法

2. 递推公式

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

3. dp数组如何初始化

需要注意的是：题目中说了*n*是一个正整数，题目根本就没说*n*有为0的情况。

所以本题其实就不应该讨论 $dp[0]$ 的初始化！

我相信 $dp[1] = 1$ ,  $dp[2] = 2$ , 这个初始化大家应该都没有争议的。

所以我的原则是：不考虑 $dp[0]$ 如何初始化，只初始化 $dp[1] = 1$ ,  $dp[2] = 2$ , 然后从*i* = 3开始递推，这样才符合 $dp[i]$ 的定义。

4. 遍历顺序

从前向后

5. 举例推导dp数组

Eg.  $n = 5$

Index  $i$ :

$i$	0	1	2	3	4	5
$dp[i]$	0	1	2	3	5	8

↓  
ignore

return  $dp[n]$

```
In [10]: class Solution:
    def climbStairs(self, n: int) -> int:
        # 定义:  $dp[i]$  is number of ways to reach the  $i$ -th step
        if n == 1: return 1
        if n == 2: return 2

        # 不考虑  $dp[0]$  的初始化, 因为  $1 \leq n \leq 45$ 
        # 只初始化  $dp[1] = 1$ ,  $dp[2] = 2$ , 然后从  $i = 3$  开始递推, 这样才符合  $dp[i]$  的定义
        dp = [0 for _ in range(n+1)]
        dp[1] = 1
        dp[2] = 2

        for i in range(3, n + 1):
            ##### $dp[i]##### = dp[i-1] + dp[i-2]

        return dp[n]

# test
n = 5
object = Solution()
print(object.climbStairs(n))
```

8

时间复杂度:  $O(n)$

空间复杂度:  $O(n)$

## LC746 min cost climbing stairs (easy)

# 746. Min Cost Climbing Stairs

已解答 [简单](#)[相关标签](#)[相关企业](#)[提示](#)[文](#)

You are given an integer array `cost` where `cost[i]` is the cost of  $i^{\text{th}}$  step on a staircase. Once you pay the cost, you can either climb one or two steps.

You can either start from the step with index `0`, or the step with index `1`.

Return *the minimum cost to reach the top of the floor.*

## Example 1:

**Input:** `cost = [10, 15, 20]`

**Output:** 15

**Explanation:** You will start at index 1.

- Pay 15 and climb two steps to reach the top.

The total cost is 15.

## Example 2:

**Input:** `cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]`

**Output:** 6

**Explanation:** You will start at index 0.

- Pay 1 and climb two steps to reach index 2.
- Pay 1 and climb two steps to reach index 4.
- Pay 1 and climb two steps to reach index 6.
- Pay 1 and climb one step to reach index 7.
- Pay 1 and climb two steps to reach index 9.
- Pay 1 and climb one step to reach the top.

The total cost is 6.

## 思路

1. 确定dp数组以及下标的含义

`dp[i]`的定义：到达第*i*台阶所花费的最少体力为`dp[i]`。

2. 递推公式

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

3. dp数组如何初始化

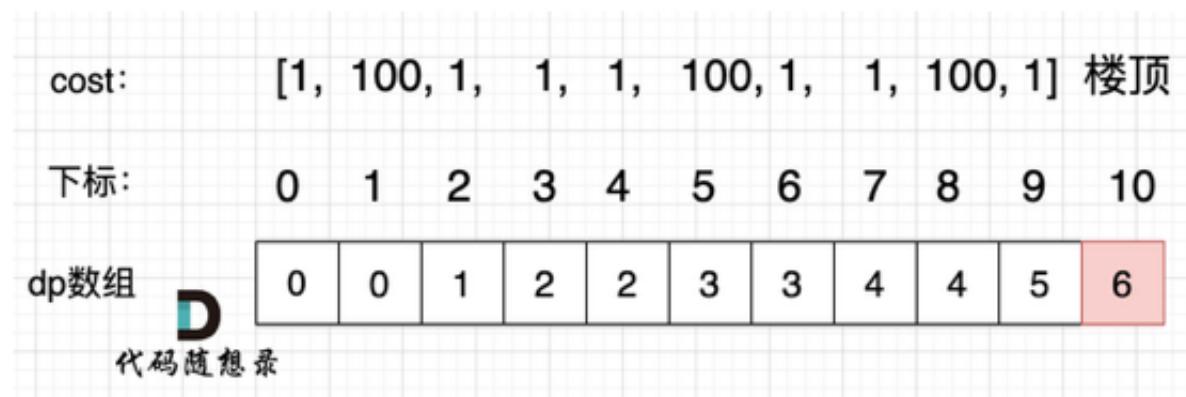
新题目描述中明确说了“你可以选择从下标为 0 或下标为 1 的台阶开始爬楼梯。”也就是说到达第 0 个台阶是不花费的，但从第 0 个台阶往上跳的话，需要花费 `cost[0]`。`dp[0] = 0, dp[1] = 0`

4. 遍历顺序

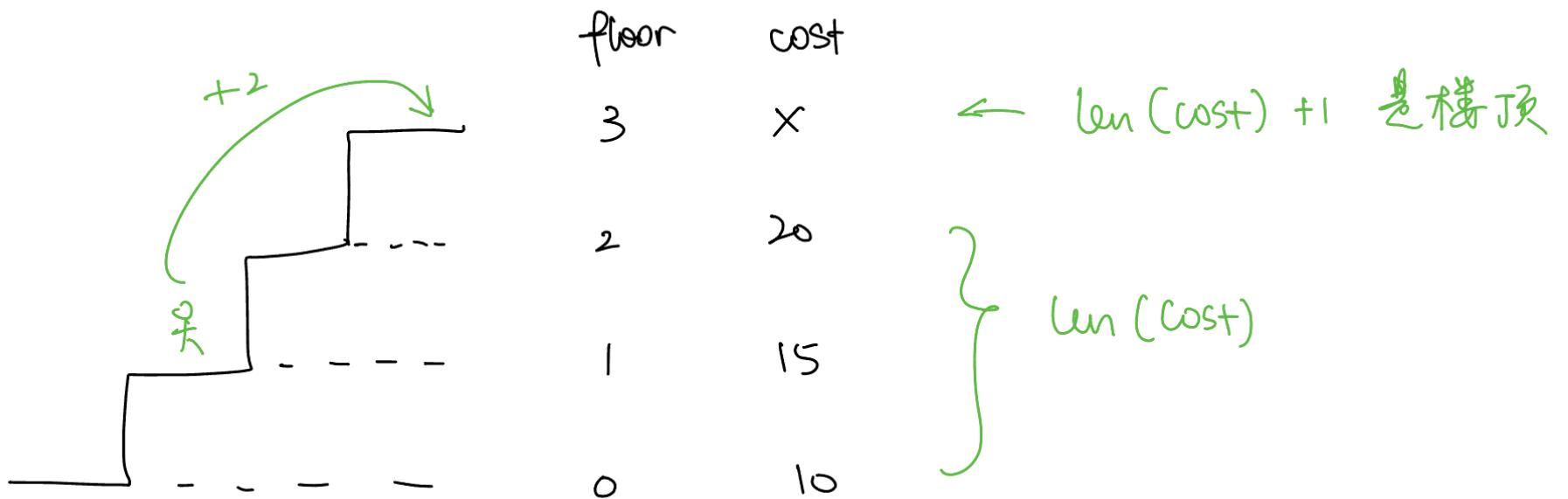
从前到后遍历`cost`数组

5. 举例推导dp数组

拿示例2: cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1] , 来模拟一下dp数组的状态变化, 如下:



提示: len(cost) + 1 是楼顶, 所以dp数组的长度应该为len(cost) + 1。拿Example 1 举例: cost = [10, 15, 20]



```
In [33]: from typing import List

class Solution:
    def minCostClimbingStairs(self, cost: List[int]) -> int:
        # dp[i] 的定义: 到达第i台阶所花费的最少cost。i: from 0 to len(cost)
        dp = [0 for _ in range(len(cost)+1)]

        # initialize
        dp[0], dp[1] = 0, 0

        # len(cost) + 1 is roof (the final stage)
        for i in range(2, len(cost)+1):
            dp[i] = min(dp[i-1]+cost[i-1], dp[i-2]+cost[i-2])

        return dp[len(cost)]

# test
cost = [10, 15, 20]
object = Solution()
print(object.minCostClimbingStairs(cost))
```

15

Time complexity:  $O(n)$ , where  $n$  is the length of the `cost` array

- This is because there's a single for-loop that iterates from 2 to  $n + 1$ , performing constant-time operations in each iteration.

Space complexity:  $O(n)$

## LC62 unique paths (medium)

## 62. Unique Paths

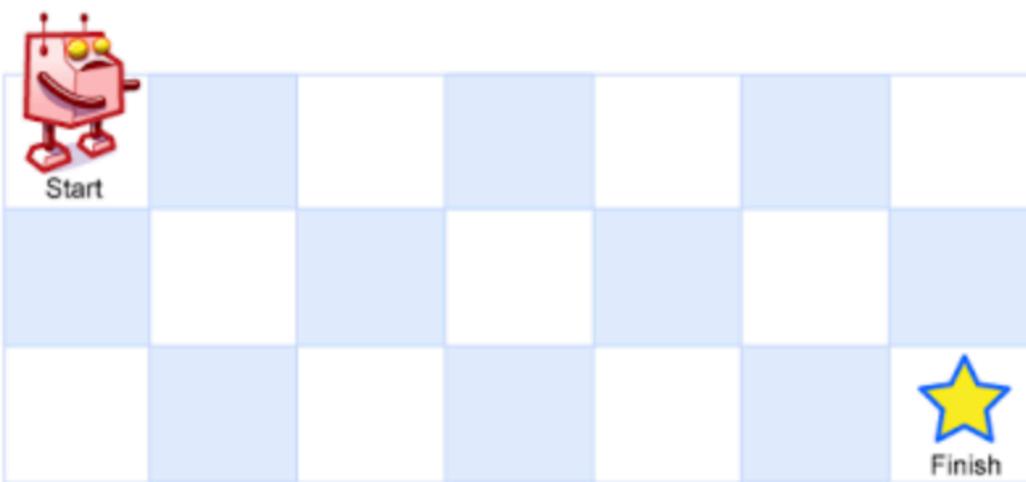
已解答 
[中等](#) [相关标签](#) [相关企业](#) [文](#)

There is a robot on an  $m \times n$  grid. The robot is initially located at the **top-left corner** (i.e.,  $\text{grid}[0][0]$ ). The robot tries to move to the **bottom-right corner** (i.e.,  $\text{grid}[m - 1][n - 1]$ ). The robot can only move either down or right at any point in time.

Given the two integers  $m$  and  $n$ , return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

The test cases are generated so that the answer will be less than or equal to  $2 * 10^9$ .

### Example 1:



**Input:**  $m = 3$ ,  $n = 7$

**Output:** 28

```
In [30]: m = 3
n = 7
dp = [[0 for _ in range(n)] for _ in range(m)] # n columns, m rows 口诀: 列里行外
print(dp)

# To get the dimensions:
rows = len(dp)           # Number of rows (m)
columns = len(dp[0])      # Number of columns (n)

print("Rows:", rows)
print("Columns:", columns)
```

```
[[0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0]]
Rows: 3
Columns: 7
```

$dp[i][j]$  : 第*i*个subarray里面的第*j*个元素

$dp[i][j]$  : 代表第*i*行第*j*列格子

```
In [13]: # dp[i][j] denotes how many ways to arrive at (i,j)
# 给dp table赋值的过程

for i in range(m):
    dp[i][0] = 1
print(dp)

for j in range(n):
    dp[0][j] = 1
print(dp)

for i in range(1, m):          #i cannot be 0 b/c it has already been initialized
    for j in range(1, n):      #j cannot be 0 b/c it has already been initialized
        dp[i][j] = dp[i-1][j] + dp[i][j-1]
```

```

print(dp)

[[1, 0, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0, 0]
 [[1, 1, 1, 1, 1, 1, 1], [1, 0, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0, 0]
 [[1, 1, 1, 1, 1, 1, 1], [1, 2, 3, 4, 5, 6, 7], [1, 3, 6, 10, 15, 21, 28]]]

In [14]: # final answer is
dp[m-1][n-1]

```

Out[14]: 28



```

In [32]: class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        dp = [ [0 for _ in range(n)] for _ in range(m)] # n cols, m rows

        # initialization
        for i in range(m):
            dp[i][0] = 1
        for j in range(n):
            dp[0][j] = 1

        # recurrence relation
        for i in range(1, m): # i cannot be 0 b/c it has already been initialized
            for j in range(1, n): # j cannot be 0 b/c it has already been initialized
                dp[i][j] = dp[i-1][j] + dp[i][j-1]

        return dp[m-1][n-1]

    # test
    object = Solution()
    print(object.uniquePaths(3, 7))

```

28

时间复杂度:  $O(m \times n)$ 空间复杂度:  $O(m \times n)$  $m, n$  就是题目中的行数和列数

## LC63 Unique Paths II (medium)

## 63. Unique Paths II

已解答 [中等](#) [相关标签](#) [相关企业](#) [提示](#) [贡献](#)

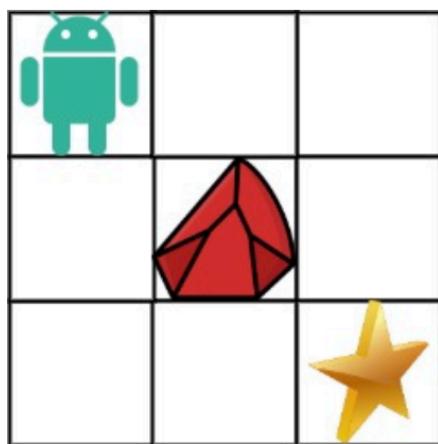
You are given an  $m \times n$  integer array `grid`. There is a robot initially located at the **top-left corner** (i.e., `grid[0][0]`). The robot tries to move to the **bottom-right corner** (i.e., `grid[m - 1][n - 1]`). The robot can only move either down or right at any point in time.

An obstacle and space are marked as `1` or `0` respectively in `grid`. A path that the robot takes cannot include **any** square that is an obstacle.

Return the *number of possible unique paths* that the robot can take to reach the bottom-right corner.

The testcases are generated so that the answer will be less than or equal to  $2 * 10^9$ .

**Example 1:**



**Input:** `obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]`

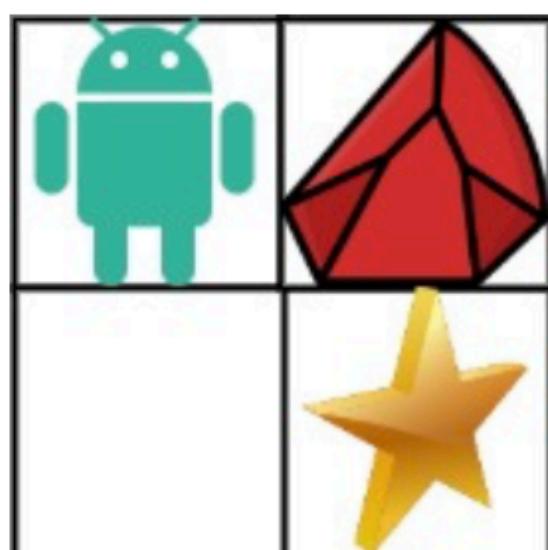
**Output:** `2`

**Explanation:** There is one obstacle in the middle of the 3x3 grid above.

There are two ways to reach the bottom-right corner:

1. Right  $\rightarrow$  Right  $\rightarrow$  Down  $\rightarrow$  Down
2. Down  $\rightarrow$  Down  $\rightarrow$  Right  $\rightarrow$  Right

## Example 2:



**Input:** obstacleGrid = [[0,1], [0,0]]

**Output:** 1

## Constraints:

- `m == obstacleGrid.length`
- `n == obstacleGrid[i].length`
- `1 <= m, n <= 100`
- `obstacleGrid[i][j]` is `0` or `1`.

备注：从现在开始，我只写简单的思路提示，具体细节请见代码随想录。

### 思路

62.不同路径 中我们已经详细分析了没有障碍的情况，有障碍的话，其实就是标记对应的dp table (dp数组) 保持初始值(0)就可以了。

#### 动规五部曲：

1. 确定dp数组 (dp table) 以及下标的含义

`dp[i][j]`: 表示从 (0, 0) 出发，到 (i, j) 有 `dp[i][j]` 条不同的路径。

2. 确定递推公式

递推公式和62.不同路径一样， $dp[i][j] = dp[i - 1][j] + dp[i][j - 1]$ 。

但这里需要注意一点，因为有了障碍，(i, j) 如果就是障碍的话应该路径数量变成初始状态0。

```
if obstacleGrid[i][j] == 1:
    dp[i][j] = 0
else:
    dp[i][j] = dp[i-1][j] + dp[i][j-1]
```

3. dp数组如何初始化

在62.不同路径不同路径中我们给出如下的初始化：

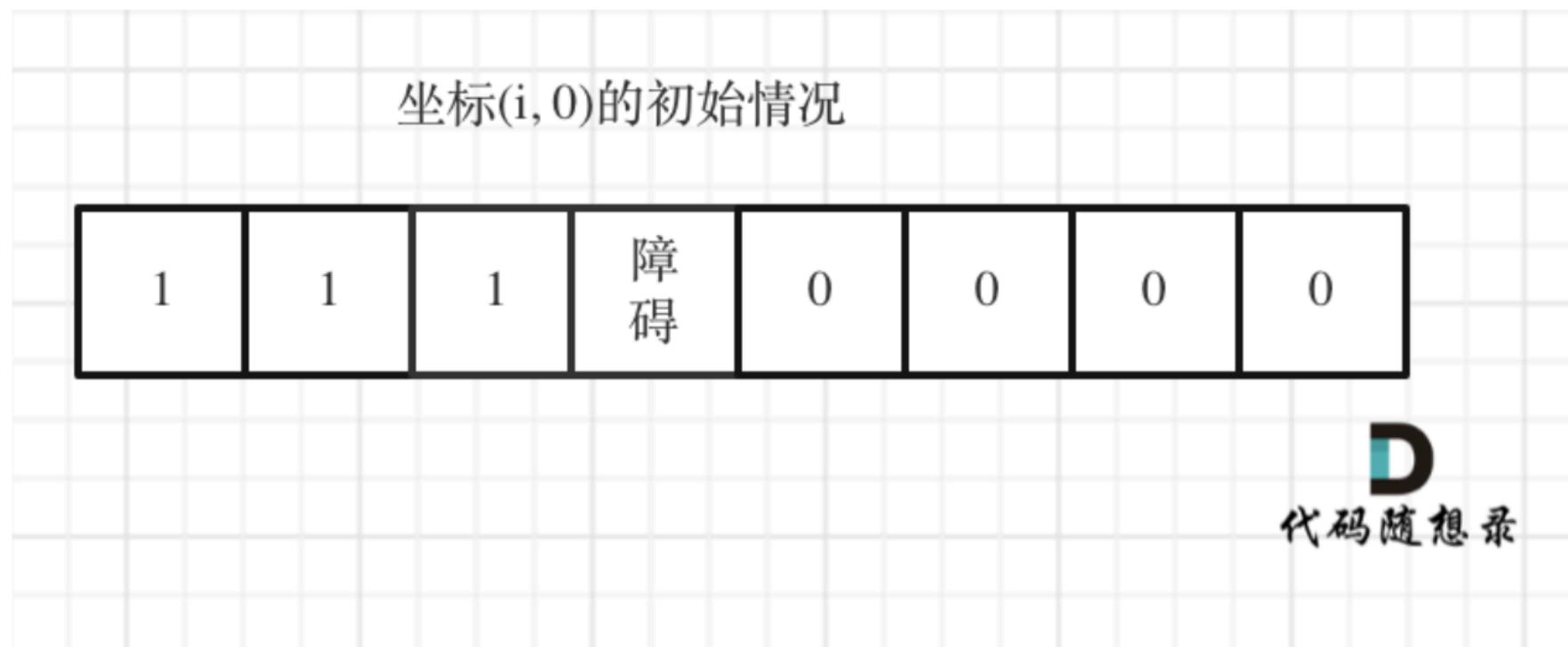
```
for i in range(m):
    dp[i][0] = 1
    for j in range(n):
```

`dp[0][j] = 1`

因为从  $(0,0)$  的位置到  $(i,0)$  的路径只有一条，所以  $dp[i][0]$  一定为1， $dp[0][j]$  也同理。

但如果  $(i,0)$  这条边有了障碍之后，障碍之后（包括障碍）都是走不到的位置了，所以障碍之后的  $dp[i][0]$  应该还是初始化0。

如图：



所以本题的初始化为：

```
#initialize
for i in range(m):
    dp[i][0] = 1
    if obstacleGrid[i][0] == 1: #once encounter obstacle, 该列后面所有的都无法到达
        dp[i][0] = 0
        break

for j in range(n):
    dp[0][j] = 1
    if obstacleGrid[0][j] == 1: #once encounter obstacle, 该行后面所有的都无法到达
        dp[0][j] = 0
        break
```

#### 4. 确定遍历顺序

从递归公式  $dp[i][j] = dp[i - 1][j] + dp[i][j - 1]$  中可以看出，一定是从左到右一层一层遍历，这样保证推导  $dp[i][j]$  的时候， $dp[i - 1][j]$  和  $dp[i][j - 1]$  一定是有数值。

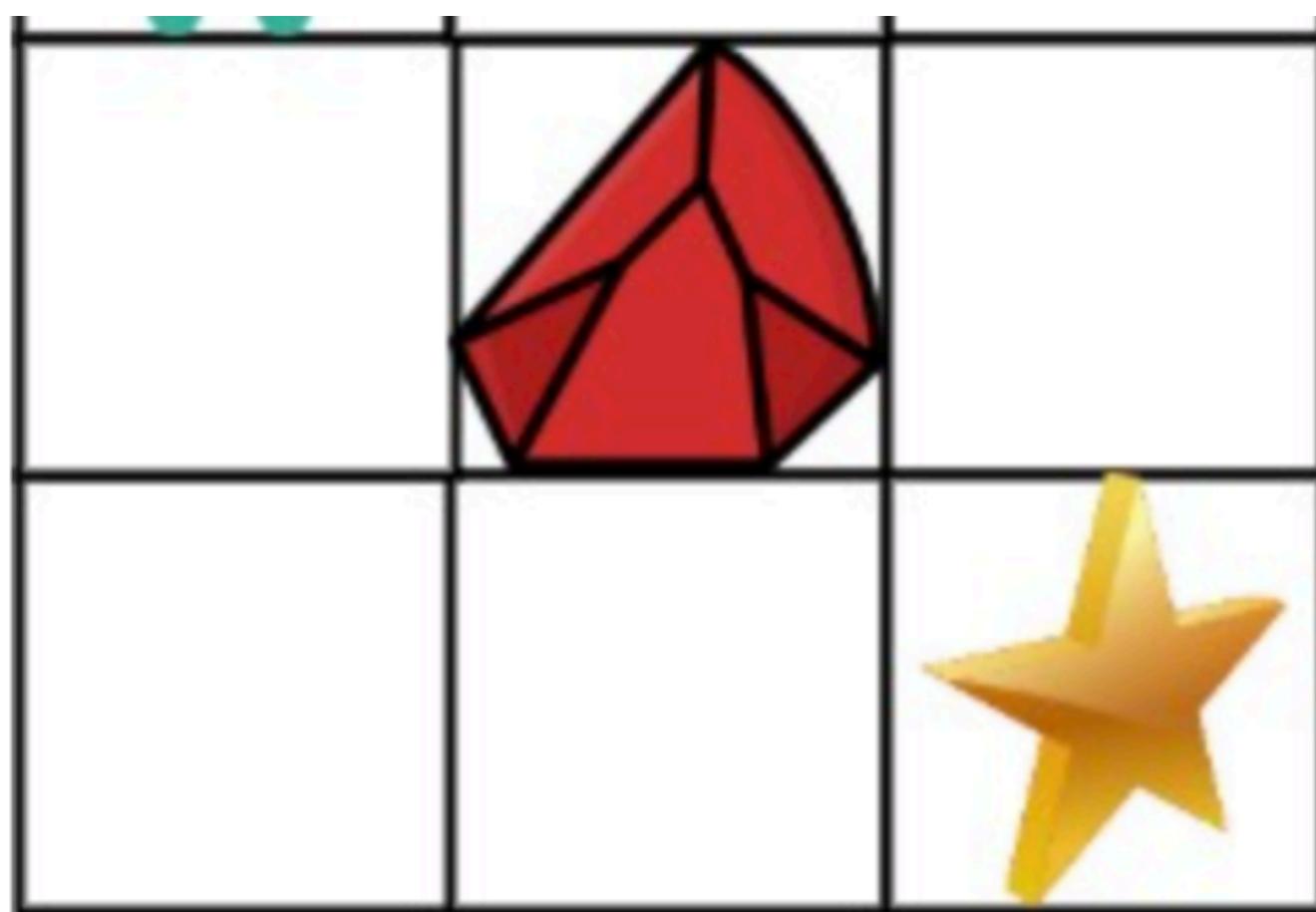
本题 **先行后列** 和 **先列后行** 是一样的。

#### 5. 举例推导dp数组

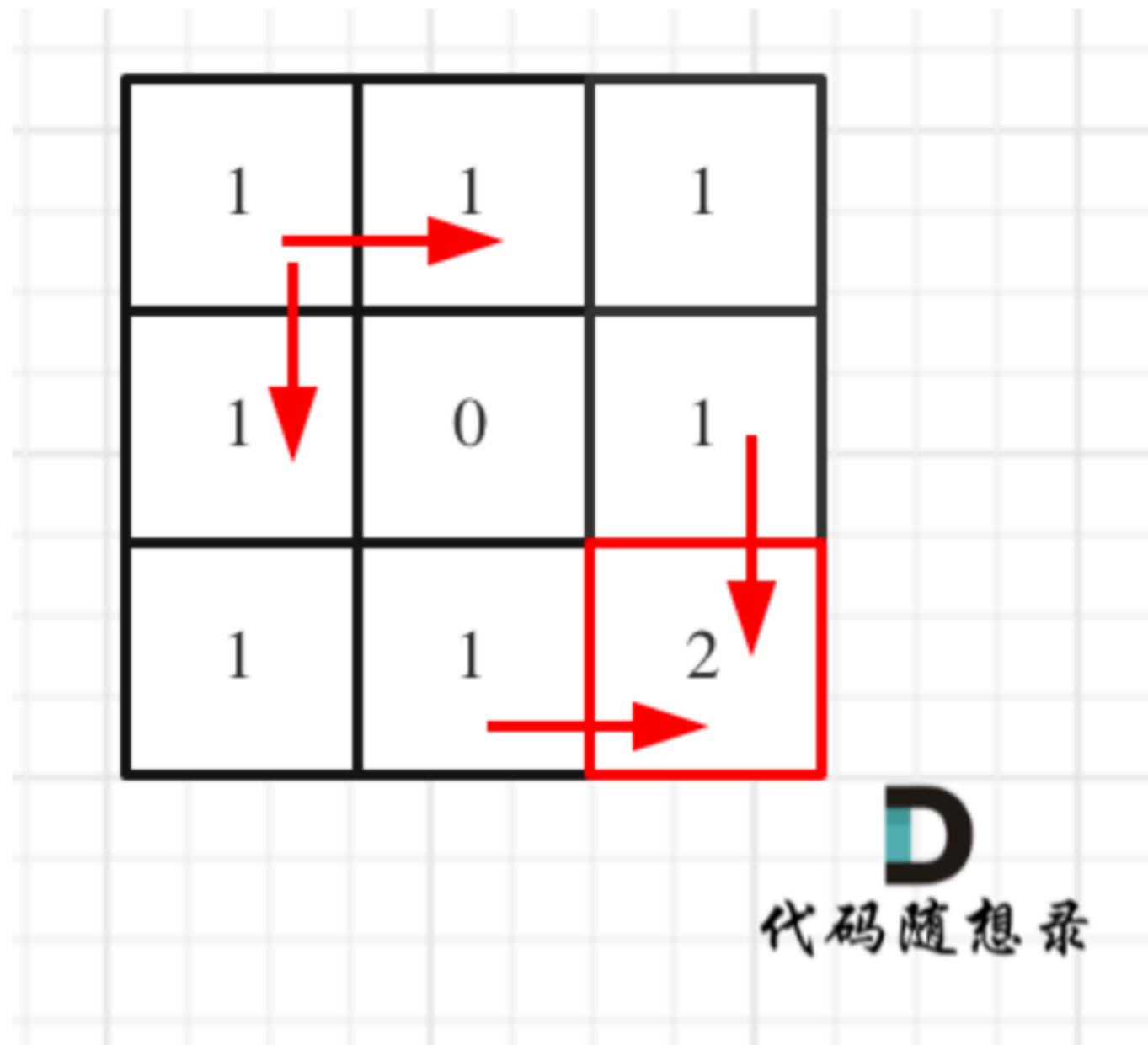
### 5. 举例推导dp数组

拿示例1来举例如题：





对应的dp table 如图：



```
In [34]: class Solution:
    def uniquePathsWithObstacles(self, obstacleGrid: List[List[int]]) -> int:
        m = len(obstacleGrid) # 行
        n = len(obstacleGrid[0]) # 列

        dp = [ [0 for _ in range(n)] for _ in range(m)] # 列里行外

        # Initialize
        for i in range(m):
            dp[i][0] = 1
            if obstacleGrid[i][0] == 1: # once encounter obstacle, 该列后面所有的都无法到达
                break
            for j in range(1, n):
                if obstacleGrid[i][j] == 1:
                    dp[i][j] = 0
                else:
                    dp[i][j] = dp[i][j-1] + dp[i-1][j]
```

```

dp[i][0] = 0
break

for j in range(n):
    dp[0][j] = 1
    if obstacleGrid[0][j] == 1: # once encounter obstacle, 该行后面所有的都无法到达
        dp[0][j] = 0
        break

# recurrence
# 本题先行后列和先列后行是一样的
for i in range(1, m):
    for j in range(1, n):
        if obstacleGrid[i][j] == 1:
            dp[i][j] = 0
        else:
            dp[i][j] = dp[i-1][j] + dp[i][j-1]

return dp[m-1][n-1]

# test
object = Solution()
obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]
print(object.uniquePathsWithObstacles(obstacleGrid))

```

2

## LC343 Integer break (medium)

### 343. Integer Break

已解答

中等 相关标签 相关企业 提示 文档

Given an integer  $n$ , break it into the sum of  $k$  positive integers, where  $k \geq 2$ , and maximize the product of those integers.

Return the maximum product you can get.

#### Example 1:

**Input:**  $n = 2$   
**Output:** 1  
**Explanation:**  $2 = 1 + 1$ ,  $1 \times 1 = 1$ .

#### Example 2:

**Input:**  $n = 10$   
**Output:** 36  
**Explanation:**  $10 = 3 + 3 + 4$ ,  $3 \times 3 \times 4 = 36$ .

#### Constraints:

- $2 \leq n \leq 58$

动态五部曲，分析如下：

1. 确定 dp 数组 (dp table) 以及下标的含义  
 $dp[i]$ : 分拆数字  $i$ , 可以得到的最大乘积为  $dp[i]$ 。  
 $dp[i]$  的定义将有助于理解整个解题过程, 下面哪一步想不懂了, 就想想  $dp[i]$  究竟表示的是什么意思!
2. 确定递推公式  
可以想  $dp[i]$  最大乘积是怎么得到的呢?  
假设对正整数  $i$  拆分出的第一个正整数是  $j$  ( $1 \leq j < i$ ) , 则有以下两种方案得到  $dp[i]$ :

- A. 将  $i$  拆分成  $j & i - j$  的和，且  $i - j$  不再拆分成多个正整数，此时的乘积是  $j * (i - j)$
  - B. 将  $i$  拆分成  $j & i - j$  的和，且  $i - j$  继续拆分成多个正整数，此时的乘积是  $j * dp[i - j]$
- 那么从1遍历到  $j$ ，遍历过程中每次都取  $(i - j) * j$  和  $dp[i - j] * j$  的最大值。

所以递推公式：

$$dp[i] = \max(dp[i], \max((i - j) * j, dp[i - j] * j))$$

- 那有同学问了，  $j$  怎么就不拆分呢？

$j$  是从1开始遍历，拆分  $j$  的情况，在遍历  $j$  的过程中其实都计算过了。

也可以这么理解， $(i - j)$  是单纯的把要拆分为两个数相乘，而  $dp[i - j] * j$  是拆分成两个以及两个以上的数相乘。如果是  $dp[i - j] * dp[i]$  也是默认将一个数强制拆成4份以及4份以上了。

- 那么在取最大值的时候，为什么还要比较  $dp[i]$  呢？

因为在递推公式推导的过程中，每次计算  $dp[i]$ ，取最大值而已。

### 3. dp 的初始化

有的题目里会给出  $dp[0] = 1, dp[1] = 1$  的初始化，但解释比较牵强，主要还是因为这么初始化何可以把题过了。

严格从  $dp$  的定义来说， $dp[0]$  和  $dp[1]$  都不应该初始化，也就是没有意义的数值。

拆分0和拆分1的最大乘积是多少呢？

这里要思考。

实际上只要知道  $dp[2] = 1$ ，从  $dp[i]$  的定义来说，拆分数字2，得到的最大乘积是1，这个没有任何异议！

`dp[2] = 1`

### 4. 确定遍历顺序

确定遍历顺序，先来看看递归公式： $dp[i] = \max(dp[i], \max((i - j) * j, dp[i - j] * j))$

$dp[i]$  是依靠  $dp[i - j]$  的状态，所以遍历  $i$  一定是从前向后遍历，先有  $dp[i - j]$  再有  $dp[i]$ 。

```
for i in range(3, n+1):
    for j in range(1, i):
        dp[i] = max(dp[i], j*(i-j), j*dp[i-j])
```

注意：

- 枚举  $j$  的时候，是从1开始的。从0开始的话，那么让拆分一个数拆个0，求最大乘积就没有意义了。
- $j$  的结束条件是  $j < i - 1$ ，其实  $j < i$  也是可以的，不过  $j < i - 1$  节省了一步：例如让  $j = i - 1$  的话，其实在  $j = 1$  的时候，这一步就已经拆出来了，重复计算，所以  $j < i - 1$ 。
- 至于  $i$  是从3开始，这样  $dp[i - j]$  就是  $dp[2]$  正好可以通过我们初始化的数值求出来。

### 5. 举例推导dp数组

$dp[8]$ : 若要拆分8，最大乘积是多少

举例当n为10 的时候，dp数组里的数值，如下：

$n = 10$									
下标i:	2	3	4	5	6	7	8	9	10
dp[i]:	1	2	4	6	9	12	18	27	36

本题  $n=10$ ，那么 i 从3开始， $i=3, 4, \dots, 10$  一个个写  
举例如对于  $i=8$ , 8 可以拆成

$$i = 8 = \begin{cases} 1 + 7 \\ 2 + 6 \\ 3 + 5 \\ 4 + 4 \end{cases}$$

j: 从1开始,  $j=1, 2, 3, 4$

没必要5, 因为  $(5+3)$  之前考虑过了.

$$1 \leq j \leq i//2 + 1$$

```
In [ ]: class Solution:
    def integerBreak(self, n: int) -> int:

        if n == 2:
            return 1

        # dp[i]: 拆分数字i, 得到的最大乘积是多少
        # 因为python的index是从0开始, 所以要想达到上面的定义, 需要dp长度为n+1
        dp = [0 for _ in range(n+1)]

        # Initialize
        dp[2] = 1

        # Recurrence
        for i in range(3, n+1):      # i是即将被拆分的数字
            for j in range(1, i // 2 + 1):  # j是拆分出来的第一个整数, j > 1

                # 计算切割点j和剩余部分(i-j)的乘积
                temp = max(j * (i - j), j * dp[i-j])

                # 并与之前的结果进行比较取较大值, 不断update dp[i]
                dp[i] = max(dp[i], temp)

        return dp[i]

# test
object = Solution()
print(object.integerBreak(10))
```

## Part 2 Knapsack problem

由于这部分太长，放到了另一个file里。

In [ ]:

## Part 3 打家劫舍问题

### LC198 House Robber (medium)

#### 198. House Robber

中等

相关标签

相关企业

文A

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given an integer array `nums` representing the amount of money of each house, return *the maximum amount of money you can rob tonight without alerting the police*.

#### Example 1:

**Input:** `nums = [1,2,3,1]`

**Output:** 4

**Explanation:** Rob house 1 (`money = 1`) and then rob house 3 (`money = 3`).

Total amount you can rob =  $1 + 3 = 4$ .

#### Example 2:

**Input:** `nums = [2,7,9,3,1]`

**Output:** 12

**Explanation:** Rob house 1 (`money = 2`), rob house 3 (`money = 9`) and rob house 5 (`money = 1`).

Total amount you can rob =  $2 + 9 + 1 = 12$ .

#### Constraints:

- `1 <= nums.length <= 100`
- `0 <= nums[i] <= 400`

#### 思路

大家如果刚接触这样的题目，会有点困惑，当前的状态我是偷还是不偷呢？

仔细一想，当前房屋偷与不偷取决于前一个房屋和前两个房屋是否被偷了。

所以这里就更感觉到，当前状态和前面状态会有一种依赖关系，那么这种依赖关系都是动规的递推公式。

<https://programmerncarl.com/0198.%E6%89%93%E5%AE%B6%E5%8A%AB%E8%88%8D.html#%E6%80%9D%E8%B7%AF>

动规五部曲分析如下：

1. 确定dp数组（dp table）以及下标的含义

$dp[i]$ : 考虑下标*i*（包括*i*）以内的房屋，最多可以偷窃的金额为 $dp[i]$ 。

2. 确定递推公式

决定 $dp[i]$ 的因素就是第*i*房间偷还是不偷。

- 如果偷第*i*房间，那么 $dp[i] = dp[i - 2] + nums[i]$ ，即：第*i - 1*房一定是不考虑的，找出下标*i - 2*（包括*i - 2*）以内的房屋，最多可以偷窃的金额为 $dp[i - 2]$ 加上第*i*房间偷到的钱。
- 如果不偷第*i*房间，那么 $dp[i] = dp[i - 1]$ ，即考虑*i - 1*房，（注意这里是考虑，并不是一定要偷*i - 1*房，这是很多同学容易混淆的点）

然后 $dp[i]$ 取最大值，即递推公式为：

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

### 3. dp数组如何初始化

从递推公式 $dp[i] = \max(dp[i - 2] + nums[i], dp[i - 1])$ 可以看出，递推公式的基础就是 $dp[0]$ 和 $dp[1]$

从 $dp[i]$ 的定义上来讲，

```
dp[0] = nums[0]
dp[1] = max(nums[0], nums[1])
```

### 4. 确定遍历顺序

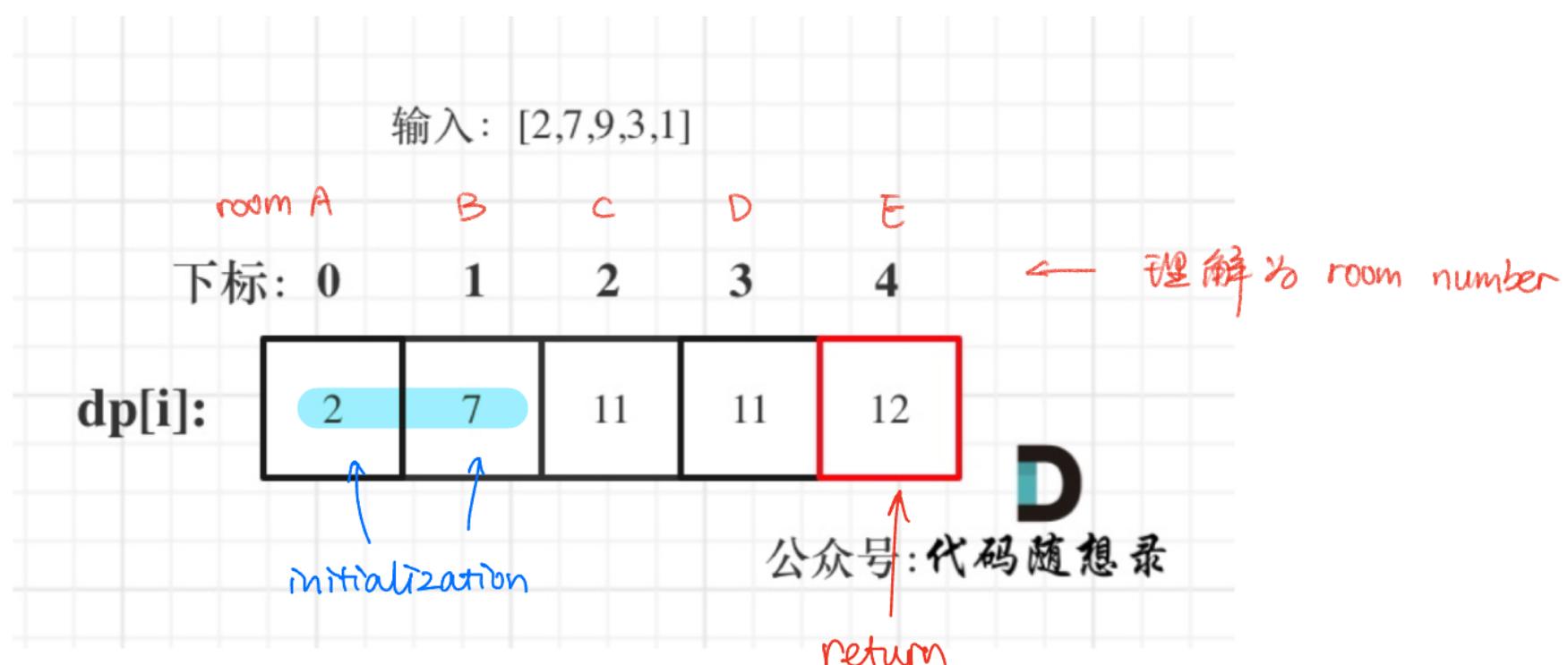
$dp[i]$ 是根据 $dp[i - 2]$ 和 $dp[i - 1]$ 推导出来的，那么一定是从前到后遍历！

代码如下：

```
for i in range(2, len(nums)):
    dp[i] = max(dp[i-2] + nums[i], dp[i-1])
```

### 5. 举例推导dp数组

以示例二，输入[2,7,9,3,1]为例。



## 版本一

```
In [39]: class Solution:
    def rob(self, nums: List[int]) -> int:
        N = len(nums)
        if N == 0: # 如果没有房屋，返回0
            return 0
        if N == 1: # 如果只有一个房屋，返回其金额
            return nums[0]

        # dp[i]: if we consider room number i, i-1, ..., 0, the max amount I can rob is dp[i]
        dp = [0 for _ in range(N)]

        # Initialize
        dp[0] = nums[0]
        dp[1] = max(nums[0], nums[1])

        # Recurrence
        for i in range(2, N):
            dp[i] = max(dp[i-2] + nums[i], dp[i-1])
```

```

for i in range(2, N):
    dp[i] = max(dp[i - 2] + nums[i], dp[i - 1])

return dp[-1]

# test
nums = [2, 7, 9, 3, 1]
object = Solution()
object.rob(nums)

```

Out[39]: 12

时间复杂度:  $O(N)$  where  $N = \text{len}(\text{nums})$ 空间复杂度:  $O(N)$ 

## 版本二 (优化版)

```

In [ ]: class Solution:
    def rob(self, nums: List[int]) -> int:
        if not nums: # 如果没有房屋, 返回0
            return 0

        prev_max = 0 # 上一个房屋的最大金额, 相当于dp[i-2]
        curr_max = 0 # 当前房屋的最大金额, 相当于dp[i-1]

        for num in nums:
            temp = curr_max # 临时变量保存当前房屋的最大金额
            curr_max = max(prev_max + num, curr_max) # 更新当前房屋的最大金额
            prev_max = temp # 更新上一个房屋的最大金额

        return curr_max # 返回最后一个房屋中可抢劫的最大金额

```

时间复杂度:  $O(N)$  where  $N = \text{len}(\text{nums})$ 空间复杂度:  $O(1)$ 

下面是两个版本代码的对应关系, 帮助理解它们的相似之处和不同之处:

### 1. 变量/数组的对应关系

- 版本1: 使用 `dp` 数组, `dp[i]` 表示到达第 `i` 个房屋时, 能够抢到的最大金额。
- 版本2: 使用 `prev_max` 和 `curr_max` 两个变量, 分别表示到达前一个和当前房屋时的最大抢劫金额。这两个变量在逻辑上分别对应 `dp[i-2]` 和 `dp[i-1]`。

### 2. 循环和递推关系

两种方法都在循环中使用递推公式来更新最大金额:

#### • 版本1:

- `dp[i] = max(dp[i - 2] + nums[i], dp[i - 1])`
- 这个公式表示, 如果抢劫第 `i` 个房屋, 最大金额就是 `dp[i-2] + nums[i]` 和 `dp[i-1]` 之间的较大值。

#### • 版本2:

- `curr_max = max(prev_max + num, curr_max)`
- 这里的 `prev_max` 相当于 `dp[i-2]`, `curr_max` 相当于 `dp[i-1]`。因此, 这一公式和版本1的公式效果相同, 但不需要存储整个 `dp` 数组。

### 3. 空间复杂度

- 版本1: 空间复杂度为  $O(N)$ , 因为使用了 `dp` 数组存储每个房屋的最大抢劫金额。
- 版本2: 空间复杂度为  $O(1)$ , 只需要几个变量, 不需要额外的数组。

### 总结对照

功能	版本1 (使用 <code>dp</code> 数组)	版本2 (使用 <code>prev_max</code> 和 <code>curr_max</code> )
初始化	<code>dp[0] = nums[0], dp[1] = max(nums[0], nums[1])</code>	<code>prev_max = 0, curr_max = 0</code>
递推关系	<code>dp[i] = max(dp[i - 2] + nums[i], dp[i - 1])</code>	<code>curr_max = max(prev_max + num, curr_max)</code>
空间复杂度	$O(N)$	$O(1)$
最终返回的最大金额	<code>dp[-1]</code>	<code>curr_max</code>

## LC213 House Robber II (medium)

# 213. House Robber II

[中等](#)[相关标签](#)[相关企业](#)[提示](#)[贡献](#)

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are **arranged in a circle**. That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have a security system connected, and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array `nums` representing the amount of money of each house, return *the maximum amount of money you can rob tonight without alerting the police*.

#### Example 1:

**Input:** nums = [2,3,2]

**Output:** 3

**Explanation:** You cannot rob house 1 (money = 2) and then rob house 3 (money = 2), because they are adjacent houses.

#### Example 2:

**Input:** nums = [1,2,3,1]

**Output:** 4

**Explanation:** Rob house 1 (money = 1) and then rob house 3 (money = 3). Total amount you can rob = 1 + 3 = 4.

#### Example 3:

**Input:** nums = [1,2,3]

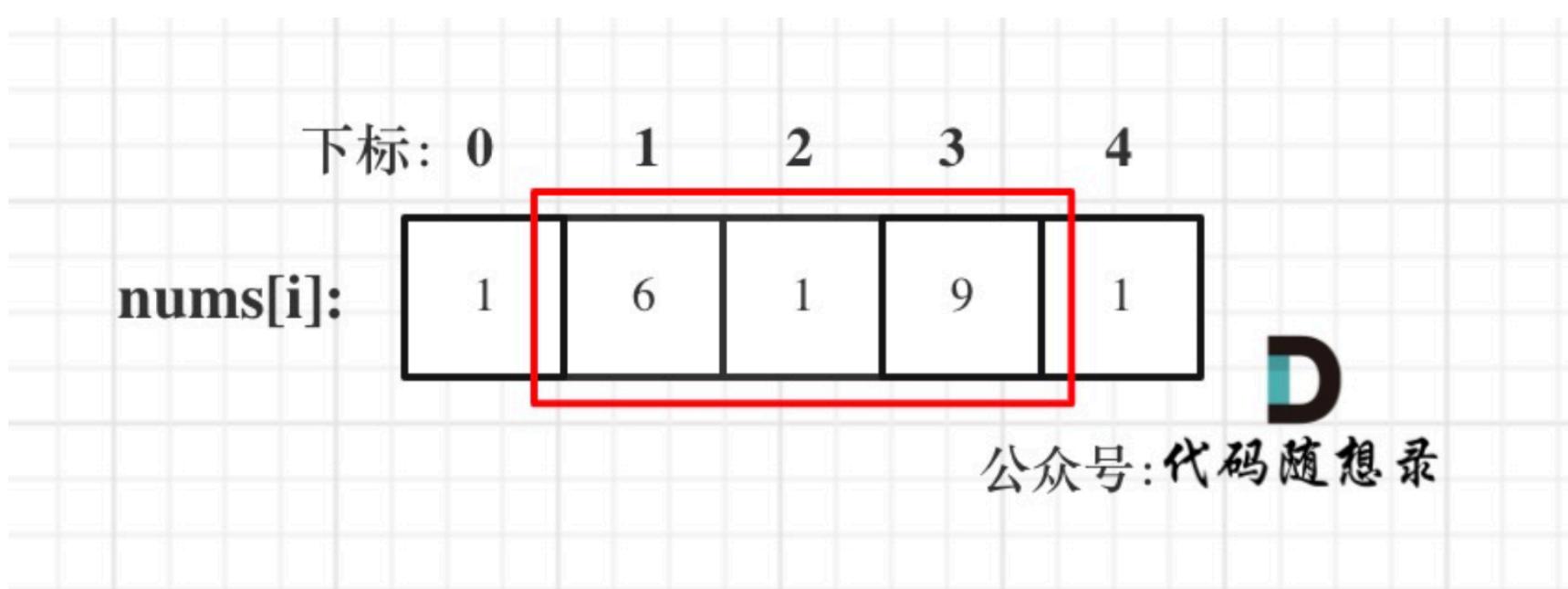
**Output:** 3

#### 思路

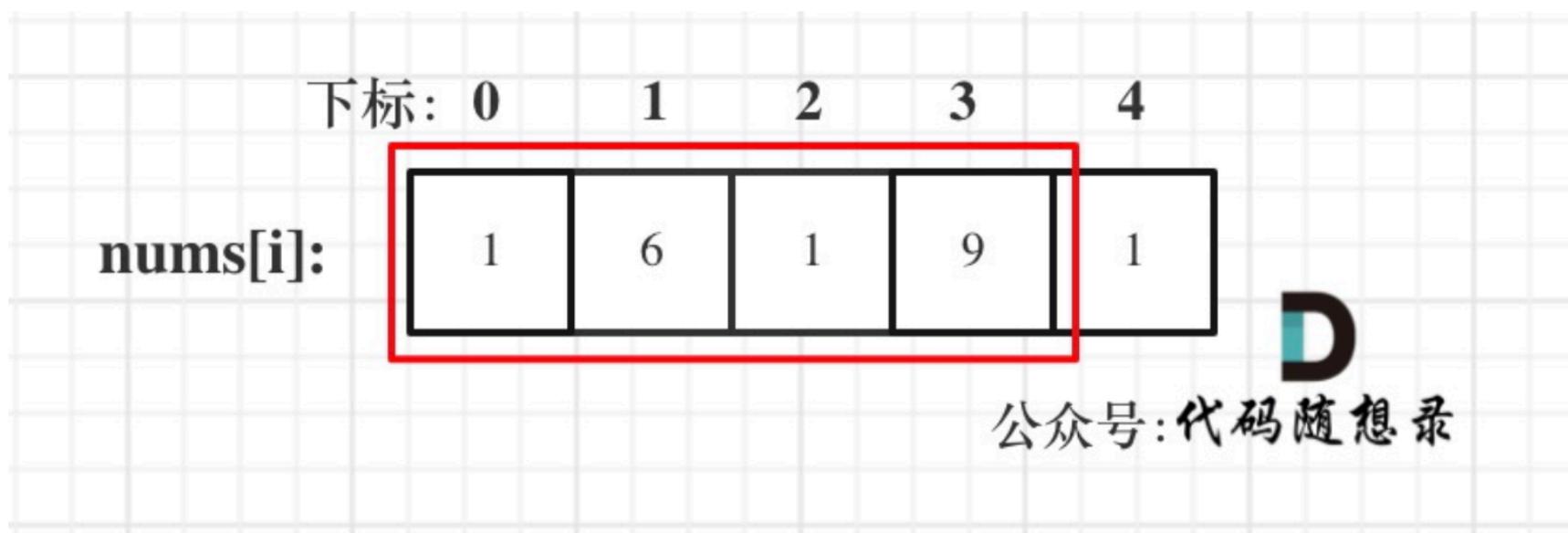
这道题目和198.打家劫舍 是差不多的，唯一区别就是成环了。

对于`nums`数组，考虑如下三种情况：

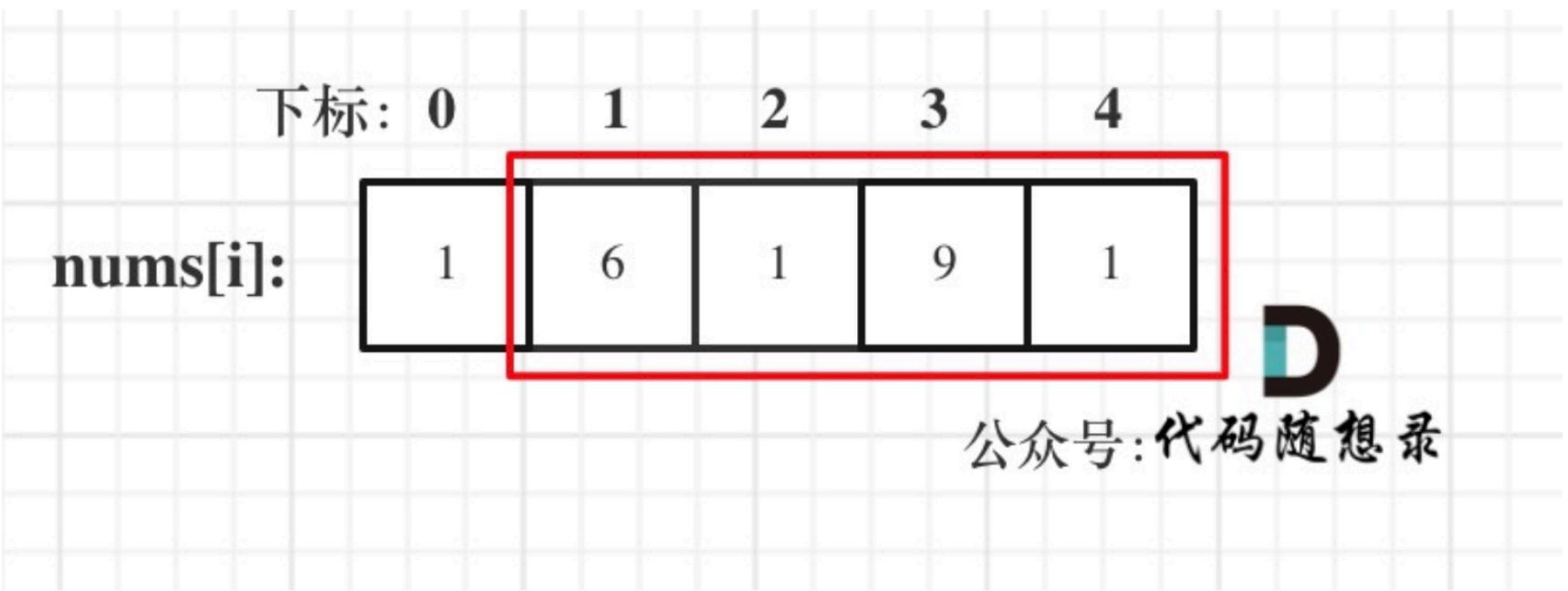
- 情况一：考虑不包含首尾元素



- 情况二：考虑包含首元素，不包含尾元素



- 情况三：考虑包含尾元素，不包含首元素



注意我这里用的是"考虑"，例如情况三，虽然是考虑包含尾元素，但不一定要选尾部元素！对于情况三，取 $\text{nums}[1]$  和  $\text{nums}[3]$ 就是最大的。

而情况二 和 情况三 都包含了情况一了，所以只考虑情况二和情况三就可以了。

分析到这里，本题其实比较简单了。剩下的和[198.打家劫舍](#)就是一样的了。

版本一：

(我自己写的解答，自己看着最熟悉)

```
In [ ]: class Solution:
    def rob(self, nums: List[int]) -> int:
        N = len(nums)
        # edge cases
        if N == 0: return 0
        if N == 1: return nums[0]

        # Take the maximum of two cases
        nums_case1 = nums[:-1]
        res_case1 = self.houseRobberI(nums_case1) # 情况二：不抢最后一个房子

        nums_case2 = nums[1:]
        res_case2 = self.houseRobberI(nums_case2) # 情况三：不抢第一个房子

        return max(res_case1, res_case2)

    # 198. 打家劫舍的逻辑
    def houseRobberI(self, nums: List[int]) -> int:
        # Edge cases
        N = len(nums)
        if N == 0: return 0
        if N == 1: return nums[0]

        dp = [0 for _ in range(N)]

        # Initialize
        dp[0] = nums[0]
        dp[1] = max(nums[0], nums[1])

        # Recurrence
        for i in range(2, N):
            dp[i] = max(dp[i-2] + nums[i], dp[i-1])

        return dp[-1]

    # test
    nums = [2,3,2]
    object = Solution()
    object.rob(nums)
```

Out[ ]: 3

时间复杂度:  $O(N)$  where  $N = \text{len}(\text{nums})$

空间复杂度:  $O(N)$

## 版本二：（随想录）

思路跟我的完全一样，只是表达方式不同

```
In [ ]: class Solution:
    def rob(self, nums: List[int]) -> int:
        if len(nums) == 0:
            return 0
        if len(nums) == 1:
            return nums[0]

        result1 = self.robRange(nums, 0, len(nums) - 2) # 情况二
        result2 = self.robRange(nums, 1, len(nums) - 1) # 情况三
        return max(result1, result2)

    # 198. 打家劫舍的逻辑
    def robRange(self, nums: List[int], start: int, end: int) -> int:
        if end == start:
            return nums[start]

        prev_max = nums[start]
        curr_max = max(nums[start], nums[start + 1])

        for i in range(start + 2, end + 1):
            temp = curr_max
            curr_max = max(prev_max + nums[i], curr_max)
            prev_max = temp

        return curr_max
```

## 版本三：（优化版）

```
In [42]: class Solution:
    def rob(self, nums: List[int]) -> int:
        if not nums: # 如果没有房屋, 返回0
            return 0

        if len(nums) == 1: # 如果只有一个房屋, 返回该房屋的金额
            return nums[0]

        # 情况二: 不抢劫第一个房屋
        prev_max = 0 # 上一个房屋的最大金额
        curr_max = 0 # 当前房屋的最大金额
        for num in nums[1:]:
            temp = curr_max # 临时变量保存当前房屋的最大金额
            curr_max = max(prev_max + num, curr_max) # 更新当前房屋的最大金额
            prev_max = temp # 更新上一个房屋的最大金额
        result1 = curr_max

        # 情况三: 不抢劫最后一个房屋
        prev_max = 0 # 上一个房屋的最大金额
        curr_max = 0 # 当前房屋的最大金额
        for num in nums[:-1]:
            temp = curr_max # 临时变量保存当前房屋的最大金额
            curr_max = max(prev_max + num, curr_max) # 更新当前房屋的最大金额
            prev_max = temp # 更新上一个房屋的最大金额
        result2 = curr_max

        return max(result1, result2)
```