最难的算法——动态规划

主讲人 令狐冲
课程版本 v7.0

为什么难？

是一种哲学思想，而不是很具体的算法

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

我可以放弃么？

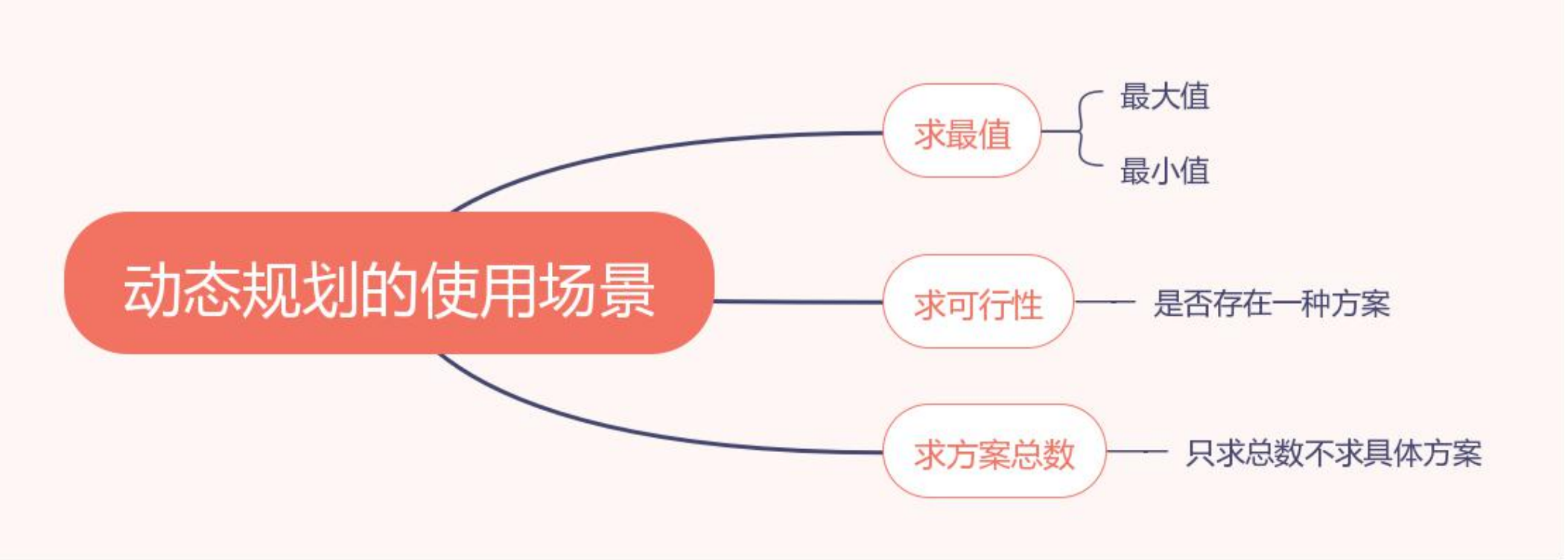
北美求职者：如果你不想去 Google，可以

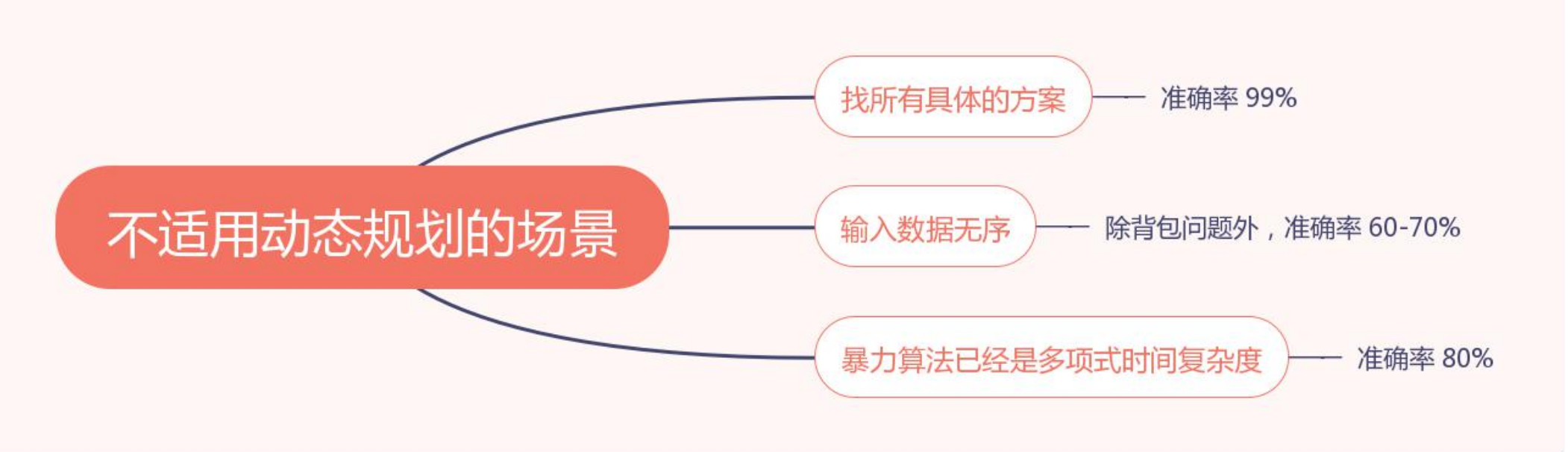
国内求职者：如果你不去大厂，可以

（国内大厂面试题超过 50% 的题都是动态规划）

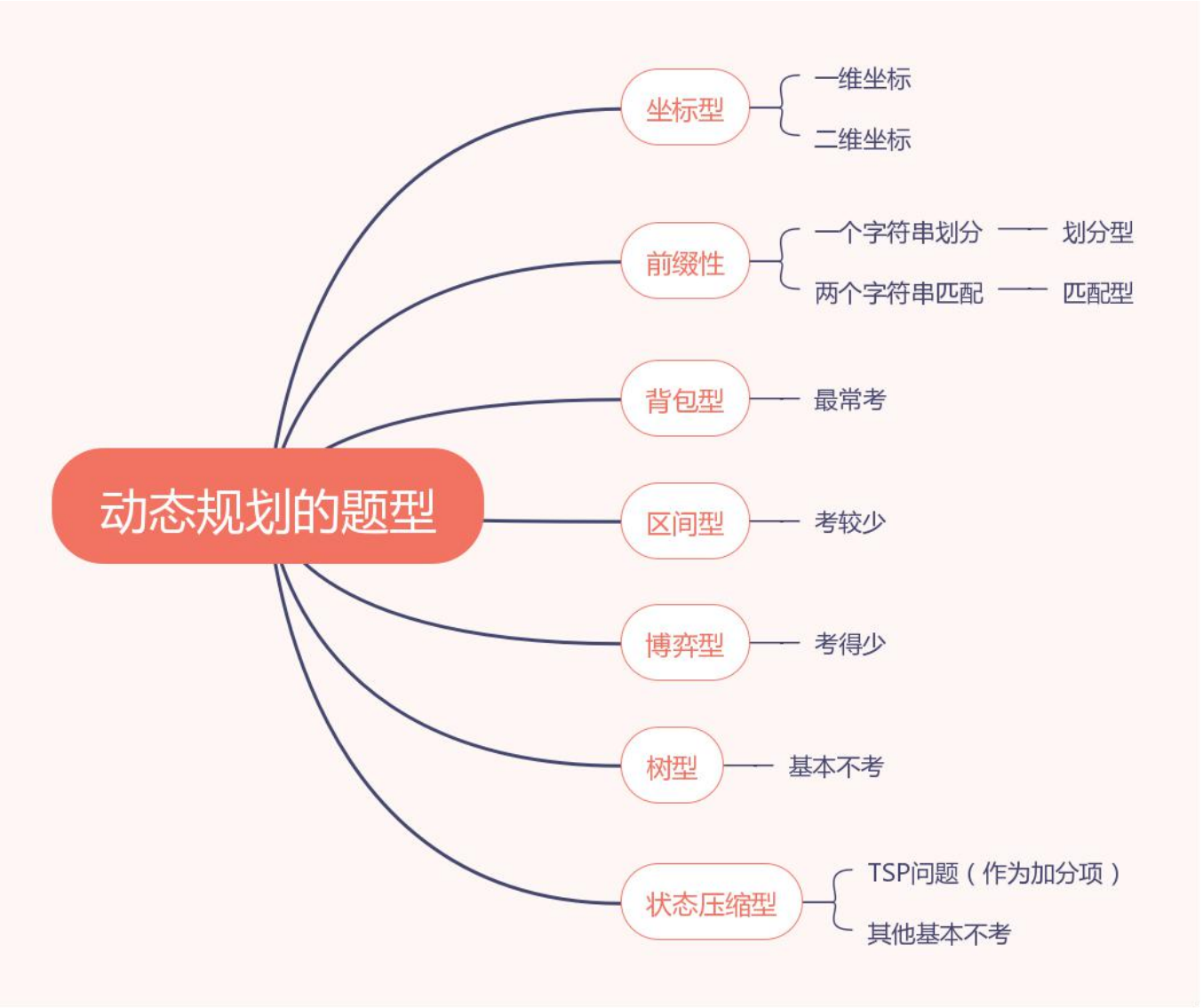
动态规划的解题步骤

- 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

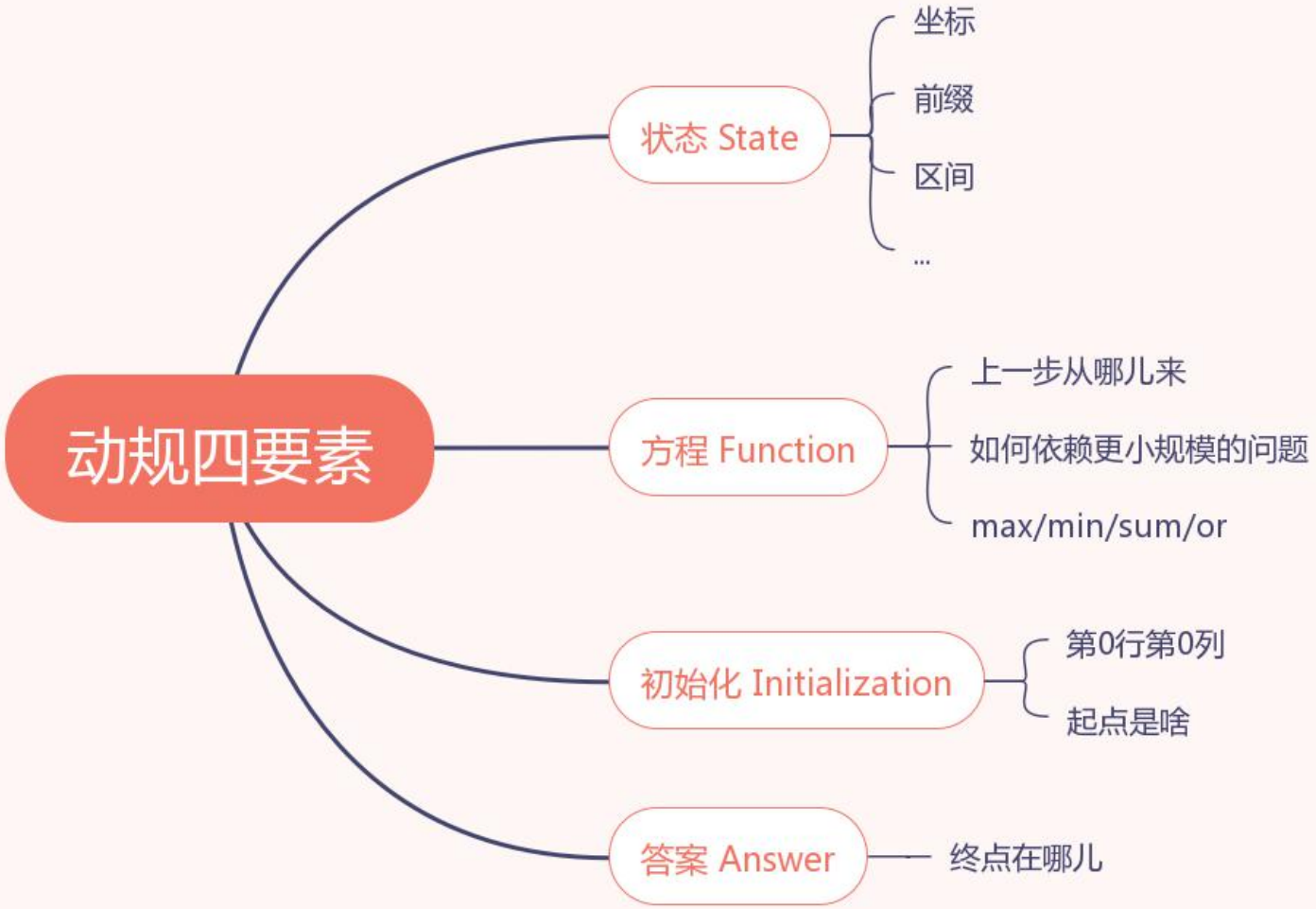




动态规划题型分类



动规四要素



动态规划的时间复杂度

O(状态总数 * 每个状态的处理耗费)

=

O(状态总数 * 决策数)

例子

dp[i][j] = max(dp[i][k], dp[k+1][j]) + cost

O(i的范围*j的范围*k的范围)

dp[i] = or{dp[j] and j+1到 i 是一个单词}

O(i的范围*j的范围) = O(n * maxWordLen)

sum(a,b,c) = a + b +c

max(a,b,c) = a,b,c 最大值

or(a,b,c) = a or b or c

动态规划的空间优化技巧
——滚动数组

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

坐标型动态规划使用滚动数组

数字三角形的状态转移方程为

dp[i][j] = min(dp[i - 1][j], dp[i - 1][j - 1]) + A[i][j]

滚动数组优化之后为

dp[i % 2][j] = min(dp[(i - 1) % 2][j], dp[(i - 1) % 2][j - 1]) + A[i][j]

数字三角形的滚动数组解法 - python


```python
def minimumTotal(self, triangle):
    n = len(triangle)

    # state: dp[i][j] 代表从 0, 0 走到 i, j 的最短路径值
    dp = [[0] * n, [0] * n]

    # initialize: 初始化起点
    dp[0][0] = triangle[0][0]

    # function: dp[i][j] = min(dp[i - 1][j - 1], dp[i - 1][j]) + triangle[i][j]
    # i, j 这个位置是从位置 i - 1, j 或者 i - 1, j - 1 走过来的
    for i in range(1, n):
        dp[i % 2][0] = dp[(i - 1) % 2][0] + triangle[i][0]
        dp[i % 2][i] = dp[(i - 1) % 2][i - 1] + triangle[i][i]
        for j in range(1, i):
            dp[i % 2][j] = min(dp[(i - 1) % 2][j - 1], dp[(i - 1) % 2][j]) + triangle[i][j]

    # answer: 最后一层的任意位置都可以是路径的终点
    return min(dp[(n - 1) % 2])
```

数字三角形滚动数组解法 - java


```java
public int minimumTotal(int[][] triangle) {
    if (triangle == null || triangle.length == 0) {
        return -1;
    }
    if (triangle[0] == null || triangle[0].length == 0) {
        return -1;
    }

    // state: dp[x][y] = minimum path value from 0,0 to x,y
    int n = triangle.length;
    int[][] dp = new int[2][n];

    // initialize
    dp[0][0] = triangle[0][0];

    // top down
    for (int i = 1; i < n; i++) {
        dp[i % 2][0] = dp[(i - 1) % 2][0] + triangle[i][0];
        dp[i % 2][i] = dp[(i - 1) % 2][i - 1] + triangle[i][i];
        for (int j = 1; j < i; j++) {
            dp[i % 2][j] = Math.min(dp[(i - 1) % 2][j - 1], dp[(i - 1) % 2][j]) + triangle[i][j];
        }
    }

    // answer
    int best = dp[(n - 1) % 2][0];
    for (int i = 1; i < n; i++) {
        best = Math.min(best, dp[(n - 1) % 2][i]);
    }
    return best;
}
```

能否两个维度一起滚动呢？

dp[i][j] = min(dp[i - 1][j], dp[i - 1][j - 1]) + A[i][j]

=>

dp[i % 2][j % 2] = min(dp[(i - 1) % 2][j % 2], dp[(i - 1) % 2][(j - 1) % 2]) + A[i][j]

能否两个维度一起滚动呢？

dp[i][j] = min(dp[i - 1][j], dp[i - 1][j - 1]) + A[i][j]

=>

dp[i % 2][j % 2] = min(dp[(i - 1) % 2][j % 2], dp[(i - 1) % 2][(j - 1) % 2]) + A[i][j]

不可以！

fibonacci数列的滚动数组优化

https://www.lintcode.com/problem/fibonacci/

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

滚动数组优化后？

fibonacci数列的滚动数组优化

https://www.lintcode.com/problem/fibonacci/

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

滚动数组优化后？

dp[i % 3] = dp[(i - 1) % 3] + dp[(i - 2) % 3]

代码


```java
public int fibonacci(int n) {
    if (n <= 2) {
        return n - 1;
    }

    int[] dp = new int[3];

    // initialize
    dp[1 % 3] = 0;
    dp[2 % 3] = 1;

    for (int i = 3; i <= n; i++) {
        dp[i % 3] = dp[(i - 1) % 3] + dp[(i - 2) % 3];
    }
    return dp[n % 3];
}
```

```python
def fibonacci(self, n):
    if n <= 2:
        return n - 1

    dp = [0] * 3

    # initialize
    dp[1 % 3] = 0
    dp[2 % 3] = 1

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

    return dp[n % 3]
```

骑士最短路径的滚动数组优化

https://www.lintcode.com/problem/knight-shortest-path-ii/

dp[i][j] = min{dp[i - 1][j - 2], dp[i + 1][j - 2], dp[i - 2][j - 1], dp[i + 2][j - 1]}

滚动数组优化后为？

骑士最短路径的滚动数组优化

https://www.lintcode.com/problem/knight-shortest-path-ii/

dp[i][j] = min{dp[i - 1][j - 2], dp[i + 1][j - 2], dp[i - 2][j - 1], dp[i + 2][j - 1]}

滚动数组优化后为？

dp[i][j] = min{dp[i - 1][(j - 2) % 3], dp[i + 1][(j - 2) % 3], dp[i - 2][(j - 1) % 3], dp[i + 2][(j - 1) % 3]}

代码


```python
def shortestPath2(self, grid):
    if not grid or not grid[0]:
        return -1

    n, m = len(grid), len(grid[0])

    # state: dp[i][j % 3] 代表从 0,0 跳到 i,j 的最少步数
    dp = [[float('inf')] * 3 for _ in range(n)]

    # initialize: 0,0 是起点
    dp[0][0] = 0

    # function
    for j in range(1, m):
        for i in range(n):
            dp[i][j % 3] = float('inf')
            if grid[i][j]:
                continue
            for delta_x, delta_y in DIRECTIONS:
                x, y = i + delta_x, j + delta_y
                if 0 <= x < n and 0 <= y < m:
                    dp[i][j % 3] = min(dp[i][j % 3], dp[x][y % 3] + 1)

    # answer
    if dp[n - 1][(m - 1) % 3] == float('inf'):
        return -1
    return dp[n - 1][(m - 1) % 3]
```

```java
public int shortestPath2(boolean[][] grid) {
    if (grid == null || grid.length == 0) {
        return -1;
    }
    if (grid[0] == null || grid[0].length == 0) {
        return -1;
    }

    int n = grid.length, m = grid[0].length;

    // state: dp[i][j % 3] 代表从起点走到 i,j 的最短路径长度
    int[][] dp = new int[n][3];

    // initialize
    for (int i = 1; i < n; i++) {
        dp[i][0] = Integer.MAX_VALUE;
    }
    dp[0][0] = 0;

    // function: dp[i][j % 3] = min(dp[x][y % 3]) + 1, (x,y) 是 i,j 的上一步
    for (int j = 1; j < m; j++) {
        for (int i = 0; i < n; i++) {
            dp[i][j % 3] = Integer.MAX_VALUE;
            if (grid[i][j]) {
                continue;
            }
            for (int direction = 0; direction < 4; direction++) {
                int x = i + deltaX[direction];
                int y = j + deltaY[direction];
                if (x < 0 || x >= n || y < 0 || y >= m) {
                    continue;
                }
                if (dp[x][y % 3] == Integer.MAX_VALUE) {
                    continue;
                }

                dp[i][j % 3] = Math.min(dp[i][j % 3], dp[x][y % 3] + 1);
            }
        }
    }

    // answer: dp[n - 1][(m - 1) % 3]
    if (dp[n - 1][(m - 1) % 3] == Integer.MAX_VALUE) {
        return -1;
    }
    return dp[n - 1][(m - 1) % 3];
}
```

这个方程可以滚动吗？

Longest Increasing Subsequence:

dp[i] = max{dp[j] + 1}, 0<=j<i, nums[j] < nums[i]

01背包的滚动数组优化

dp[i][j] 代表前 i 个物品取出若干装满大小 j 的背包最多装多少

dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - A[i - 1]] + A[i - 1])

优化之后

dp[i % 2][j] = max(dp[(i - 1) % 2][j], dp[(i - 1) % 2][j - A[i - 1]] + A[i - 1])

更极端的写法

- 由于 dp[i][j] 依赖的是 dp[i - 1][j] 和 dp[i - 1][j - A[i]]
- 每个 i 和 j 都依赖更小的 i 和 j
- 因此只需要保证 j 的循环是从大到小枚举
- 就可以使用一维数组来完成 dp

```
for (int i = 0; i < n; i++) {
    // 滚动数组优化 倒序枚举j
    for (int j = m; j >= A[i]; j--) {
        dp[j] = Integer.max(dp[j], dp[j - A[i]] + A[i]);
    }
}
```

```
for i in range(len(A)):
    # 滚动数组优化 倒序枚举j
    for j in range(m, A[i] - 1, -1):
        dp[j] = max(dp[j], dp[j - A[i]] + A[i])
```

- 该方法：不推荐
- 理由：不通用 + 优化效果和取余的形式无差

滚动数组小结

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

滚动数组也只能滚一个维度

不能两个维度一起滚动

休息一下

Take a break

接龙型动态规划

属于“坐标型”动态规划的一种

题型一般是告诉你一个接龙规则，让你找最长的龙

Longest Increasing Subsequence

http://www.lintcode.com/problem/longest-increasing-subsequence/

http://www.jiuzhang.com/solutions/longest-increasing-subsequence/

接龙规则：从左到右一个比一个大

该问题简称 LIS

状态表示

A: dp[i] 表示前i个数的 LIS 是多长

B: dp[i] 表示以第i个数结尾的 LIS 是多长

哪个是对的？

LIS 的动规四要素

state: dp[i] 表示以第 i 个数为龙尾的最长的龙有多长

function: dp[i] = max{dp[j] + 1}, j < i && nums[j] < nums[i]

initialization: dp[0..n-1] = 1

answer: max{dp[0..n-1]}

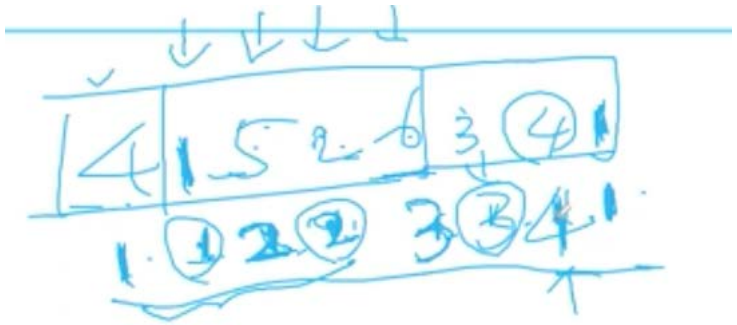
代码


```python
def longestIncreasingSubsequence(self, nums):
    if nums is None or not nums:
        return 0

    # state: dp[i] 表示以第 i 个数结尾的 LIS 的长度
    # initialization: dp[0..n-1] = 1
    dp = [1] * len(nums)

    # function: dp[i] = max(dp[j] + 1), j < i && nums[j] < nums[i]
    for i in range(len(nums)):
        for j in range(i):
            if nums[j] < nums[i]:
                dp[i] = max(dp[i], dp[j] + 1)

    # answer, 任意一个位置都可能是 LIS 的结尾
    return max(dp)
```



```java
public int longestIncreasingSubsequence(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }

    int n = nums.length;

    // state
    int[] dp = new int[n];

    // initialization
    for (int i = 0; i < n; i++) {
        dp[i] = 1;
    }

    // function
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[j] < nums[i]) {
                dp[i] = Math.max(dp[i], dp[j] + 1);
            }
        }
    }

    // answer
    int max = 0;
    for (int i = 0; i < n; i++) {
        max = Math.max(max, dp[i]);
    }
    return max;
}
```

如何获得最长的龙的具体方案？

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

代码

- 改动要点
  1. prev 数组记录前继最优状态
  2. max() 的写法要改为 if 的写法
  3. 找到最长龙的结尾，从结尾倒推出整条龙

index	0	1	2	3	4	5	6	7
nums	4	5	6	1	2	3	7	0
dp	1	2	3	1	2	3	4	1
prev	-1	0	1	-1	3	4	2	-1

```python
def longestIncreasingSubsequence(self, nums):
    if nums is None or not nums:
        return 0

    # state: dp[i] 表示从左到右跳到i的最长sequence 的长度
    # initialization: dp[0..n-1] = 1
    dp = [1] * len(nums)

    # prev[i] 代表 dp[i] 的最优值是从哪个 dp[j] 算过来的
    prev = [-1] * len(nums)

    # function dp[i] = max{dp[j] + 1},  j < i and nums[j] < nums[i]
    for i in range(len(nums)):
        for j in range(i):
            if nums[j] < nums[i] and dp[i] < dp[j] + 1:
                dp[i] = dp[j] + 1
                prev[i] = j

    # answer: max(dp[0..n-1])
    longest, last = 0, -1
    for i in range(len(nums)):
        if dp[i] > longest:
            longest = dp[i]
            last = i

    path = []
    while last != -1:
        path.append(nums[last])
        last = prev[last]
    print(path[::-1])

    return longest
```

代码 - java


```java
public int longestIncreasingSubsequence(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }

    int n = nums.length;

    // state
    // dp[i] 表示从左到右跳到 i 的最长 sequence 的长度
    // prev[i] 代表 dp[i] 的最优值是从哪个 dp[j] 算过来的
    int[] dp = new int[n];
    int[] prev = new int[n];

    // initialization: dp[0..n-1] = 1
    for (int i = 0; i < n; i++) {
        dp[i] = 1;
        prev[i] = -1;
    }

    // function dp[i] = max{dp[j] + 1}, j < i and nums[j] < nums[i]
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[j] < nums[i] && dp[j] + 1 > dp[i]) {
                dp[i] = dp[j] + 1;
                prev[i] = j;
            }
        }
    }

    // answer: max(dp[0..n-1])
    int longest = 0, last = -1;
    for (int i = 0; i < n; i++) {
        if (dp[i] > longest) {
            longest = dp[i];
            last = i;
        }
    }

    // print solution
    ArrayList<Integer> path = new ArrayList();
    while (last != -1) {
        path.add(nums[last]);
        last = prev[last];
    }
    for (int i = path.size() - 1; i >= 0; i--) {
        System.out.print(path.get(i) + "-");
    }
    return longest;
}
```

Longest Continuous Increasing Subsequence II

https://www.lintcode.com/problem/longest-continuous-increasing-subsequence-ii/

https://www.jiuzhang.com/solution/longest-continuous-increasing-subsequence-ii/

四个方向，依然可以动态规划

代码 - python


```python
def longestContinuousIncreasingSubsequence2(self, A):
    if not A or not A[0]:
        return 0

    n, m = len(A), len(A[0])
    points = []
    for i in range(n):
        for j in range(m):
            points.append((A[i][j], i, j))

    points.sort()

    longest_hash = {}
    for i in range(len(points)):
        key = (points[i][1], points[i][2])
        longest_hash[key] = 1
        for dx, dy in [(1, 0), (0, -1), (-1, 0), (0, 1)]:
            x, y = points[i][1] + dx, points[i][2] + dy
            if x < 0 or x >= n or y < 0 or y >= m:
                continue
            if (x, y) in longest_hash and A[x][y] < points[i][0]:
                longest_hash[key] = max(longest_hash[key], longest_hash[(x, y)]
                    + 1)

    return max(longest_hash.values())
```

代码 - java


```java
public int longestContinuousIncreasingSubsequence2(int[][] matrix) {
    if (matrix == null || matrix.length == 0) {
        return 0;
    }

    if (matrix[0] == null || matrix[0].length == 0) {
        return 0;
    }

    int n = matrix.length, m = matrix[0].length;
    int[] dx = {0, 0, 1, -1};
    int[] dy = {1, -1, 0, 0};

    List<List<Integer>> points = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            points.add(Arrays.asList(i, j, matrix[i][j]));
        }
    }
    points.sort((p1, p2) -> Integer.compare(p1.get(2), p2.get(2)));

    // State: dp[i][j] 表示以坐标(i,j)的点matrix[i][j]为龙尾的最长龙的长度
    int[][] dp = new int[n][m];

    // Function
    for (int i = 0; i < points.size(); i++) {
        int x = points.get(i).get(0);
        int y = points.get(i).get(1);

        dp[x][y] = 1;  // Initialization

        for (int j = 0; j < 4; j++) {
            int prevX = x - dx[j];
            int prevY = y - dy[j];

            if (prevX < 0 || prevX >= n || prevY < 0 || prevY >= m) {
                continue;
            }
            if (matrix[prevX][prevY] >= matrix[x][y]) {
                continue;
            }

            dp[x][y] = Math.max(dp[x][y], dp[prevX][prevY] + 1);
        }
    }

    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;
}
```

Largest Divisible Subset

http://www.lintcode.com/en/problem/largest-divisible-subset/

http://www.jiuzhang.com/solutions/largest-divisible-subset/

接龙规则：后面的数可以整除前面的数

代码 - python


```python
def largestDivisibleSubset(self, nums):
    if not nums:
        return []

    nums = sorted(nums)
    n = len(nums)
    dp, prev = {}, {}
    for num in nums:
        dp[num] = 1
        prev[num] = -1

    last_num = nums[0]
    for num in nums:
        for factor in self.get_factors(num):
            if factor not in dp:
                continue
            if dp[num] < dp[factor] + 1:
                dp[num] = dp[factor] + 1
                prev[num] = factor
        if dp[num] > dp[last_num]:
            last_num = num

    return self.get_path(prev, last_num)
```

```python
def get_path(self, prev, last_num):
    path = []
    while last_num != -1:
        path.append(last_num)
        last_num = prev[last_num]
    return path[::-1]

def get_factors(self, num):
    if num == 1:
        return []
    factor = 1
    factors = []
    while factor * factor <= num:
        if num % factor == 0:
            factors.append(factor)
            if factor * factor != num and factor != 1:
                factors.append(num // factor)
        factor += 1
    return factors
```

代码 - java


```java
public List<Integer> largestDivisibleSubset(int[] nums) {
    if (nums == null || nums.length == 0) {
        return new ArrayList();
    }

    Arrays.sort(nums);
    int n = nums.length;
    HashMap<Integer, Integer> dp = new HashMap();
    HashMap<Integer, Integer> prev = new HashMap();

    for (int i = 0; i < n; i++) {
        dp.put(nums[i], 1);
        prev.put(nums[i], -1);
    }

    int lastNum = nums[0];
    for (int i = 0; i < n; i++) {
        int num = nums[i];
        for (Integer factor : getFactors(num)) {
            if (!dp.containsKey(factor)) {
                continue;
            }
            if (dp.get(num) < dp.get(factor) + 1) {
                dp.put(num, dp.get(factor) + 1);
                prev.put(num, factor);
            }
        }
        if (dp.get(num) > dp.get(lastNum)) {
            lastNum = num;
        }
    }

    return getPath(prev, lastNum);
}

private List<Integer> getPath(HashMap<Integer, Integer> prev, int lastNum) {
    List<Integer> path = new ArrayList();
    while (lastNum != -1) {
        path.add(lastNum);
        lastNum = prev.get(lastNum);
    }
    Collections.reverse(path);
    return path;
}

private List<Integer> getFactors(int num) {
    List<Integer> factors = new ArrayList();
    if (num == 1) {
        return factors;
    }
    int factor = 1;
    while (factor * factor <= num) {
        if (num % factor == 0) {
            factors.add(factor);
            if (factor != 1 && num / factor != factor) {
                factors.add(num / factor);
            }
        }
        factor++;
    }
    return factors;
}
```

单词接龙问题

给一个单词集合，两个单词可以接在一起当且仅当前一个单词的后缀和后一个单词的前缀，能够重叠（至少1个字符），如

World + dog = Worldog

Hello + low = Hellow

may + maybe = maybe

每个单词可以且仅可以使用一次，请问最长接出来的龙的长度多少？

这个题是否可以使用动规规划？

回顾一下经典算法的简称

LIS, LCS, LCA, TSP, MST