

九章算法班2021版直播第9章

最难的算法

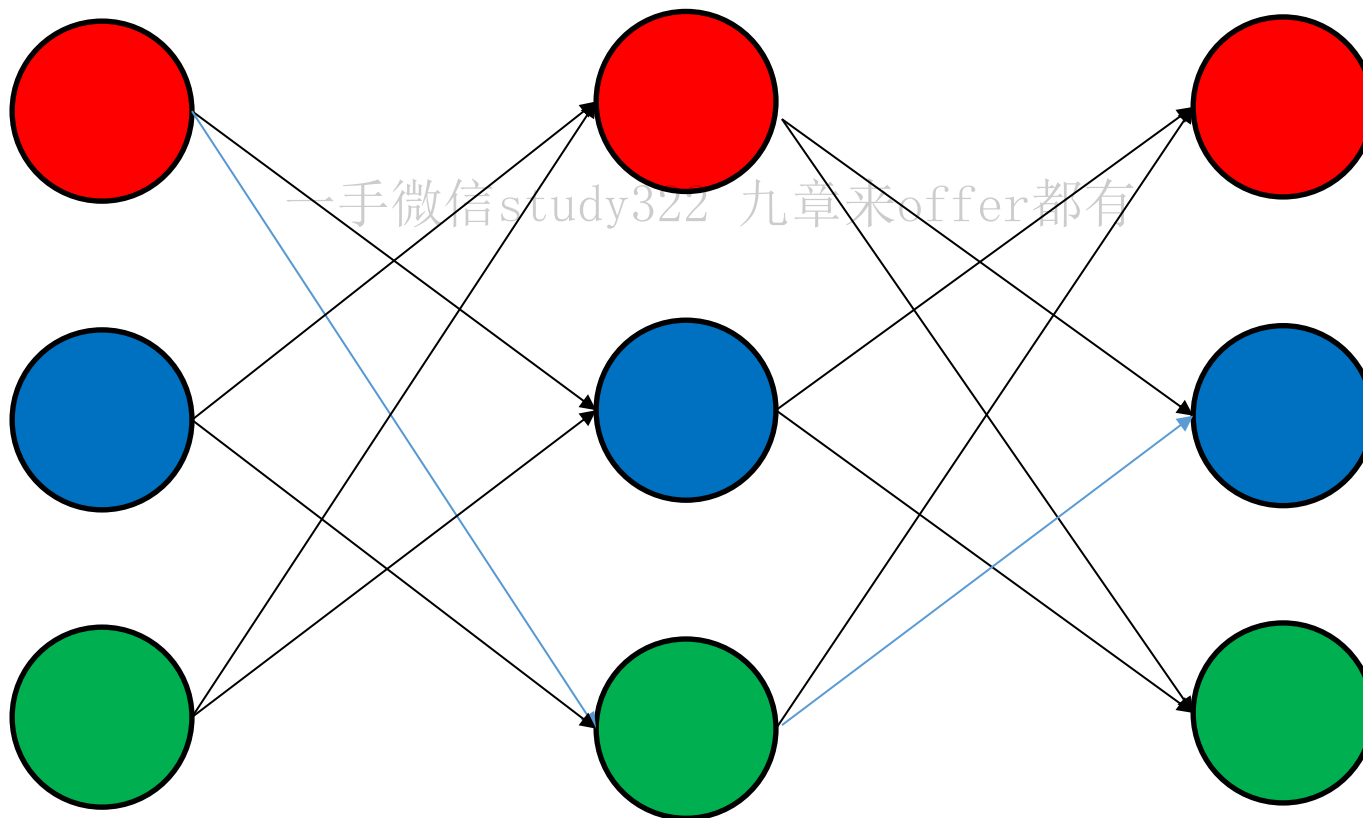
一手微信study322 九章来offer都有

—— 动态规划 Dynamic Programming

主讲：夏天

由大化小

动态规划的算法思想：大规模问题的依赖于小规模问题的计算结果



动态规划的两种实现方式

- 递归 (Recursion) 方式 => 记忆化搜索 (上节课)
- 循环 (Iteration) 方式 (本节课)

一手微信study322 九章来offer都有

记忆化搜索时间复杂度

= 动态规划的时间复杂度

= $O(\text{状态总数} * \text{计算每个状态的时间耗费})$

一手微信study322 九章来offer都有

动态规划为什么难？

动态规划是一种算法范式（算法思想，Algorithmic Paradigm，而不是很具体的算法（Algorithm））

每个子类型都是一个新的算法

学习周期很长，一月入门，两月上手

一手微信study322 九章来offer都有



能别讲了嘛
我真的学不动了

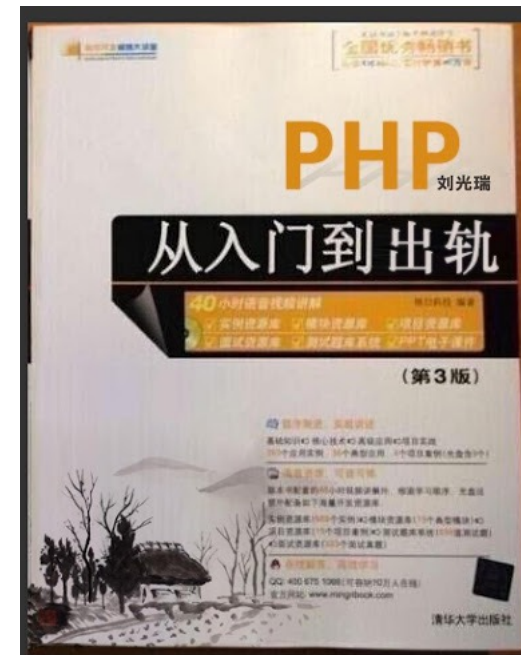
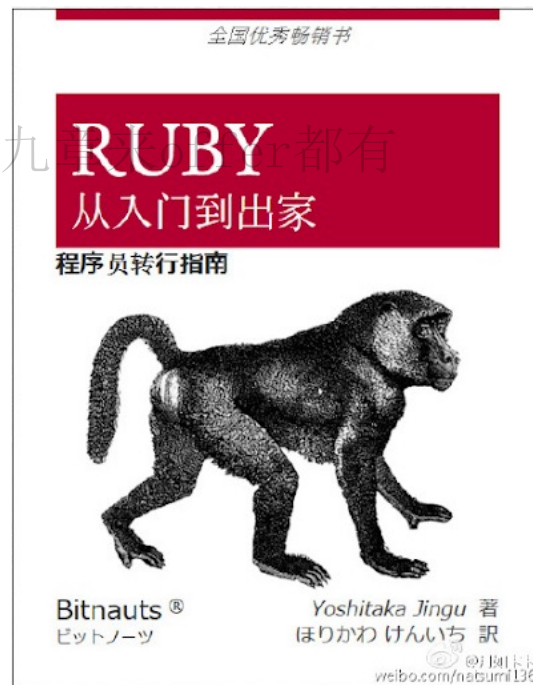
我可以放弃么？

北美求职者：如果你不想去Google等某些炙手可热的公司，可以

国内求职者：如果你不去大厂，可以（国内大厂面试题超过 50% 的题都是动态规划）



知乎 @吾牛



三种适用动规的场景

求可行性

- $dp[]$ 的值是 true / false
- $dp[\text{大问题}] = dp[\text{小问题1}] \text{ or } dp[\text{小问题2}] \text{ or } \dots$
- 代码通常用 for 小问题 if $dp[\text{小问题}] == \text{true}$ then break 的形式实现

求方案数

- $dp[]$ 的值的类型是方案数 (整数)
- $dp[\text{大问题}] = \sum(dp[\text{小问题1}], dp[\text{小问题2}], \dots)$
- $\sum = \text{sum}$

一手微信study322 九章来offer都有

求最值

- $dp[]$ 的值的类型是最优值的类型
- $dp[\text{大问题}] = \max\{dp[\text{小问题1}], dp[\text{小问题2}], \dots\}$
- $dp[\text{大问题}] = \min\{dp[\text{小问题1}], dp[\text{小问题2}], \dots\}$

三种不适用 DP 的场景

求所有的具体方案

- <http://www.lintcode.com/problem/palindrome-partitioning/>
- 只求出一个具体方案还是可以用 DP 来做的（下节课）
- 该判断标准成功率 99%

输入数据是无序的

- <http://www.lintcode.com/problem/longest-consecutive-sequence/>
- 背包类动态规划不适用此判断条件，除去背包问题后方向：逐行生成数据
- 该判断标准成功率 60-70%，有一些题可以先排序之后按序处理

暴力算法的复杂度已经是多项式级别

- <http://www.lintcode.com/problem/largest-rectangle-in-histogram/>
- 动态规划擅长与优化指数级别复杂度($2^n, n!$)到多项式级别复杂度(n^2, n^3)
- 不擅长优化 n^3 到 n^2
- 该判断标准成功率 80%

则极不可能使用动态规划求解

则极不可能使用动态规划求解

第一步

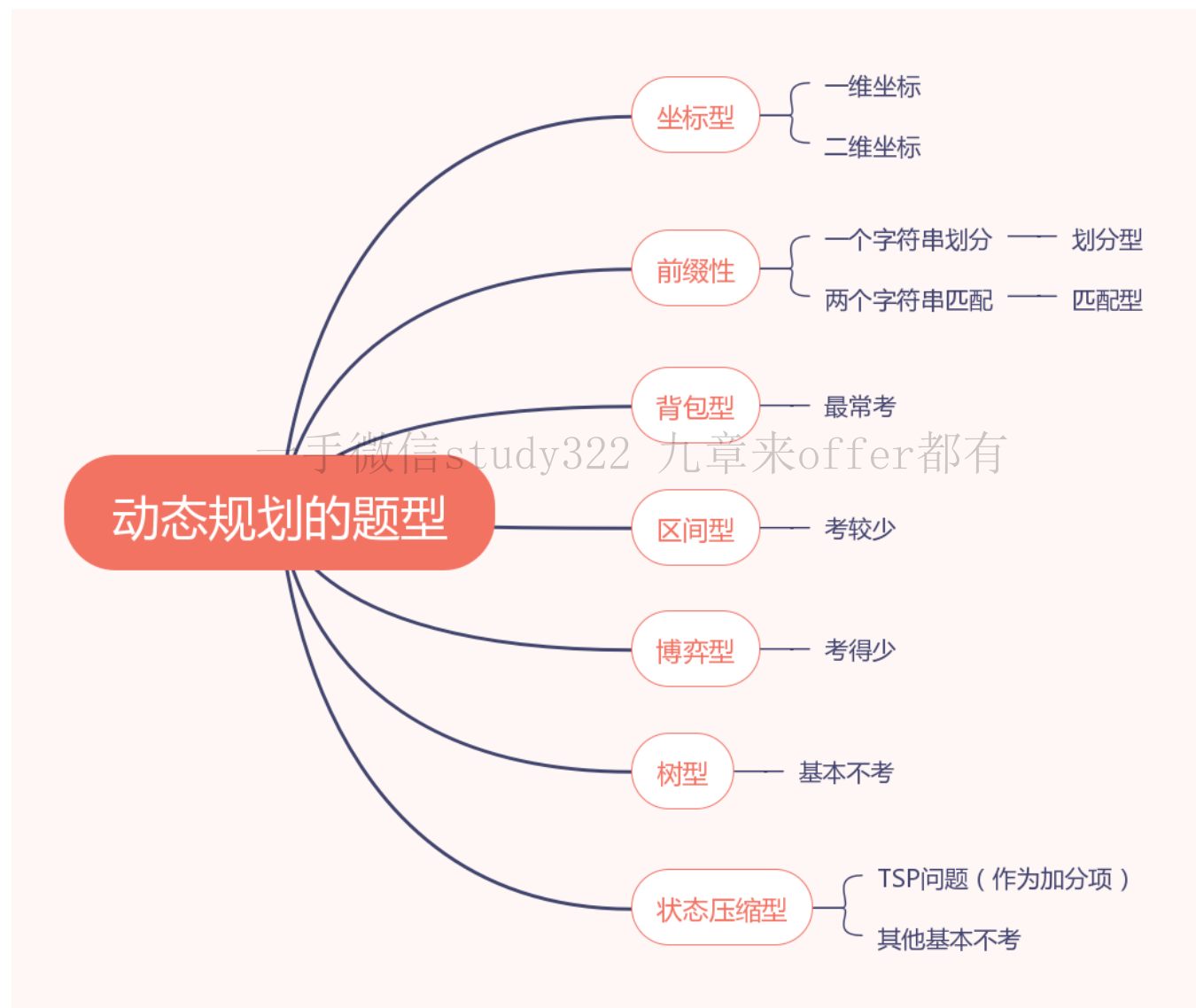
判断是否能够使用动态规划算法

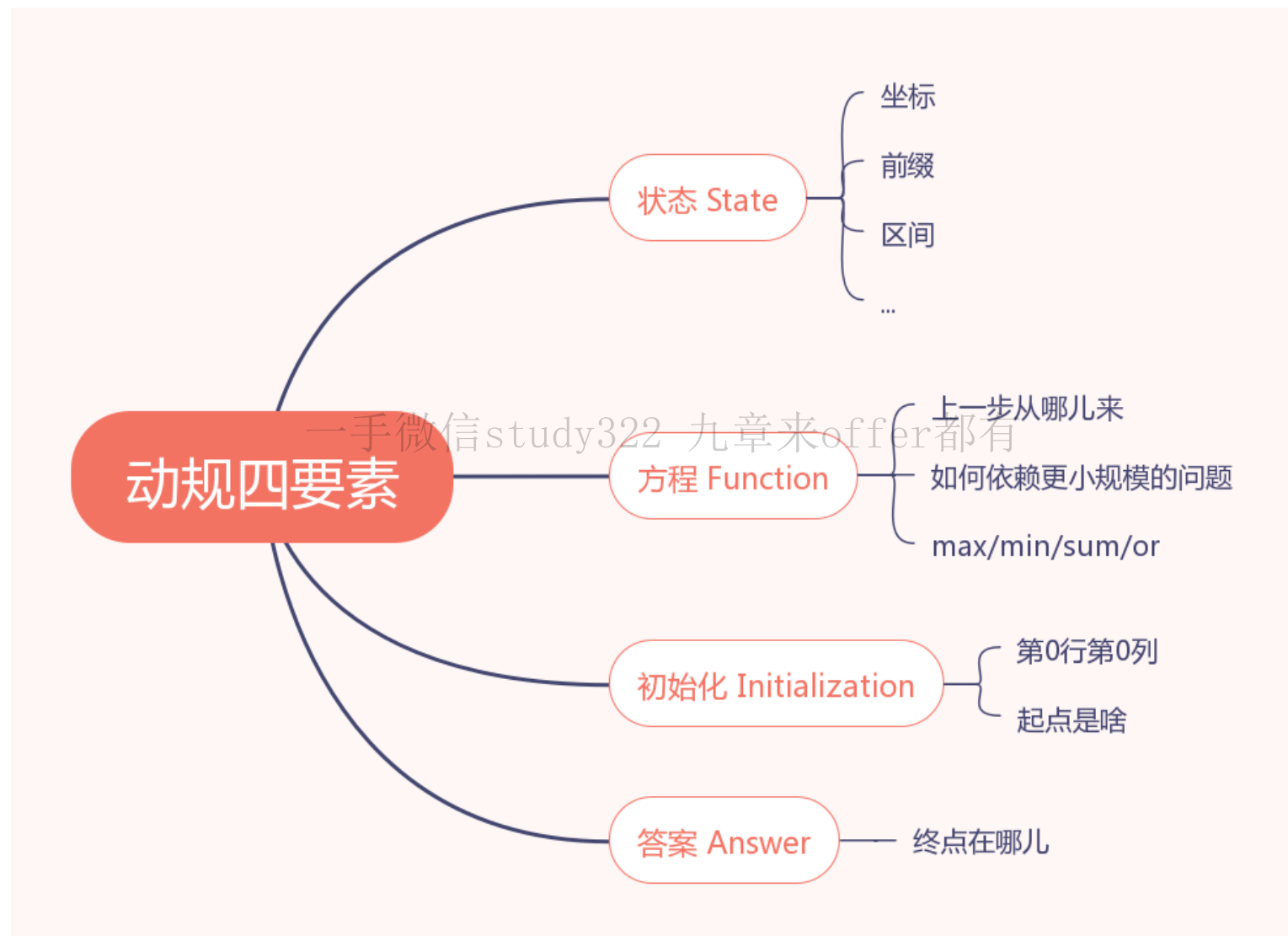
第二步

判断动态规划的题型

第三步

使用动规四要素进行解题





动态规划的空间优化技巧

一手微信study322 九章来offer都有

—— 滚动数组 Rolling Array

滚动数组 Rolling Array

如果状态依赖关系只在相邻的几层之间

则可以使用滚动数组进行优化

滚动数组可以让空间复杂度降维

一手微信study322 九章来offer都有

109 Triangle 数字三角形

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

给定一个数字三角形，找到从顶部到底部的最小路径和。每一步可以移动到下面一行的相邻数字上。

输入：

```
triangle = [  
    [2],  
    [3, 4],  
    [6, 5, 7],  
    [4, 1, 8, 3]  
]
```

输出：

11

解释：

从顶到底部的最小路径和为11（ $2 + 3 + 5 + 1 = 11$ ）。

具有有序性，求最优解，有可能使用动态规划

坐标型动态规划

状态的转移是考虑是从哪儿走到(i,j)这个坐标的

一手微信study322 九章来offer都有



$dp[i][j]$ 代表以(0, 0)为起点，以(i, j)为终点的最小路径和

2 (2)			
3 (5)	4 (6)		
6 (11)	5 (10)	7 (13)	
4 (15)	1 (11)	8 (18)	3 (16)



2	10	13	
5	6	18	16

空间复杂度： $O(2*N) = O(N)$

空间复杂度： $O(N*N) = O(N^2)$ 时间复杂度： $O(N^2)$ ，两层嵌套for循环

```
6 public int minimumTotal(int[][] triangle) {
7     // 特殊情况处理
8     if (triangle == null || triangle.length == 0 ||
9         triangle[0] == null || triangle[0].length == 0) {
10         return -1;
11     }
12
13     // 三角形边长为n，存储在n行n列矩阵
14     int n = triangle.length;
15
16     // 状态 (State) : dp[x][y] = 从三角形顶点(0, 0)到(x, y)的最小路径和
17     int[][] dp = new int[n][n];
18
19     // 初始化 (Initialization)
20     // dp[0][0]点的最短路径和为三角形中(0, 0)点的值
21     dp[0][0] = triangle[0][0];
22
23     // 自顶向下的方式的多重循环动态规划
24     for (int i = 1; i < n; i++) {
25         // 初始化本行的最左点与最右点
26         dp[i%2][0] = dp[(i-1)%2][0] + triangle[i][0];
27         dp[i%2][i] = dp[(i-1)%2][i-1] + triangle[i][i];
28         for (int j = 1; j < i; j++) {
29             // 方程 (Function):
30             // dp[i][j] = min(dp[i-1][j-1], dp[i-1][j]) + triangle[i][j]
31             // 通过上层左右两点推得当前点的最小路径
32             dp[i%2][j] = Math.min(dp[(i-1)%2][j-1], dp[(i-1)%2][j])
33                 + triangle[i][j];
34         }
35     }
36
37     // 答案 (Answer) : 三角形底层任意一个节点都有可能是最小路径的终点
38     // 遍历所有路径终点，返回最小值
39     int best = dp[(n-1)%2][0];
40     for (int j = 1; j < n; j++) {
41         best = Math.min(best, dp[(n-1)%2][j]);
42     }
43     return best;
44 }
```

```
6     def minimumTotal(self, triangle):
7         # 特殊情况处理
8         if not triangle or not triangle[0]:
9             return -1
10
11        # 三角形边长为n, 存储在n行n列矩阵
12        n = len(triangle)
13
14        # 状态 (State) : dp[x][y] = 从三角形顶点(0, 0)到(x, y)的最小路径和
15        dp = [[0] * n, [0] * n]
16
17        # 初始化 (Initialization)
18        # dp[0][0]点的最短路径和为三角形中(0, 0)点的值
19        dp[0][0] = triangle[0][0]
20
21        # 自顶向下的方式的多重循环动态规划
22        for i in range(1, n):
23            # 初始化本行的最左点与最右点
24            dp[i % 2][0] = dp[(i - 1) % 2][0] + triangle[i][0]
25            dp[i % 2][i] = dp[(i - 1) % 2][i - 1] + triangle[i][i]
26            # 方程 (Function):
27            # dp[i][j] = min(dp[i-1][j - 1], dp[i-1][j]) + triangle[i][j]
28            # 通过上层左右两点推得当前点的最小路径
29            for j in range(1, i):
30                dp[i % 2][j] = min(dp[(i - 1) % 2][j], dp[(i - 1) % 2][j - 1]) + triangle[i][j]
31
32        # 答案 (Answer) : 三角形底层行任意一个节点都有可能是最小路径的终点
33        # 遍历所有路径终点, 返回最小值
34        return min(dp[(n - 1) % 2])
```


朝哪个方向滚？

方向：逐行生成数据

当前格子 = 上面格子 + 左面格子, $dp[i][j] = dp[i-1][j] + dp[i][j-1]$

0	1	2	3
0	1	3	6

方向：逐行生成数据

当前格子 = 上面格子 + 右面格子, $dp[i][j] = dp[i-1][j] + dp[i][j+1]$

0	1	2	3
6	6	5	3

套路

逐行（列）生成数据，就在行（列）上滚动

逐行滚动可能从左至右，或者从右至左；逐列滚动可能从上至下，或者从下至上

从非滚动变成滚动，只需要：在行（列）上滚动，行（列） $\text{index} \% \text{滚动行（列）数}$

方向：逐列生成数据

当前格子 = 上面格子 + 左面格子, $dp[i][j] = dp[i-1][j] + dp[i][j-1]$

0	0
1	1
2	3
3	6

方向：逐列生成数据

当前格子 = 下面格子 + 左面格子, $dp[i][j] = dp[i+1][j] + dp[i][j-1]$

0	6
1	6
2	5
3	3

可以继续压缩吗？

方向：逐行生成数据

当前格子 = 上面格子 + 左面格子， $dp[i][j] = dp[i - 1][j] + dp[i][j - 1]$

0	1	2	3
0	1	3	6
0	1	4	10

可以行和列都滚动吗？

一手微信study322 九章来offer都有

0	1	2	3

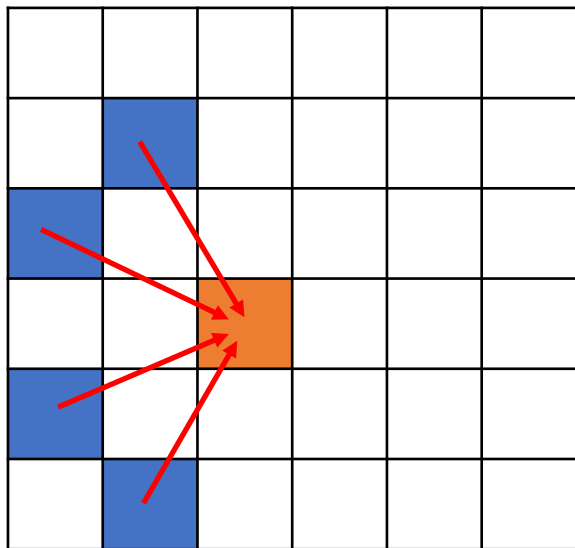
可以两行变一行吗？

0	1	2	3

630 Knight Shortest Path II

Given a knight in a chessboard $n * m$ (a binary matrix with 0 as empty and 1 as barrier). the knight initialize position is (0, 0) and he wants to reach position (n - 1, m - 1), **Knight can only be from left to right**. Find the shortest path to the destination position, return the length of the route. Return -1 if knight can not reach.

在一个 $n * m$ 的棋盘(二维矩阵中 0 表示空 1 表示有障碍物), 骑士的初始位置是 (0, 0), 他想要达到 (n - 1, m - 1) 这个位置, **骑士只能从左边走到右边**。找出骑士到目标位置所需要走的最短路径并返回其长度, 如果骑士无法达到则返回 -1.



输入: 一手微信study322 九章来offer都有

[[0,0,0,0],[0,0,0,0],[0,0,0,0]]

输出:

3

解释:

[0,0]->[2,1]->[0,2]->[2,3]

输入:

[[0,1,0],[0,0,1],[0,0,0]]

输出:

-1

具有有序性, 求最优解, 有可能使用动态规划

坐标型动态规划

状态的转移是考虑: 从哪儿走到(i,j)这个坐标的

思路分析 & 代码解析

求从左上角到右
下角的最短路径

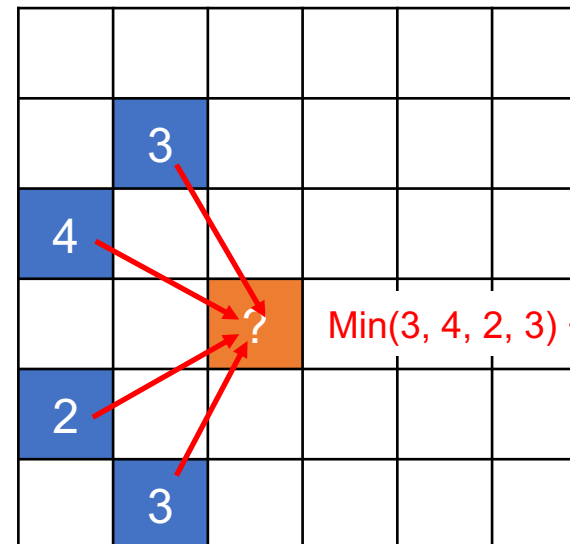
0	max	2	max
max	max	1	2
max	1	max	3

```
1  # 四个“之前点”的偏移量
2  # 可以看到，所有deltaY都是负数，保证从左到右的顺序
3  DIRECTIONS = [
4      (-1, -2),
5      (1, -2),
6      (-2, -1),
7      (2, -1),
8  ]
9
10 class Solution:
11     # @param {boolean[][]} grid a chessboard included 0 and 1
12     # @return {int} the shortest path
13     def shortestPath2(self, grid):
14         # 特殊情况处理
15         if not grid or not grid[0]:
16             return -1
17
18         # 棋盘的行数和列数
19         n, m = len(grid), len(grid[0])
20
21         # 状态 (State) : dp[i][j % 3] 代表从起点走到 i, j 的最短路径长度
22         # 初始化 (Initialization)
23         # 第0列除 (0, 0) 点，其他点均不可达，第最短路径初始值为最大值
24         dp = [[float('inf')] * 3 for _ in range(n)]
25
26         # (0, 0)的最短路径为0
27         dp[0][0] = 0
```

```
29 # 方程 (Function): dp[i][j % 3] = min(dp[x][y % 3]) + 1, (x,y)是i,j的上一步
30 # 因为是从左到右走的，所以外层循环是列，内层循环是行
31 for j in range(1, m):
32     for i in range(n):
33         # 1为障碍物，不可到达
34         if grid[i][j]:
35             continue
36
37         # 从四个方向上，遍历 (i, j) 的上一步
38         for delta_x, delta_y in DIRECTIONS:
39             x, y = i + delta_x, j + delta_y
40             # 如果(x, y)在界内，则状态转移推导，在所有的路径中选择最小值
41             if 0 <= x < n and 0 <= y < m:
42                 dp[i][j % 3] = min(dp[i][j % 3], dp[x][y % 3] + 1)
43
44 print(dp)
45 # 答案 (Answer) : dp[n - 1][(m - 1) % 3]
46 # 如果右下角可以到达，返回右下角
47 if dp[n - 1][(m - 1) % 3] == float('inf'):
48     return -1
49 return dp[n - 1][(m - 1) % 3]
```

逐列生成数据，就在列上滚动

从非滚动变成滚动，只需要：在列上滚动，列index % 滚动行（列）数



$$\text{Min}(3, 4, 2, 3) + 1 = 2 + 1 = 3$$

0	max	2	max
max	max	1	2
max	1	max	3

	3				
4					
		?			
2					
	3				

$$\text{Min}(3, 4, 2, 3) + 1 = 2 + 1 = 3$$

```

1 public class Solution {
2     // 四个“之前点”的偏移量
3     // 可以看到，所有deltaY都是负数，保证从左到右的顺序
4     public static int[] deltaX = {-2, -1, 1, 2};
5     public static int[] deltaY = {-1, -2, -2, -1};
6
7     /**
8      * @param grid: a chessboard included 0 and 1
9      * @return: the shortest path
10     */
11     public int shortestPath2(boolean[][] grid) {
12         // 特殊情况处理
13         if (grid == null || grid.length == 0) {
14             return -1;
15         }
16         if (grid[0] == null || grid[0].length == 0) {
17             return -1;
18         }
19
20         // 棋盘的行数与列数
21         int n = grid.length, m = grid[0].length;
22
23         // 状态 (State) : dp[i][j % 3] 代表从起点走到 i, j 的最短路径长度
24         int[][] dp = new int[n][3];
25
26         // 初始化 (Initialization)
27         // 第0列除 (0, 0) 点，其他点均不可达，第最短路径初始值为最大值
28         for (int i = 1; i < n; i++) {
29             dp[i][0] = Integer.MAX_VALUE;
30         }
31         // (0,0)的最短路径为0
32         dp[0][0] = 0;

```

```

34     // 方程 (Function): dp[i][j % 3] = min(dp[x][y % 3]) + 1
35     // (x,y) 是 i,j 的上一步
36     // 因为是从左到右走的，所以外层循环是列，内层循环是行
37     for (int j = 1; j < m; j++) {
38         for (int i = 0; i < n; i++) {
39             dp[i][j % 3] = Integer.MAX_VALUE;
40             // 1为障碍物，不可到达
41             if (grid[i][j]) {
42                 continue;
43             }
44             // 从四个方向上，遍历 (i, j) 的上一步
45             for (int direction = 0; direction < 4; direction++) {
46                 // 得到上一步坐标
47                 int x = i + deltaX[direction];
48                 int y = j + deltaY[direction];
49                 // 如果出界，continue
50                 if (x < 0 || x >= n || y < 0 || y >= m) {
51                     continue;
52                 }
53                 // 如果 (x, y) 路径为最大值，不可到达，continue
54                 // 为什么python没有这部分代码？因为python会自动处理数字溢出
55                 if (dp[x][y % 3] == Integer.MAX_VALUE) {
56                     continue;
57                 }
58                 // 在所有路径中选择最小值
59                 dp[i][j % 3] = Math.min(dp[i][j % 3], dp[x][y % 3] + 1);
60             }
61         }
62     }
63
64     // 答案 (Answer) : dp[n - 1][(m - 1) % 3]
65     // 如果右下角可以到达，返回右下角
66     if (dp[n - 1][(m - 1) % 3] == Integer.MAX_VALUE) {
67         return -1;
68     }
69
70     return dp[n - 1][(m - 1) % 3];
71 }

```

滚动数组滚动的是第一重循环的变量，而不是第二重甚至第三重

外层循环决定了是逐行还是逐列。如果外层循环是列，就是逐列；如果外层循环是行，就是逐行。

滚动数组也只能滚一个维度，不能两个维度一起滚动

逐行（列）生成数据，就在行（列）上滚动

逐行滚动可能从左至右，或者从右至左；逐列滚动可能从上至下，或者从下至上

从非滚动变成滚动，只需要：在行（列）上滚动，行（列） $\text{index} \% \text{滚动行（列）数}$

快扶我起来
我还能学

一手微信study322 九章来offer都有



76 Longest Increasing Subsequence 最长上升子序列

Given a sequence of integers, find the longest increasing subsequence (LIS).
You code should return the length of the LIS.

给定一个整数序列，找到最长上升子序列（LIS），返回LIS的长度。

要求时间复杂度为 $O(n^2)$ 或者 $O(n\log n)$

输入:

nums = [5,4,1,2,3]

输出:

3

解释:

LIS 是 [1,2,3]

输入:

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

输出:

4

解释:

LIS 是 [2,4,5,7]

具有有序性，求最优解，有可能使用动态规划

坐标型动态规划

状态的转移是考虑是从哪儿走到(i,j)这个坐标的

下标	0	1	2	3	4	5
数值	4	2	4	5	3	7
dp	1	1	2	3	2	4



$$dp[2] = \max(dp[1]) + 1 = 1 + 1 = 2$$

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

$$dp[4] = \max(dp[1]) + 1 = 1 + 1 = 2$$

$$dp[5] = \max(dp[0], dp[1], dp[2], dp[3], dp[4]) + 1 = 3 + 1 = 4$$

```
6 def longestIncreasingSubsequence(self, nums):
7     # 特殊情况处理
8     if nums is None or not nums:
9         return 0
10
11     # 状态 (State) : dp[i] 表示从最左开始, 到第i个数结尾的LIS
12     # 初始化 (Initialization) : dp[0..n-1] = 1
13     # 所有点的LIS初始长度为1, 仅包含自身
14     dp = [1] * len(nums)
15
16     # 方程 (Function): max{dp[j] + 1}, j < i && nums[j] < nums[i]
17     # 如果j在i的前面, # 方程 (Function): 并且nums[j] < nums[i],
18     # 那么j点和点可以拼凑成上升序列
19     # 在所有的上升序列中找到最长的, 就是从起点到i点的最长子序列
20     for i in range(len(nums)):
21         for j in range(i):
22             if nums[j] < nums[i]:
23                 dp[i] = max(dp[i], dp[j] + 1)
24
25     print(dp)
26     # 答案 (Answer) : max{dp[0..n-1]}
27     # 最长子序列可能以任意一点为终点,
28     # 在所有Increasing Subsequence中寻找最大值
29     return max(dp)
```

```
2 public int longestIncreasingSubsequence(int[] nums) {
3     // 特殊情况处理
4     if (nums == null || nums.length == 0) {
5         return 0;
6     }
7
8     // 数列长度
9     int n = nums.length;
10
11     // 状态 (State) : dp[i] 表示以第i个数结尾的LIS
12     int[] dp = new int[n];
13
14     // 初始化 (Initialization) : dp[0..n-1] = 1
15     // 所有点的LIS初始长度为1, 仅包含自身
16     for (int i = 0; i < n; i++) {
17         dp[i] = 1;
18     }
19
20     // 方程 (Function): max{dp[j] + 1}, j < i && nums[j] < nums[i]
21     // 如果j在i的前面, 并且nums[j] < nums[i], 那么j点和点可以拼凑成上升序列
22     // 在所有的上升序列中找到最长的, 就是从起点到i点的最长子序列
23     for (int i = 0; i < n; i++) {
24         for (int j = 0; j < i; j++) {
25             if (nums[i] > nums[j]) {
26                 dp[i] = Math.max(dp[i], dp[j] + 1);
27             }
28         }
29     }
30
31     // 答案 (Answer) : max{dp[0..n-1]}
32     // 最长子序列可能以任意一点为终点,
33     // 在所有子序列中寻找最大值
34     int max = 0;
35     for (int end = 0; end < n; end++) {
36         max = Math.max(max, dp[end]);
37     }
38     return max;
39 }
```

如何获得Longest Increasing Subsequence的具体方案？

动态规划算法虽然不擅于找所有方案, 但是找最优值的具体方案还是可以的

倒推法

记录每个状态的最优值是从哪个前继状态来的

通常需要一个和状态数组同样维度的数组

prev[i] 记录使得dp[i]获得最优值的那个j是谁

j是方程 $dp[i] = \max\{dp[j] + 1\}$ 里的j

一手微信study322 九章来offer都有

思路分析 & 代码解析

下标	0	1	2	3	4	5
数值	4	2	4	5	3	7
dp	1	1	2	3	2	4
prev	-1	-1	1	2	1	3



$$dp[2] = \max(dp[1]) + 1 = 1 + 1 = 2$$

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

$$dp[4] = \max(dp[1]) + 1 = 1 + 1 = 2$$

$$dp[5] = \max(dp[0], dp[1], dp[2], dp[3], dp[4]) + 1 = 3 + 1 = 4$$

```
6 def longestIncreasingSubsequence(self, nums):
7     # 特殊情况处理
8     if nums is None or not nums:
9         return 0
10
11     # 状态 (State) : dp[i]表示从最左开始, 到第i个数结尾的LIS
12     # 初始化 (Initialization) : dp[0..n-1] = 1
13     # 所有点的LIS初始长度为1, 仅包含自身
14     dp = [1] * len(nums)
15
16     # prev[i]代表dp[i]的最优值是从哪个dp[j]推导过来的
17     # (也就是i的前一个点是哪个点)
18     prev = [-1] * len(nums)
```

```
20 # 方程 (Function): max{dp[j] + 1}, j < i && nums[j] < nums[i]
21 # 如果j在i的前面, 并且nums[j] < nums[i], 那么j点和点可以拼凑成上升序列
22 # 在所有的上升序列中找到最长的, 就是从起点到i点的最长子序列
23 for i in range(len(nums)):
24     for j in range(i):
25         if nums[j] < nums[i] and dp[i] < dp[j] + 1:
26             dp[i] = dp[j] + 1
27             prev[i] = j
28
29 # 答案 (Answer) : max{dp[0..n-1]}
30 # 最长子序列可能以任意一点为终点,
31 # 在所有Increasing Subsequence中寻找最大值, 并记录最大值所对应的下标last
32 longest, last = 0, -1
33 # 倒推最优解路径
34 for i in range(len(nums)):
35     if dp[i] > longest:
36         longest = dp[i]
37         last = i
38
39 path = []
40 # 如果last为-1, 则没有先导点了, 已经到了路径的最开端
41 while last != -1:
42     path.append(nums[last])
43     last = prev[last]
44 # 通过python的切片操作, 翻转得到正序路径, 打印输出
45 print(path[::-1])
46
47 return longest
```

下标	0	1	2	3	4	5
数值	4	2	4	5	3	7
dp	1	1	2	3	2	4
prev	-1	-1	1	2	1	3

```
3 public int longestIncreasingSubsequence(int[] nums) {
4     // 特殊情况处理
5     if (nums == null || nums.length == 0) {
6         return 0;
7     }
8     // 数列长度
9     int n = nums.length;
10
11     // 状态 (State) : dp[i] 表示从最左开始, 到第i个数结尾的LIS
12     // prev[i]代表dp[i]的最优值是从哪个dp[j]推导过来
13     // 的 (也就是i的前一个点是哪个点)
14     int[] prev = new int[n];
15     int[] dp = new int[n];
16
17     // 初始化 (Initialization) : dp[0..n-1] = 1
18     // 所有点的LIS初始长度为1, 仅包含自身
19     // prev[i] = -1表示i的先导点未知
20     for (int i = 0; i < n; i++) {
21         dp[i] = 1;
22         prev[i] = -1;
23     }
```

```
25 // 方程 (Function): max{dp[j] + 1, j < i && nums[j] < nums[i]}
26 // 如果j在i的前面, 并且nums[j] < nums[i], 那么j点和点可以拼凑成上升序列
27 // 在所有的上升序列中找到最长的, 就是从起点到i点的最长子序列
28 for (int i = 0; i < n; i++) {
29     for (int j = 0; j < i; j++) {
30         if (nums[j] < nums[i] && dp[j] + 1 > dp[i]) {
31             dp[i] = dp[j] + 1;
32             // 在更新dp[i]时, 同时把i的先导点更新为j
33             prev[i] = j;
34         }
35     }
36 }
```

```
38 // 答案 (Answer) : max{dp[0..n-1]}
39 // 最长子序列可能以任意一点为终点,
40 // 在所有Increasing Subsequence中寻找最大值, 并记录最大值所对应的下标last
41 int longest = 0, last = -1;
42 for (int i = 0; i < n; i++) {
43     if (dp[i] > longest) {
44         longest = dp[i];
45         last = i;
46     }
47 }
48
49 // 倒推最优解路径, 打印输出
50 ArrayList<Integer> path = new ArrayList();
51 // 如果last为-1, 则没有先导点了, 已经到了路径的最开端
52 while (last != -1) {
53     path.add(nums[last]);
54     last = prev[last];
55 }
56 // 翻转打印正序路径, 用-作为分隔符
57 for (int i = path.size() - 1; i >= 0; i--) {
58     System.out.print(path.get(i) + "-");
59 }
```

```
60
61 return longest;
62 }
```

398 Longest Continuous Increasing Subsequence II 最长上升连续子序列 II

Given an integer matrix. Find the longest increasing continuous subsequence in this matrix and return the length of it.

The longest increasing continuous subsequence here can start at any position and go up/down/left/right.

给定一个整数矩阵. 找出矩阵中的最长连续上升子序列, 返回它的长度.

最长连续上升子序列可以从任意位置开始, 向上/下/左/右移动.

输入:

1	2	3	4	5
16	17	24	23	6
15	18	25	22	7
14	19	20	21	8
13	12	11	10	9

输出:

25

解释:

1 -> 2 -> 3 -> 4 -> 5 -> ... -> 25 (由外向内螺旋)

具有有序性, 求最优解, 有可能使用动态规划

一手微信study322 坐标型动态规划 都有

状态的转移是考虑: 从哪儿走到(i, j)这个坐标的

1 (1)	4 (2)	5 (3)
6 (2)	2 (1)	8 (4)
3 (1)	7 (2)	9 (5)

```

2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
public int longestContinuousIncreasingSubsequence2(int[][] matrix) { 28
    // 特殊情况处理 29
    if (matrix == null || matrix.length == 0 || 30
        matrix[0] == null || matrix[0].length == 0) { 31
        return 0; 32
    } 33

    // 矩阵的行数和列数 34
    int n = matrix.length, m = matrix[0].length; 35
    int[] dx = {0, 0, 1, -1}; 36
    int[] dy = {1, -1, 0, 0}; 37

    // 把矩阵内的每个点转化成 [行, 列, 值] 的形式, 存入list 38
    List<List<Integer>> points = new ArrayList<>(); 39
    for (int i = 0; i < n; i++) { 40
        for (int j = 0; j < m; j++) { 41
            points.add(Arrays.asList(i, j, matrix[i][j])); 42
        } 43
    } 44

    // 按照点的值, 从小到大排序 45
    points.sort((p1, p2) -> Integer.compare(p1.get(2), p2.get(2))); 46

    // 状态 (State) : dp[i][j] 表示以坐标(i,j)点结尾的LCIS的长度 47
    int[][] dp = new int[n][m]; 48
}

```

```

// 按照值的从小到大排序遍历所有点
for (int i = 0; i < points.size(); i++) {
    int x = points.get(i).get(0);
    int y = points.get(i).get(1);

    // 初始化 (Initialization) : 初始LCIS为1
    dp[x][y] = 1;

    // 遍历上下左右四个点(之前的点)
    for (int j = 0; j < 4; j++) {
        int prevX = x - dx[j];
        int prevY = y - dy[j];

        // 如果点出界, 跳过
        if (prevX < 0 || prevX >= n || prevY < 0 || prevY >= m) {
            continue;
        }

        // 方程 (Function): max{上下左右LCIS + 1}
        // 如果周边点小于当前点, 那么longest_hash[(x, y)] + 1为CIS
        // 在所有的CIS中选择LCIS
        if (matrix[prevX][prevY] < matrix[x][y]) {
            dp[x][y] = Math.max(dp[x][y], dp[prevX][prevY] + 1);
        }
    }
}

// 答案 (Answer) : max{ 所有坐标对应的LCIS长度 }
// 最长子序列可能以任意一点为终点,
// 在所有Increasing Subsequence中寻找最大值
int longest = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        longest = Math.max(longest, dp[i][j]);
    }
}

return longest;
}

```

1 (1)	4 (2)	5 (3)
6 (2)	2 (1)	8 (4)
3 (1)	7 (2)	9 (5)

```
6 def longestContinuousIncreasingSubsequence2(self, A):
7     # 特殊情况处理
8     if not A or not A[0]:
9         return 0
10
11     # 矩阵的行数和列数
12     n, m = len(A), len(A[0])
13
14     # 把矩阵内的每个点转化成 (值, 行, 列) 的形式, 存入list
15     points = []
16     for i in range(n):
17         for j in range(m):
18             points.append((A[i][j], i, j))
19
20     # 按照点的值, 从小到大排序
21     points.sort()
```

```
23 # State: dp[(i, j)]表示以坐标(i, j)点结尾的LCIS的长度
24 longest_hash = {}
25 # 按照值的从小到大排序遍历所有点
26 for i in range(len(points)):
27     # 以坐标为key
28     key = (points[i][1], points[i][2])
29     # 初始化 (Initialization): 初始LCIS为1
30     longest_hash[key] = 1
31     # 遍历上下左右四个点(之前的点)
32     for dx, dy in [(1, 0), (0, -1), (-1, 0), (0, 1)]:
33         x, y = points[i][1] + dx, points[i][2] + dy
34         # 如果点出界, 跳过
35         if x < 0 or x >= n or y < 0 or y >= m:
36             continue
37         # 方程 (Function): max{上下左右LCIS + 1}
38         # 这里(x, y) in longest_hash有必要吗?
39         # 没有必要, 符合A[x][y] < points[i][0]的点一定在longest_hash里面
40         # 如果周边点小于当前点, 那么longest_hash[(x, y)] + 1为CIS
41         # 在所有的CIS中选择LCIS
42         if (x, y) in longest_hash and A[x][y] < points[i][0]:
43             longest_hash[key] = max(longest_hash[key], longest_hash[(x, y)] + 1)
44
45     # 答案 (Answer): max{ 所有坐标对应的LCIS长度 }
46     # 最长子序列可能以任意一点为终点,
47     # 在所有Increasing Subsequence中寻找最大值
48     return max(longest_hash.values())
49
```

603 Largest Divisible Subset 最大整除子集

Given a set of distinct positive integers, find the largest subset which has the most elements, and every pair of two elements (S_i, S_j) in this subset satisfies: $S_i \% S_j = 0$ or $S_j \% S_i = 0$.

给一个由 无重复的正数组成的集合，找出一个元素最多的子集，满足集合中任意两个元素 (S_i, S_j) 都有 $S_i \% S_j = 0$ 或 $S_j \% S_i = 0$

输入:

nums = [1,2,3]

输出:

[1,2] or [1,3]

输入:

nums = [1,2,4,8]

输出:

[1,2,4,8]

一手微信study322, 九章来offer都有

具有有序性，求最优解，有可能使用动态规划

坐标型动态规划

状态的转移是考虑是从哪儿走到 (i, j) 这个坐标的

思路分析 & 代码解析

数值	1	2	3	6	8	12
dp	1	2	2	3	3	4
prev	-1	1	1	2	2	6



$$dp[2] = \max(dp[1]) + 1 = 1 + 1 = 2$$

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

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

$$dp[8] = \max(dp[1], dp[2]) + 1 = 2 + 1 = 3$$

$$dp[12] = \max(dp[1], dp[2], dp[3], dp[6]) + 1 = 3 + 1 = 4$$

```

6  def largestDivisibleSubset(self, nums):
7      # 特殊情况处理
8      if not nums:
9          return []
10
11     # 所有数字从小到大排序
12     # 先要找到小的数字的最大整除子集, 才能得到大的数字的最大整除子集
13     nums = sorted(nums)
14     n = len(nums)
15
16     # 状态 (State): dp[i] 表示以第i个数结尾 (最大数) 的最大整除子集个数
17     # 这里用dict表示状态, 因为输入数字是离散的, 比如1, 2, 4, 8
18     # prev来记录 (当前点 => 之前点) 的映射关系, 用于倒推路径
19     dp, prev = {}, {}
20
21     # 初始化 (Initialization): dp[0..n-1] = 1
22     # 所有点的最大整除子集初始长度为1, 仅包含自身
23     # 先导点为-1, 表示未知
24     for num in nums:
25         dp[num] = 1
26         prev[num] = -1
27
28     # 最大整除子集的最后一个数字 (最大数字)
29     last_num = nums[0]
30
31     # 遍历从小到大的每个数字
32     for num in nums:
33         # 可能的优化: 找一个接龙的数的时候, 不是for循环所有比他小的数,
34         # 而是直接for循环他的因子。而获取因子可以用O(sqrt(value))的时间做到。
35         # 如果n (nums个数) 是个特别大的值, 这里是优化
36         # 如果value是个特别大的值, 这里不是优化
37         for factor in self.get_factors(num):
38             # 如果factor没有在dp/nums中, 跳过
39             if factor not in dp:
40                 continue
41             # 方程 (Function): max(dp.get(factor) + 1)
42             # factor为num的因子
43             if dp[num] < dp[factor] + 1:
44                 dp[num] = dp[factor] + 1
45                 prev[num] = factor
46         # 更新最大整除子集的最后一个数字 (最大数字)
47         if dp[num] > dp[last_num]:
48             last_num = num
49
50     # 答案 (Answer):
51     # 通过最大整除子集的最后一个数字 (最大数字) 倒推路径,
52     # 得到Largest Divisible Subset
53     return self.get_path(prev, last_num)

```

```

55     # 倒推求路径
56     def get_path(self, prev, last_num):
57         path = []
58         # 如果lastNum为-1, 没有先导点了, 已经到了路径的最开端
59         while last_num != -1:
60             path.append(last_num)
61             last_num = prev[last_num]
62         # 通过python的切片操作, 翻转得到正序路径
63         return path[::-1]
64
65     # O(sqrt(num)) 求num的所有因子 (不包括自身)
66     def get_factors(self, num):
67         # 不包括自身, 所以1没有因子
68         if num == 1:
69             return []
70         factor = 1
71         factors = []
72         while factor * factor <= num:
73             if num % factor == 0:
74                 factors.append(factor)
75                 # 如果num为4, factor为1, 那么4/1 = 4, 不加入 (不包括自身)
76                 # 如果加入了自身会怎样? 会出现num => dp.get(num) + 1, 平白
77                 # 无故多了一个数, 答案错误
78                 # 如果num为4, factor为2, 那么4/2 = 2, 不加入 (会重复)
79                 # 如果加入重复会怎样? 会多循环一次, 对答案正确性无影响
80                 if factor * factor != num and factor != 1:
81                     # 在此使用两个斜线的除号, 结果为整数, 而不是浮点数
82                     # 本题目一个斜线的除号, 也可以通过
83                     factors.append(num // factor)
84             factor += 1
85         return factors

```

思路分析 & 代码解析

数值	1	2	3	6	8	12
dp	1	2	2	3	3	4
prev	-1	1	1	2	2	6



$$dp[2] = \max(dp[1]) + 1 = 1 + 1 = 2$$

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

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

$$dp[8] = \max(dp[1], dp[2]) + 1 = 2 + 1 = 3$$

$$dp[12] = \max(dp[1], dp[2], dp[3], dp[6]) + 1 = 3 + 1 = 4$$

```
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63

public List<Integer> largestDivisibleSubset(int[] nums) {
    // 特殊情况处理
    if (nums == null || nums.length == 0) {
        return new ArrayList();
    }

    // 所有数字从小到大排序
    // 先要找到小的数字的最大整除子集, 才能得到大的数字的最大整除子集
    Arrays.sort(nums);
    int n = nums.length;

    // 状态 (State) : dp.get(i) 表示以第i个数结尾 (最大数) 的最大整除子集个数
    // 这里用HashMap表示状态, 因为输入数字是离散的, 比如1, 2, 4, 8
    // prev来记录 (当前点 => 前点) 的映射关系, 用于倒推路径
    HashMap<Integer, Integer> dp = new HashMap();
    HashMap<Integer, Integer> prev = new HashMap();

    // 初始化 (Initialization) : dp[0..n-1] = 1
    // 所有点的最大整除子集初始长度为1, 仅包含自身
    // 先导点为-1, 表示未知
    for (int i = 0; i < n; i++) {
        dp.put(nums[i], 1);
        prev.put(nums[i], -1);
    }

    // 最大整除子集的最后一个数字 (最大数字)
    int lastNum = nums[0];

    // 遍历从小到大的每个数字
    for (int num : nums) {
        // 可能的优化: 找一个接龙的数的时候, 不是for循环所有比他小的数,
        // 而是直接for循环他的因子。而获取因子可以用O(√num)的时间做到。
        // 如果n (nums个数) 是个特别大的值, 这里是优化
        // 如果value是个特别大的值, 这里不是优化
        for (Integer factor : getFactors(num)) {
            // 如果factor没有在dp/nums中, 跳过
            if (!dp.containsKey(factor)) {
                continue;
            }
            // 方程 (Function): max{dp.get(factor) + 1}
            // factor为num的因子
            if (dp.get(num) < dp.get(factor) + 1) {
                dp.put(num, dp.get(factor) + 1);
                // 更新prev, num => factor (当前点 => 前点)
                prev.put(num, factor);
            }
        }
        // 更新最大整除子集的最后一个数字 (最大数字)
        if (dp.get(num) > dp.get(lastNum)) {
            lastNum = num;
        }
    }

    // 答案 (Answer) :
    // 通过最大整除子集的最后一个数字 (最大数字) 倒推路径,
    // 得到Largest Divisible Subset
    return getPath(prev, lastNum);
}
```

```
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101

// 倒推求路径
private List<Integer> getPath(HashMap<Integer, Integer> prev, int lastNum) {
    List<Integer> path = new ArrayList();
    // 如果lastNum为-1, 则没有先导点了, 已经到了路径的最开端
    while (lastNum != -1) {
        path.add(lastNum);
        lastNum = prev.get(lastNum);
    }
    // 翻转得到正序路径
    Collections.reverse(path);
    return path;
}

// O(√num)求num的所有因子 (不包括自身)
private List<Integer> getFactors(int num) {
    List<Integer> factors = new ArrayList();
    // 不包括自身, 所以1没有因子
    if (num == 1) {
        return factors;
    }
    int factor = 1;
    while (factor * factor <= num) {
        if (num % factor == 0) {
            factors.add(factor);
            // 如果num为4, factor为1, 那么4/1 = 4, 不加入 (不包括自身)
            // 如果加入了自身会怎样? 会出现num => dp.get(num) + 1, 平白
            // 无故多了一个数, 答案错误
            // 如果num为4, factor为2, 那么4/2 = 2, 不加入 (会重复)
            // 如果加入重复会怎样? 会多循环一次, 对答案正确性无影响
            if (factor != 1 && factor * factor != num) {
                factors.add(num / factor);
            }
        }
        factor++;
    }
    return factors;
}
```

谢谢大家，夏天老师会想念你们的 $o(\pi_{{\sim}}\pi)o$

离别时
依依不舍的眼神



你要习惯相遇与离别



手机微信st...322 九章来offer都



谢谢你这么长时间
对我的照顾



有缘江湖再见,告辞

[鸡血视频](#)

Thanks♪(・ω・)ﾉ



同学们，下次再见👋！记得课后复习，课前预习！😊

做作业

做ladder

群里提问题

看回放



课前预习

课后复习

刷题

互动课