

# 极客时间算法训练营

## 第十二课

### 动态规划（一）

李煜东

《算法竞赛进阶指南》作者



# 目录

1. 动态规划总论：状态设计的要点和技巧
2. 简单的线性动态规划

# 再探：零钱兑换问题

零钱兑换

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

给一个硬币面额的可选集合 coins，求拼成金额 amount 最少需要多少枚硬币。

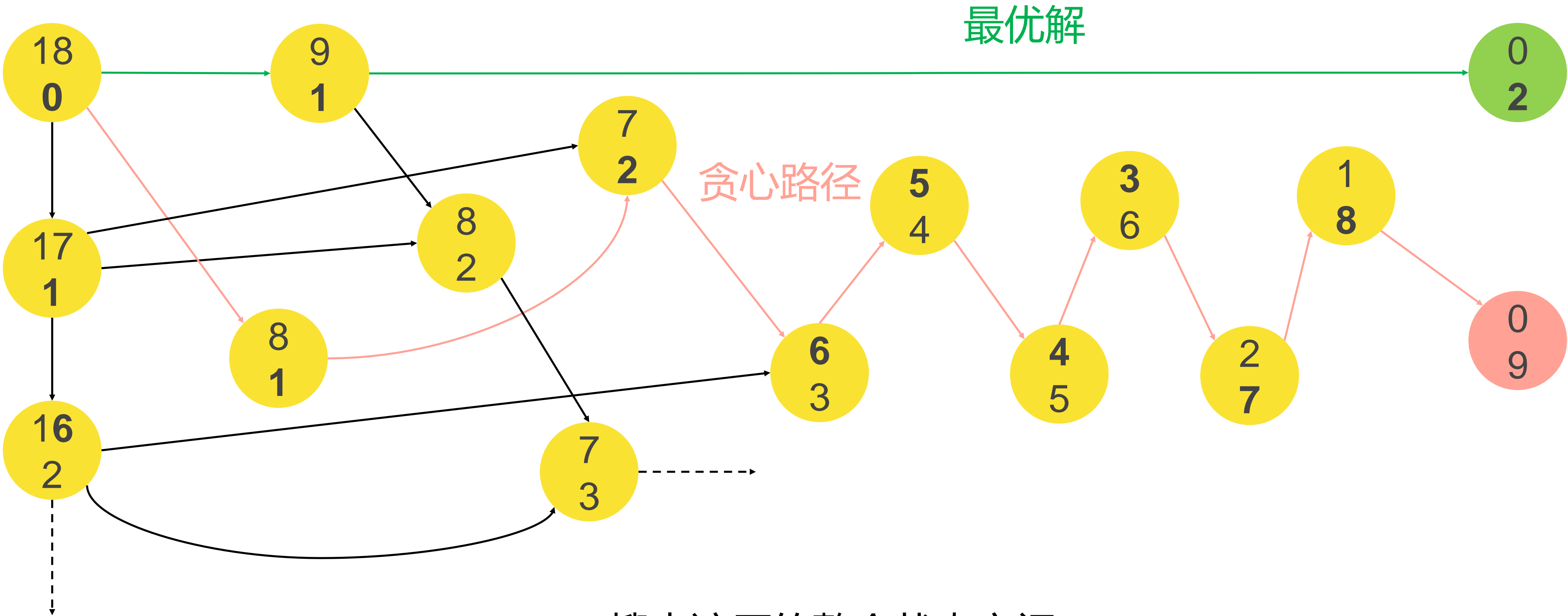
例：coins = [20, 10, 5, 1], amount = 46

答案：46 = 20 + 20 + 5 + 1



# 零钱兑换：搜索

状态：剩余金额、已用硬币枚数



搜索遍历的整个状态空间

# 零钱兑换：最优子结构

状态中没有必要包含“已用硬币枚数”

对于每个“剩余金额”，搜一次，求出“兑换这个金额所需的**最少**硬币枚数”就足够了

原始状态：剩余金额、已用硬币枚数，目标：到达终点（0元）

新状态：剩余金额，**最优化**目标：硬币枚数最少

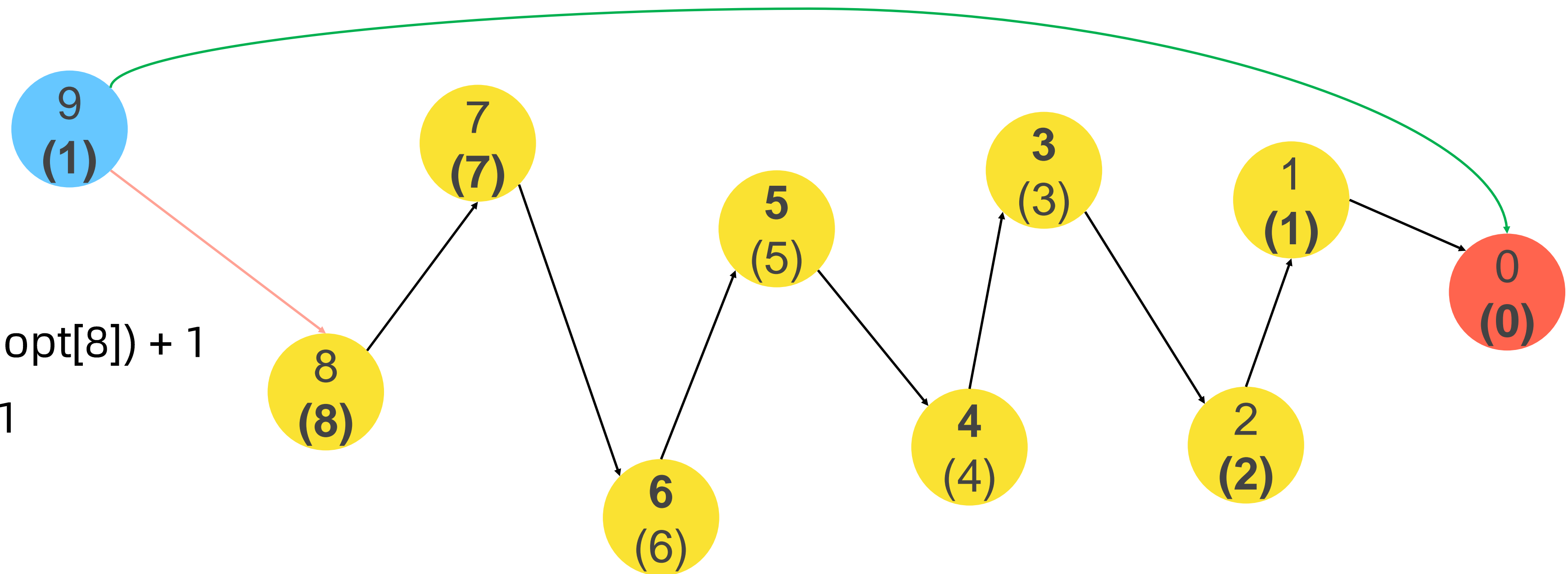
推导关系：“兑换 n 元的最少硬币枚数”  $\text{opt}(n) = \min(\text{opt}(n - 1), \text{opt}(n - 9), \text{opt}(n - 10)) + 1$

状态 + 最优化目标 + 最优化目标之间具有递推关系 = **最优子结构**

# 零钱兑换：最优子结构

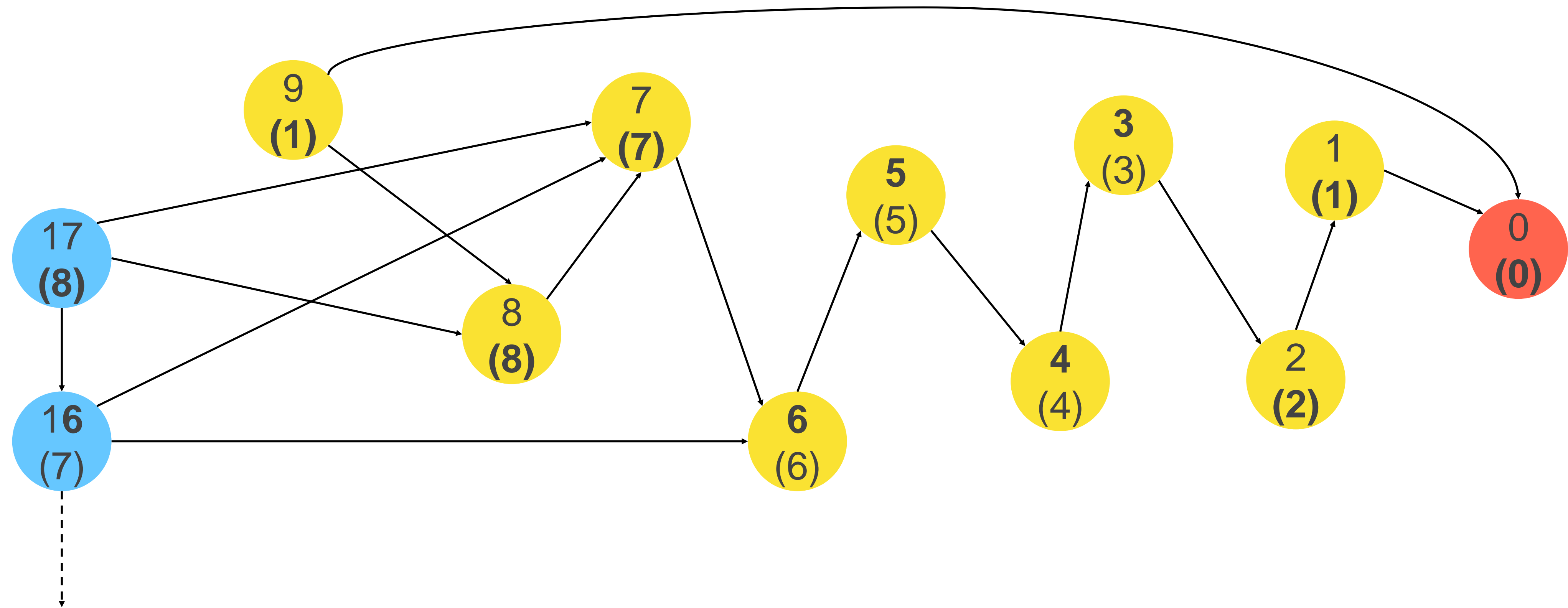
状态：剩余金额（括号中表示凑成这个金额的最少硬币数）

$\text{opt}[9]$   
 $= \min(\text{opt}[0], \text{opt}[8]) + 1$   
 $= \min(0, 8) + 1$   
 $= 1$



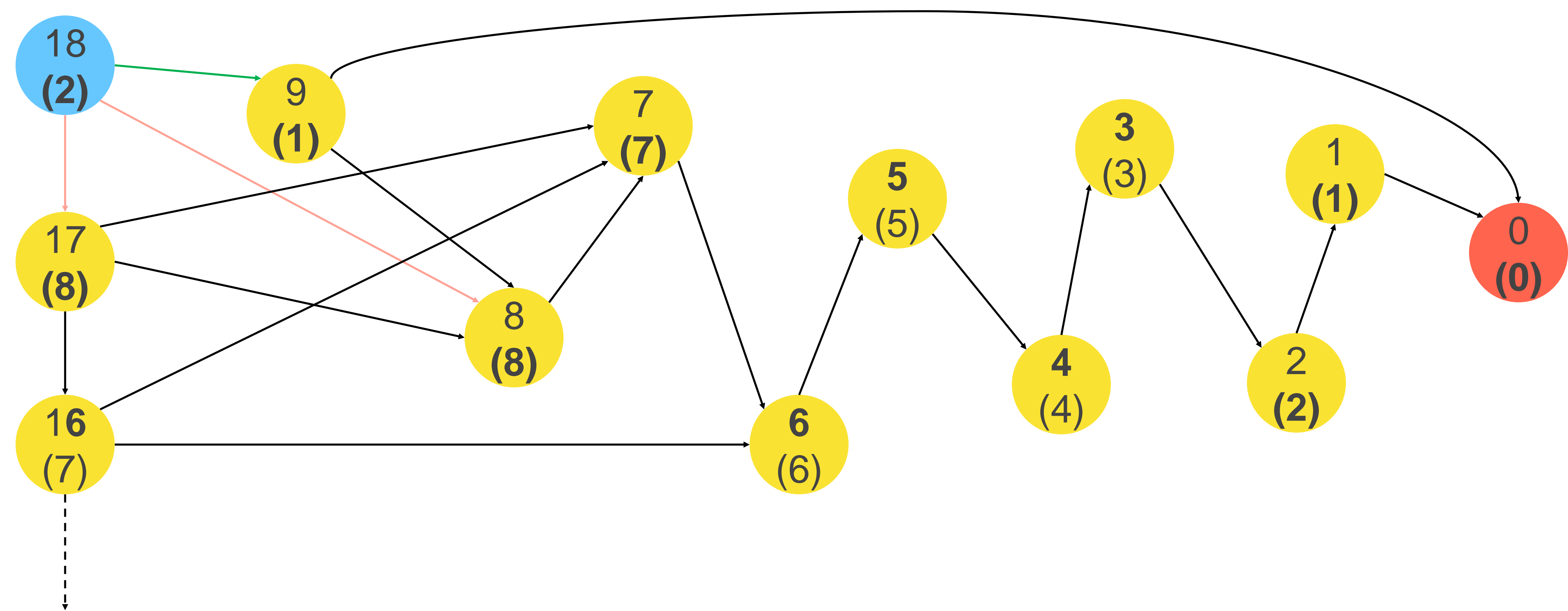
# 零钱兑换：最优子结构

状态：剩余金额（括号中表示凑成这个金额的最少硬币数）



# 零钱兑换：最优子结构

状态：剩余金额（括号中表示凑成这个金额的最少硬币数）





# 零钱兑换：递推实现

```
class Solution {
    public int coinChange(int[] coins, int amount) {
        int INF = (int)1e9;
        int[] opt = new int[amount + 1];
        opt[0] = 0;
        for (int i = 1; i <= amount; i++) {
            opt[i] = INF;
            for (int j = 0; j < coins.length; j++)
                if (i - coins[j] >= 0)
                    opt[i] = Math.min(opt[i], opt[i - coins[j]] + 1);
        }
        if (opt[amount] >= INF) opt[amount] = -1;
        return opt[amount];
    }
}
```

# 零钱兑换：记忆化搜索

```
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        self.coins = coins
        self.opt = list([-1]) * (amount + 1)
        ans = self.calc(amount)
        return ans if ans < 1e9 else -1

    def calc(self, amount):
        if amount == 0:
            return 0
        if amount < 0:
            return 1e9
        if self.opt[amount] != -1:
            return self.opt[amount]
        self.opt[amount] = 1e9
        for coin in self.coins:
            self.opt[amount] = min(self.opt[amount], self.calc(amount - coin) + 1)
        return self.opt[amount]
```

无论是递推实现还是记忆化搜索（递归实现）  
这种定义状态 + 最优子结构 + 推导关系的解题方法  
其实就是 **动态规划** 算法

# 动态规划

**动态规划 (DP, dynamic programming)** 是一种对问题的状态空间进行分阶段、有顺序、不重复、决策性遍历的算法

动态规划的关键与前提：

重叠子问题 —— 与递归、分治等一样，要具有同类子问题，用若干维状态表示

最优子结构 —— 状态对应着一个最优化目标，并且最优化目标之间具有推导关系

无后效性 —— 问题的状态空间是一张有向无环图（可按一定的顺序遍历求解）

动态规划一般采用递推的方式实现

也可以写成递归或搜索的形式，因为每个状态只遍历一次，也被称为**记忆化搜索**

# 动态规划

动态规划三要素：阶段、状态、决策

```
opt[0] = 0;
for (int i = 1; i <= amount; i++) {
    opt[i] = INF;
    for (int j = 0; j < coins.length; j++)
        if (i - coins[j] >= 0)
            opt[i] = Math.min(opt[i], opt[i - coins[j]] + 1);
}
```

阶段（线性增长）

决策（找到子问题）

重叠子问题

状态（具有最优子结构）



# 动态规划

一道动态规划题目的标准题解：

设  $opt[i]$  表示凑成金额  $i$  所需的最少硬币数

状态转移方程

$$opt[i] = \min_{coin \in coins} \{ opt[i - coin] \} + 1$$

边界

$$opt[0] = 0, \quad opt[i] = +\infty \ (i > 0)$$

目标

$$opt[amount]$$

时间复杂度：  $O(amount * |coins|)$

一道动态规划题目，写出状态转移方程，就已经完成了大半的工作  
剩下的任务就是照着方程写几个循环递推实现就行了

# 实战：路径计数

不同路径 II

<https://leetcode-cn.com/problems/unique-paths-ii/>

一个机器人位于一个  $m \times n$  网格的左上角

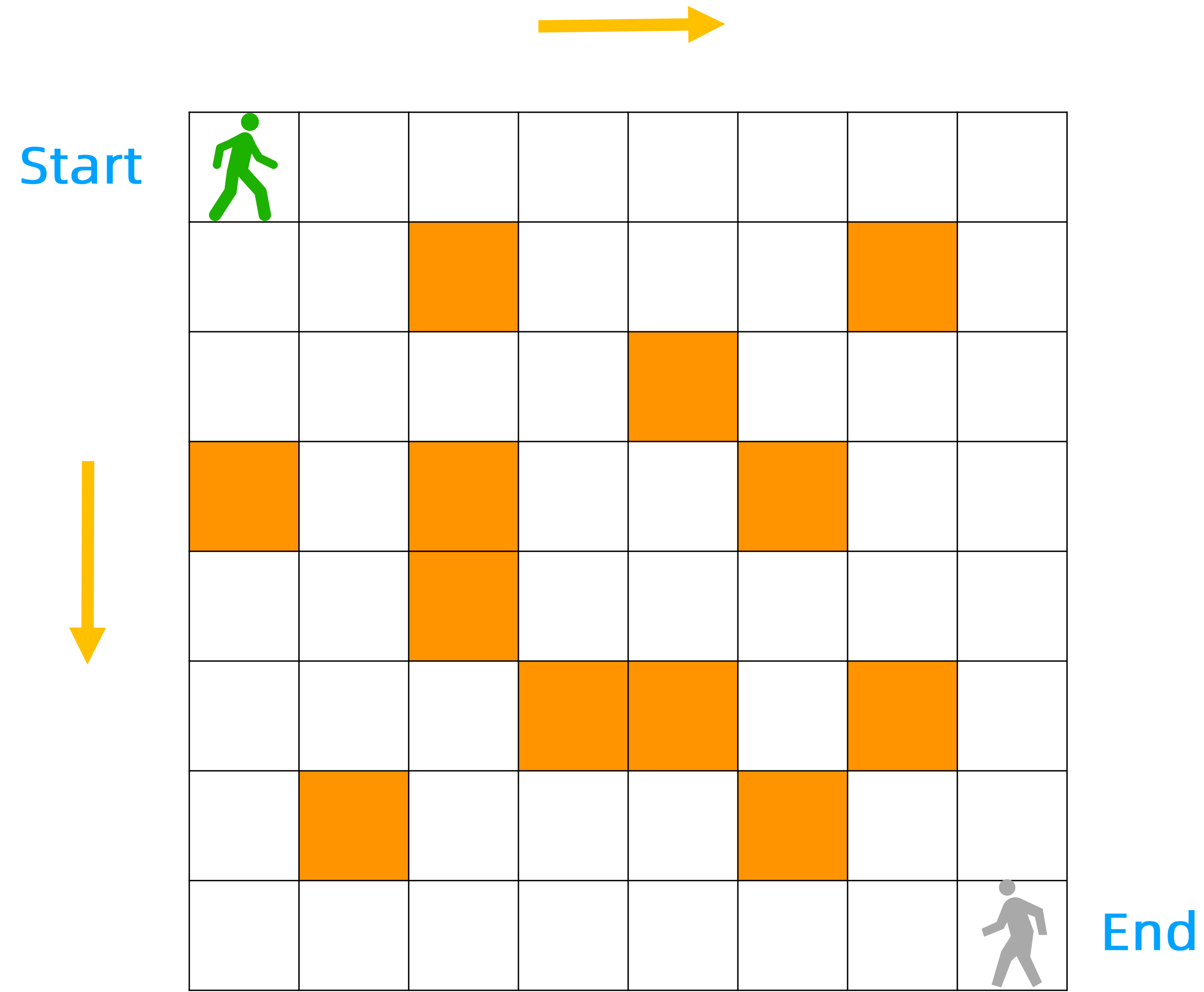
机器人每次只能向下或者向右移动一步

机器人试图达到网格的右下角

现在考虑网格中有障碍物

那么从左上角到右下角将会有多少条不同的路径？

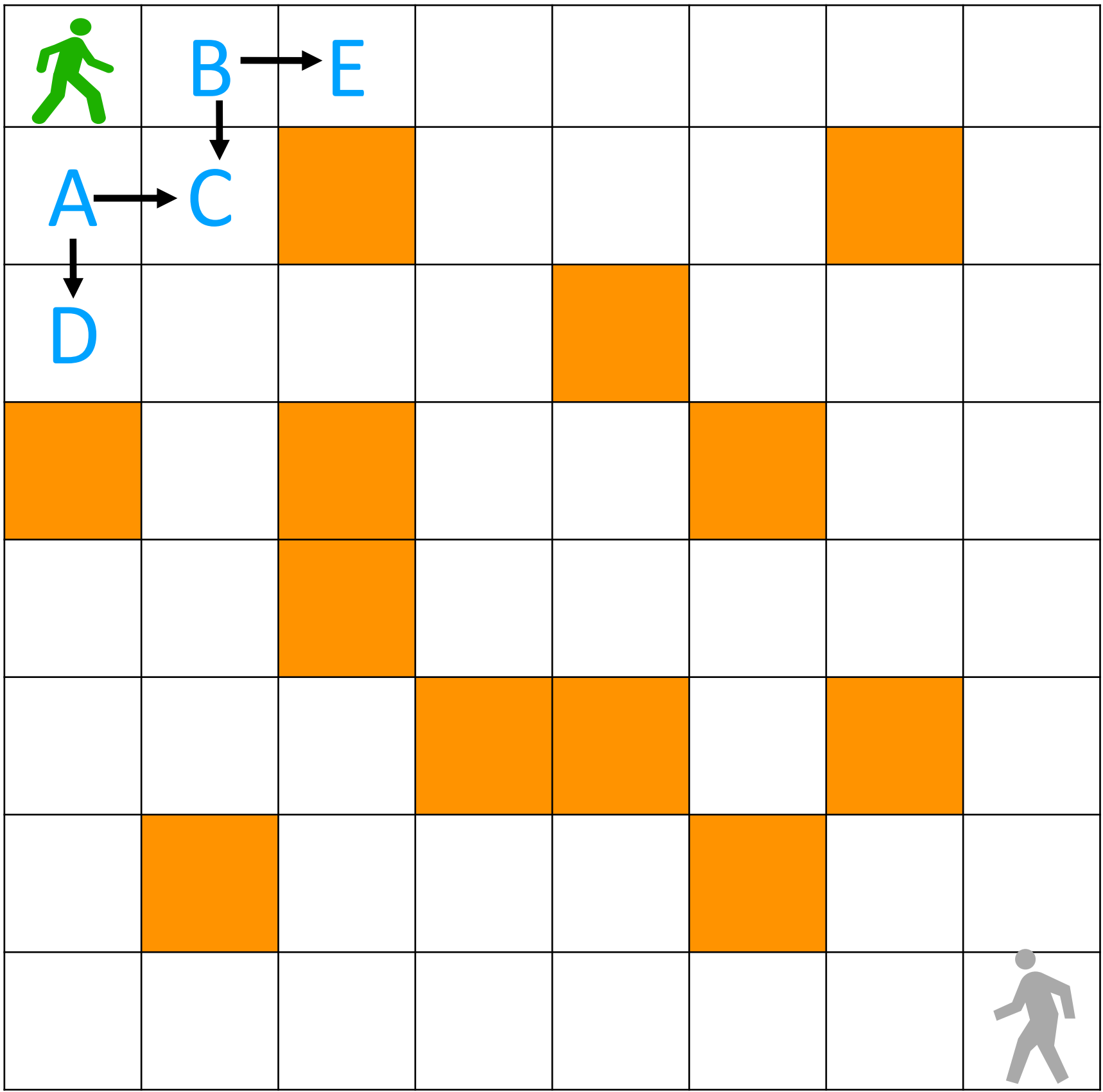
# 路径计数



# 路径计数



Start



End



paths (start) =

paths (A)

+

paths (B)

||

||

paths (D) +  
paths (C)

paths (C) +  
paths (E)

# 路径计数

Bottom-up

$f[i, j]$  表示从  $(i, j)$  到 End 的路径数

如果  $(i, j)$  是空地,  $f[i, j] = f[i + 1, j] + f[i, j + 1]$

否则  $f[i, j] = 0$

Top-down

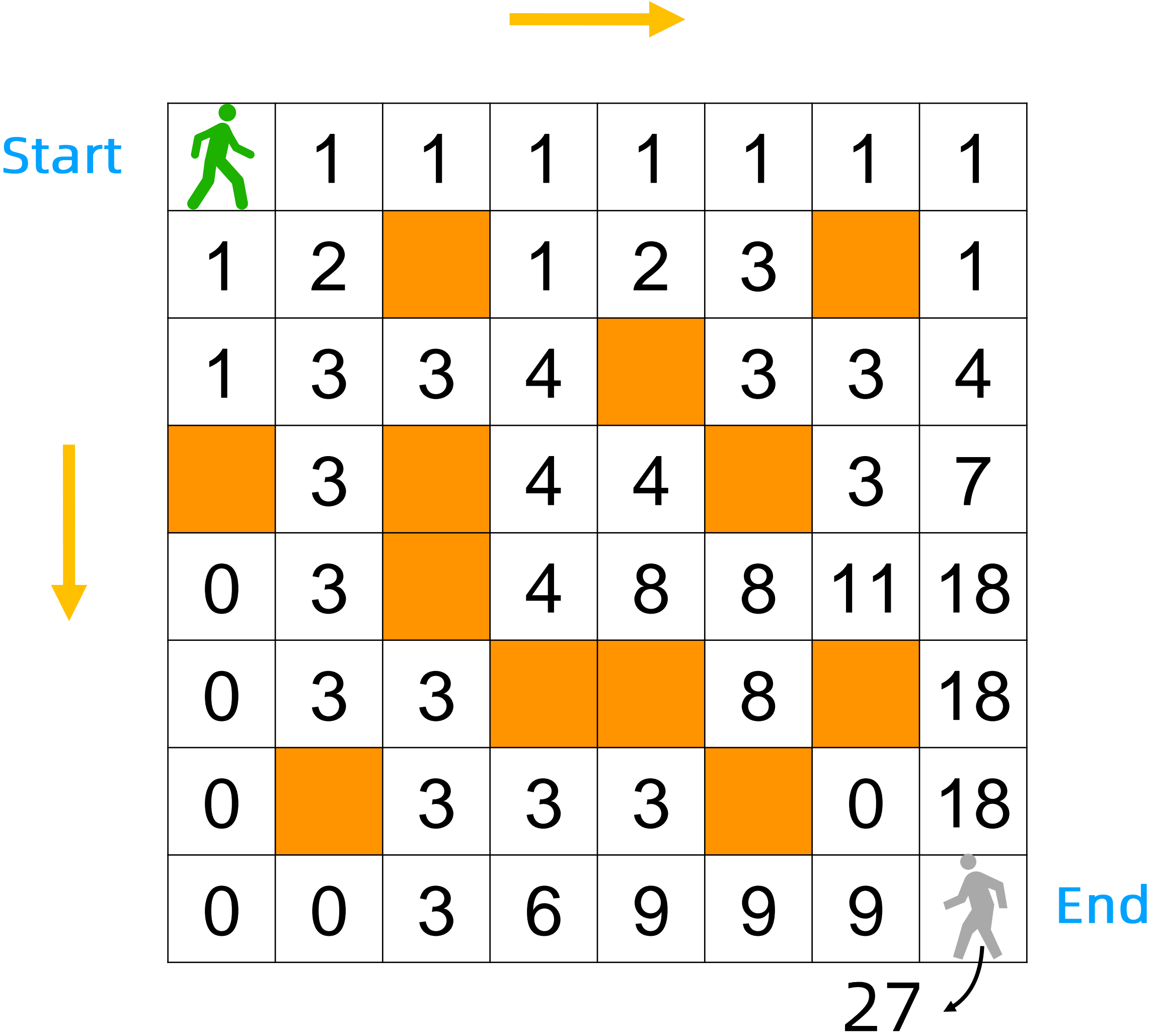
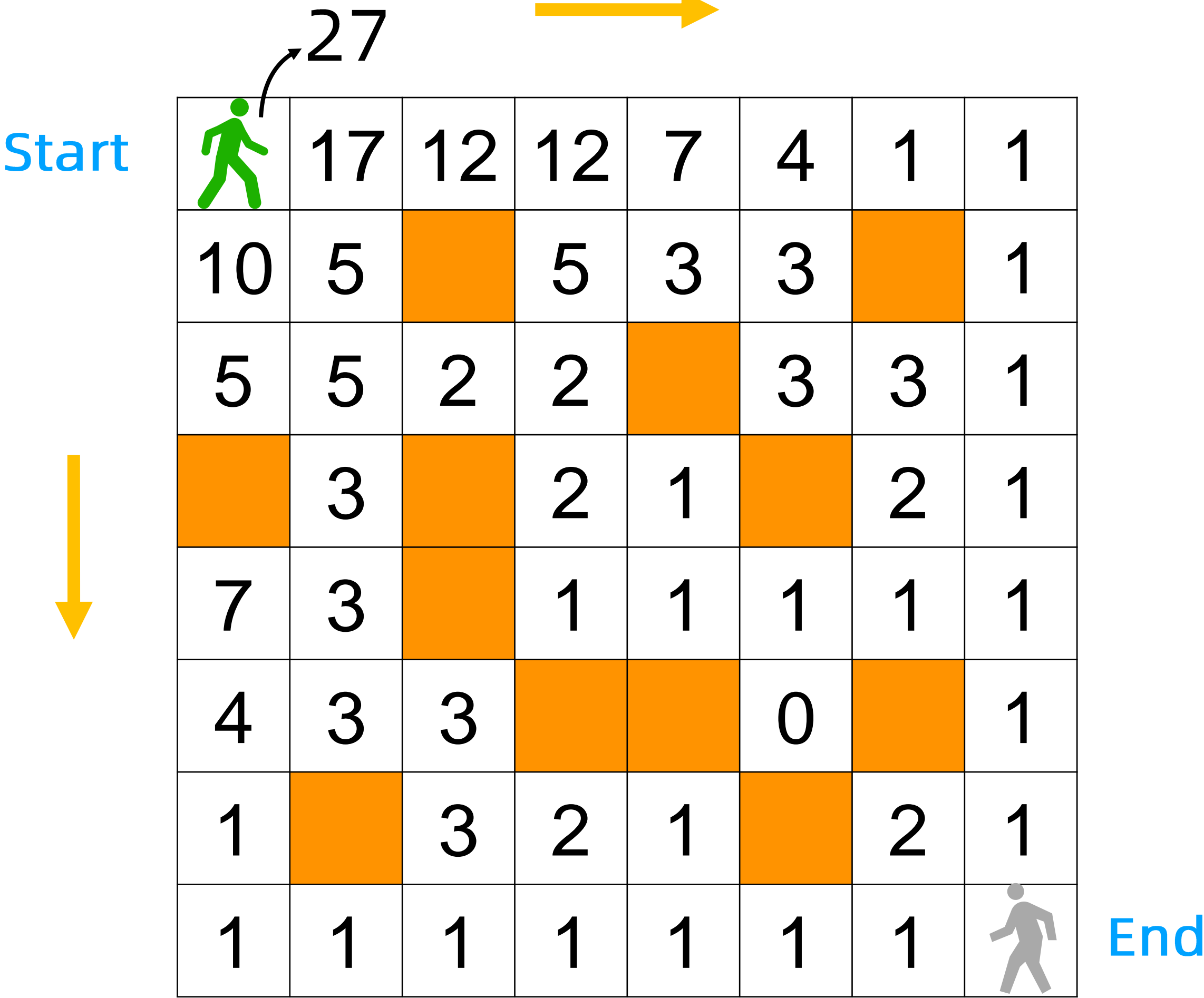
$f[i, j]$  表示从 Start 到  $(i, j)$  的路径数

如果  $(i, j)$  是空地,  $f[i, j] = f[i - 1, j] + f[i, j - 1]$

否则  $f[i, j] = 0$



# 路径计数



# 路径计数

Bottom-up 记忆化搜索（递归、分治思想）

```
int countPaths(boolean[][]grid, int row, int col) {  
    if (!validSquare(grid, row, col)) return 0;  
    if (isAtEnd(grid, row, col)) return 1;  
    return countPaths(grid, row + 1, col) + countPaths(grid, row, col + 1);  
}
```

# 路径计数

```
int n = grid.length, m = grid[0].length;
int[][] f = new int[n][m];
for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++) {
        if (grid[i][j] == 1) f[i][j] = 0;
        else if (i == 0 && j == 0) f[i][j] = 1;
        else if (i == 0) f[i][j] = f[i][j - 1];
        else if (j == 0) f[i][j] = f[i - 1][j];
        else f[i][j] = f[i - 1][j] + f[i][j - 1];
    }
return f[n - 1][m - 1];
```

# 实战：最长公共子序列

<https://leetcode-cn.com/problems/longest-common-subsequence/>

给两个字符串，求最长公共子序列（LCS），例如：

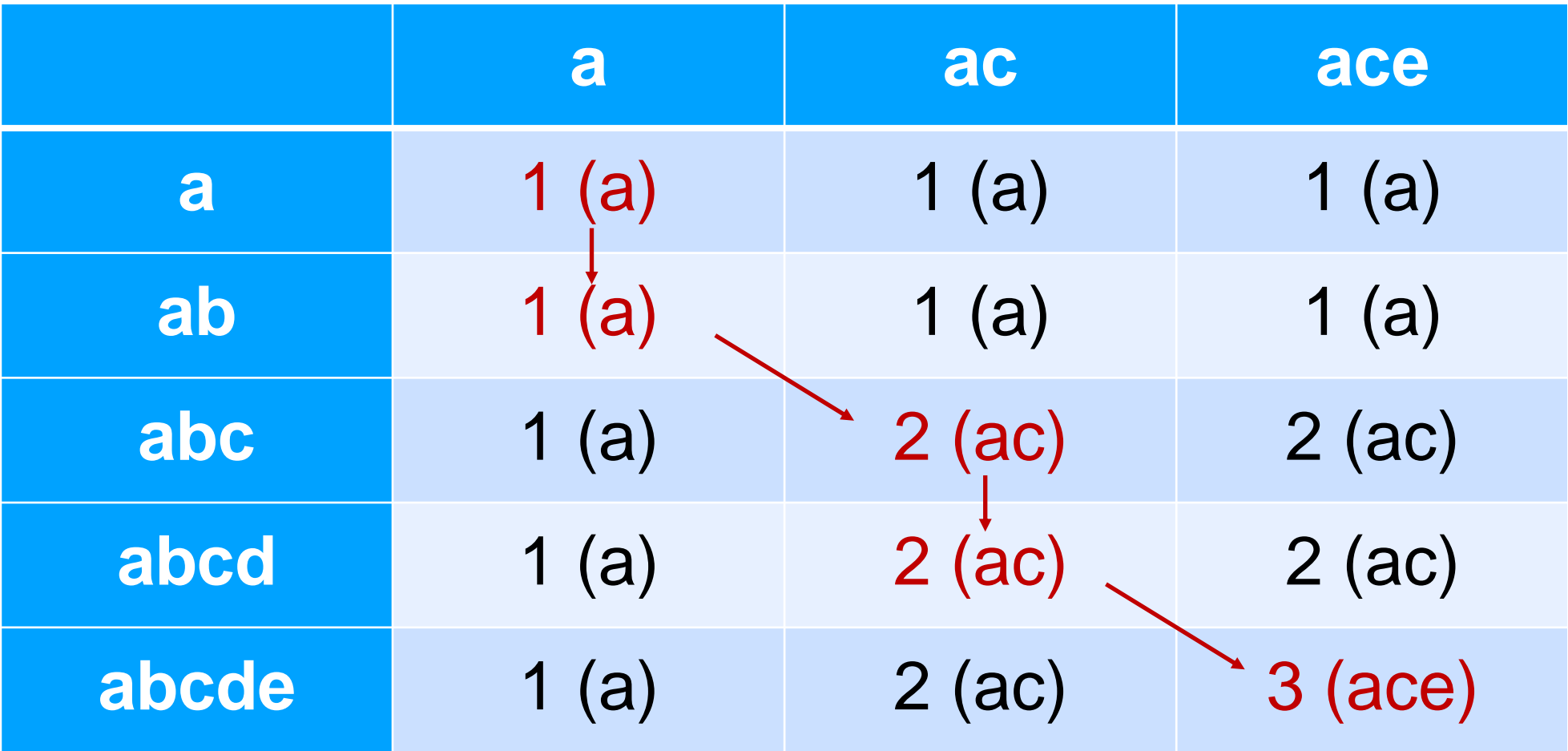
text1 = "abcde"

text2 = "ace"

入手DP问题第一步：人工模拟的话怎么算？

拿一个小例子，画个表

	a	ac	ace
a	1 (a)	1 (a)	1 (a)
ab	1 (a)	1 (a)	1 (a)
abc	1 (a)	2 (ac)	2 (ac)
abcd	1 (a)	2 (ac)	2 (ac)
abcde	1 (a)	2 (ac)	3 (ace)



# 最长公共子序列 (LCS)

确定“状态”的原则：寻找变化信息

- `abcd, ac (ac) => abcde, ace (ace)`
- 两个子串的长度是变化的，内容是固定的

确定“最优子结构”的原则：寻找代表

- 同样的两个子串，能组成很多公共子序列
- 只关心最大长度，不关心具体长什么样

确定“阶段”的原则：线性增长的轮廓

- “轮廓”是已计算区域与未计算区域的分界

确定“决策”的原则：人工模拟时考虑了哪些选项？

	a	ac	ace
a	1 (a)	1 (a)	1 (a)
ab	1 (a)	1 (a)	1 (a)
abc	1 (a)	2 (ac)	2 (ac)
abcd	1 (a)	2 (ac)	2 (ac)
abcde	1 (a)	2 (ac)	3 (ace)



# 最长公共子序列 (LCS)

$f[i, j]$  表示 text1 的前  $i$  个字符和 text2 的前  $j$  个字符能组成的 LCS 的长度

如果  $\text{text1}[i] = \text{text2}[j]$ :  $f[i, j] = f[i - 1, j - 1] + 1$

如果  $\text{text1}[i] \neq \text{text2}[j]$ :  $f[i, j] = \max(f[i - 1, j], f[i, j - 1])$

动规题目的边界处理技巧

方法一:

$f[0, 0] = 0$ , 然后递推时用 **if 语句判断**, 目标  $f[n - 1, m - 1]$

方法二:

认为字符串**下标从1开始**,  $f[i, 0] = 0$  与  $f[0, j] = 0$  均作为边界, 目标  $f[n, m]$

# 实战：最长上升子序列（LIS）

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

例：0,3,1,6,2,7

选 0?	0						
选 3?	3	0,3					
选 1?	1	0,1					
选 6?	6	0,6	3,6	1,6	0,3,6	0,1,6	
选 2?	2	0,2	1,2	0,1,2			
选 7?	7	0,7	3,7	.....	0,3,6,7	0,1,6,7	0,1,2,7

# 实战：最长上升子序列（LIS）

同样是以 6 为结尾，我们真的需要  $[6]$   $[0,6]$   $[3,6]$   $[1,6]$  吗？

对于  $[0,3,6]$  和  $[0,1,6]$ ，6 前面是什么数对后续的选择有影响吗？

我们只关心：选了几个数，结尾的数是哪个



以  $a[0] = 0$  为结尾的 LIS 长度是 1

以  $a[1] = 3$  为结尾的 LIS 长度是 2

以  $a[2] = 1$  为结尾的 LIS 长度是 2

以  $a[3] = 6$  为结尾的 LIS 长度是 3

以  $a[4] = 2$  为结尾的 LIS 长度是 3

以  $a[5] = 7$  为结尾的 LIS 长度是 4

# 实战：最长上升子序列（LIS）

蛮力搜索本来要遍历状态空间中的所有可能上升序列

但我们可以选取“**结尾数，最长长度**”作为一系列序列的**代表**

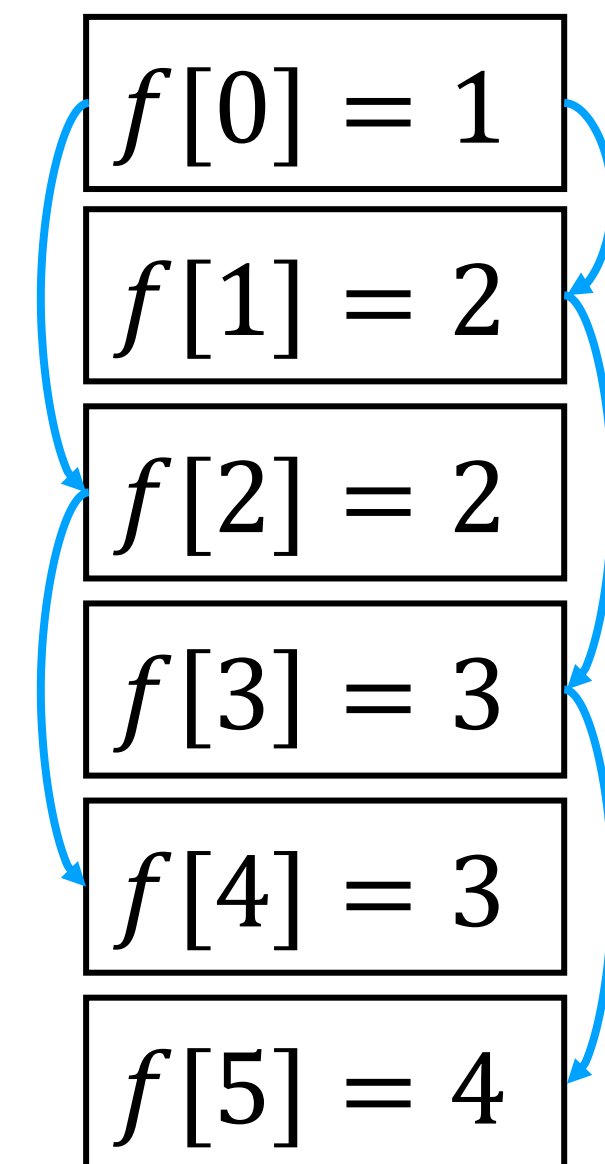
并且它们之间还具有推导关系——找到了最优子结构！

$f[i]$  表示前  $i$  个数构成的以  $a[i]$  为结尾的最长上升子序列的长度

$$f[i] = \max_{j < i, a[j] < a[i]} \{f[j] + 1\}$$

边界：  $f[i] = 1$  ( $0 \leq i < n$ )

目标：  $\max_{0 \leq i < n} \{f[i]\}$



# 动态规划解题步骤



关注 “轮廓变化”

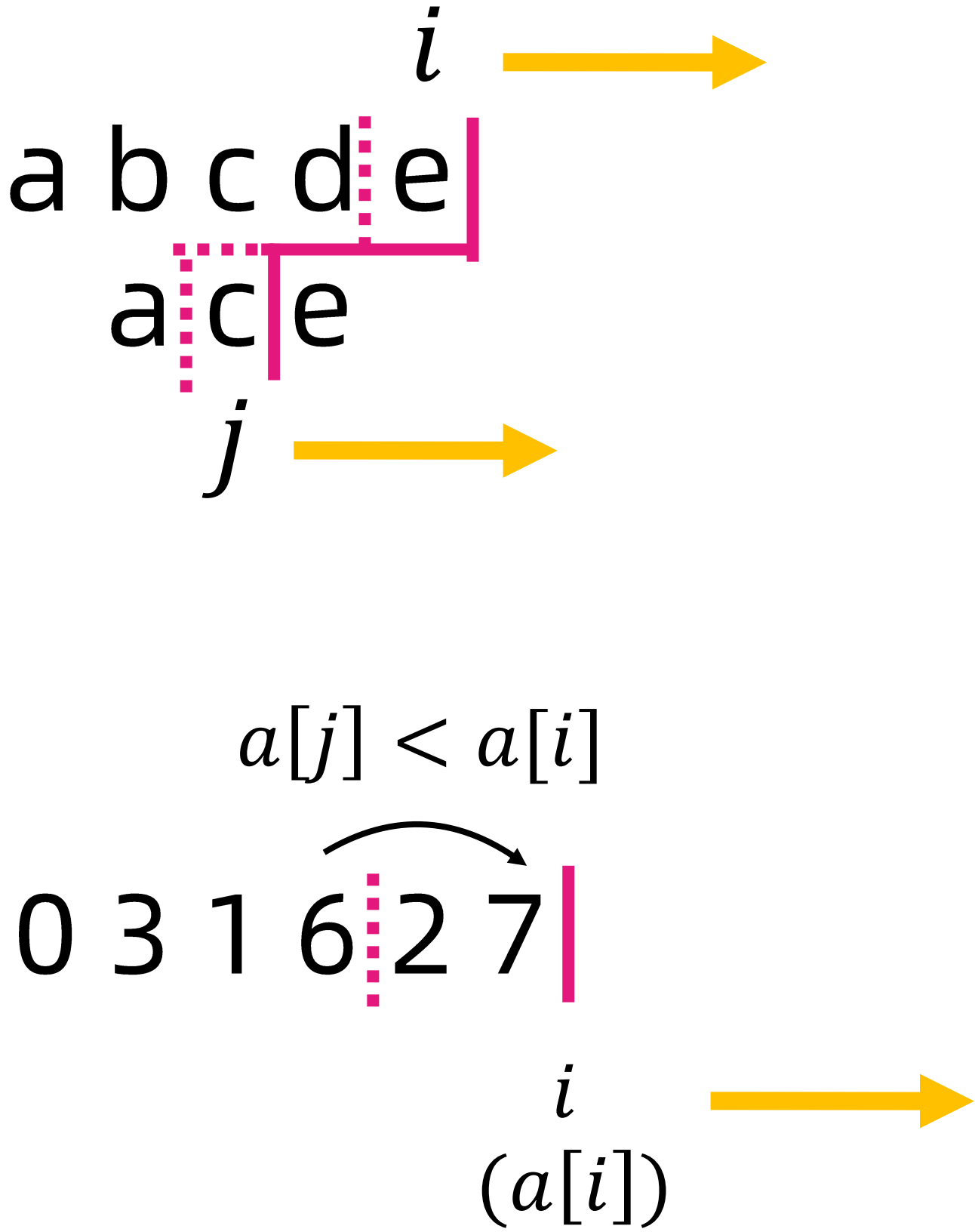
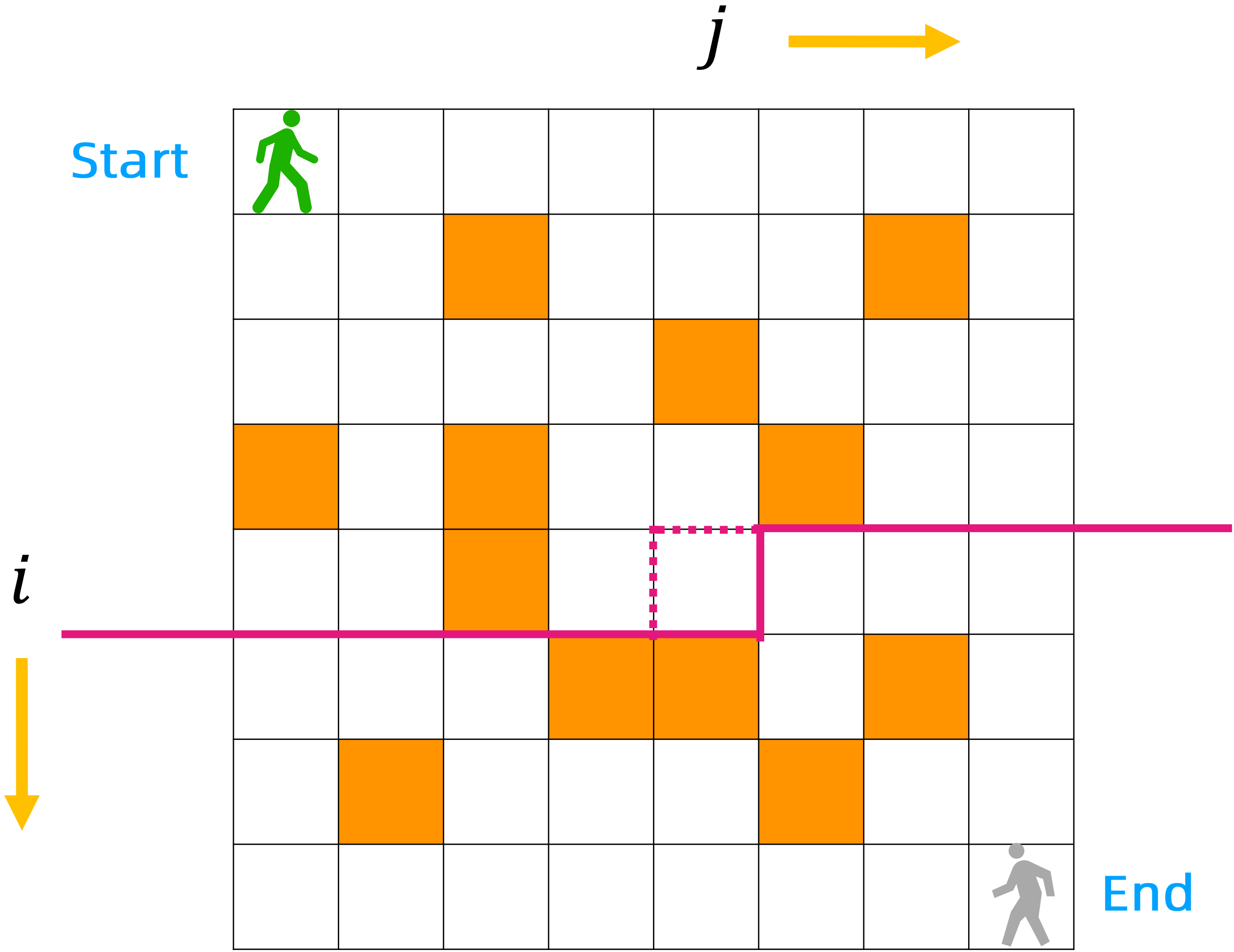
关注 “代表” 以及它们之间的推导关系

关注人力模拟时做出的决策

边界包括：起点、会访问到的不合法状态



# 动态规划 “轮廓变化”



# 动态规划打印方案

动态规划题目打印方案的原则：**记录转移路径 + 递归输出**

动态规划选取“代表”，维护了一个最优子结构

如果记录每个最优子结构的详细方案，时间复杂度会上升

以 LCS 为例，本来是  $O(len^2)$ ，每个  $opt[i, j]$  记录一个方案（字符串），就变成了  $O(len^3)$

正确做法：

记录每个  $f[i, j]$  是从哪里转移过来的（ $f[i - 1, j]$ ， $f[i, j - 1]$  还是  $f[i - 1, j - 1]$ ）

整个动规完成，求出  $f[n, m]$  后，再从  $(n, m)$  开始递归复原最优路径

# 实战：最大子序和

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

$f[i]$  表示以  $i$  为结尾的最大子序和

$$f[i] = \max(f[i - 1] + \text{nums}[i], \text{nums}[i])$$

边界:  $f[0] = \text{nums}[0]$

目标:  $\max_{0 \leq i < n} f[i]$

状态中何时需要“包含结尾”？

——当结尾参与判定条件时，例如 LIS 中  $a[j] < a[i]$ ，此处子序和要“连续”

# 实战：乘积最大子数组

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

$f[i]$  表示以  $i$  为结尾的乘积最大子数组

$f[i] = \max(f[i - 1] * \text{nums}[i], \text{nums}[i])$  —— 这样还对吗？

请注意，“代表”之间要有推导关系，才满足最优子结构！

当  $\text{nums}[i]$  是负数的时候， $\max$  还能推导出  $\max$  吗？

$\max$  和  $\min$  一起作为代表，才满足最优子结构！

$f_{\max}[i], f_{\min}[i]$  表示以  $i$  为结尾的乘积最大、最小子数组

$f_{\max}[i] = \max(f_{\max}[i - 1] * \text{nums}[i], f_{\min}[i - 1] * \text{nums}[i], \text{nums}[i])$

$f_{\min}[i] = \min(f_{\max}[i - 1] * \text{nums}[i], f_{\min}[i - 1] * \text{nums}[i], \text{nums}[i])$

# Homework

爬楼梯

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

三角形最小路径和

<https://leetcode-cn.com/problems/triangle/description/>

最长递增子序列的个数

<https://leetcode-cn.com/problems/number-of-longest-increasing-subsequence/>



# THANKS

 极客时间 | 训练营