

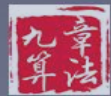
# Dynamic Programming I

课程尚未开始, 请大家耐心等待



关注微信  
ninechapter

获得最新面试  
题、面经、题解



# Outline

从递归到动规 - Triangle

什么样的题适合使用动态规划？

当我们谈论动态规划的时候，我们在谈论什么？

面试中常见动态规划的分类

矩阵动态规划

单序列动态规划(上)



# Triangle

<http://www.lintcode.com/problem/triangle/>  
<http://www.jiuzhang.com/solutions/triangle/>



# DFS - Traverse

```
1- void traverse(int x, int y, int sum) {  
2-     if (x == n) {  
3-         // found a whole path from top to bottom  
4-         if (sum < best) {  
5-             best = sum;  
6-         }  
7-         return;  
8-     }  
9-  
10    traverse(x + 1, y, sum + A[x][y]);  
11    traverse(x + 1, y + 1, sum + A[x][y]);  
12 }  
13  
14 best = MAXINT;  
15 traverse(0, 0, 0);  
16 // best is the answer
```

时间复杂度？

A -  $O(n^2)$

B -  $O(2^n)$

C -  $O(n!)$

D - I don't know



# DFS - Divide & Conquer

```
1 // return the minimum path from (x, y) to bottom
2 int divideConquer(int x, int y) {
3     if (x == n) {
4         return 0;
5     }
6
7     return (A[x][y]
8         + divideConquer(x + 1, y)
9         + divideConquer(x + 1, y + 1));
10 }
11
12 // divideConquer(0, 0) is the answer.
```

时间复杂度？

A -  $O(n^2)$

B -  $O(2^n)$

C -  $O(n!)$

D - I don't know



# Divide & Conquer + Memorization

```
1 // return the minimum path from (x, y) to bottom
2 int divideConquer(int x, int y) {
3     if (x == n) {
4         return 0;
5     }
6
7     if (hash[x][y] != -1) {
8         return hash[x][y];
9     }
10
11     hash[x][y] = (A[x][y] + divideConquer(x + 1, y)
12                 + divideConquer(x + 1, y + 1));
13     return hash[x][y];
14 }
15
16 // hash[*][*] = -1
17 // divideConquer(0, 0) is the answer.
```

时间复杂度？

A -  $O(n^2)$

B -  $O(2^n)$

C -  $O(n!)$

D - I don't know



# 记忆化搜索

本质上:动态规划

动态规划就是 \*解决了重复计算\* 的搜索

动态规划的实现方式:

1. 记忆化搜索
2. 循环



# 自底向上的动态规划

```
A[][]  
  
// 状态定义  
f[i][j] 表示从i,j出发走到最后一层的最小路径长度  
  
// 初始化, 终点先有值  
for (int i = 0; i < n; i++) {  
    f[n-1][i] = A[n-1][i];  
}  
  
// 循环递推求解  
for (int i = n - 2; i >= 0; i--) {  
    for (int j = 0; j <= i; j++) {  
        f[i][j] = Math.min(f[i + 1][j], f[i + 1][j + 1]) + A[i][j];  
    }  
}  
  
// 求结果: 起点  
f[0][0]
```

复杂度？





# 自顶向下的动态规划

// 自顶向下的动态规划

状态定义:

$f[i][j]$  表示从0,0出发, 到达 $i, j$ 的最短路径是什么

// 初始化

$f[0][0] = A[0][0]$

// 递推求解

```
for (int i = 1; i < n; i++) {  
    for (int j = 1; j <= i; j++) {  
        //  $f[i][j] = \min(f[i-1][j], f[i-1][j-1]) + A[i][j];$   
         $f[i][j] = \infty;$   
        if (i-1, j存在) {  
             $f[i][j] = \min(f[i][j], f[i-1][j]);$   
        }  
        if (i-1, j-1存在) {  
            ..  
        }  
         $f[i][j] += A[i][j];$   
    }  
}
```

// 答案

$\min(f[n-1][0], f[n-1][1], f[n-1][2] \dots)$

复杂度?



# 循环求解 vs 记忆化搜索

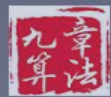
循环求解更\*正规\*, 存在空间优化的可能, 大多数面试官  
可以接受

记忆化搜索空间耗费更大, 但思维难度小, 编程难度小,  
时间效率很多情况下更高, 少数有\*水平\*的面试官可以接  
受



# 自顶向下 vs 自底向上

两种方法没有太大优劣区别  
思维模式一个正向，一个逆向  
为了方便教学，后面我们统一采用\*自顶向下\*的方式



# 什么情况下可能是动态规划？

满足下面三个条件之一

1. Maximum/Minimum
2. Yes/No
3. Count(\*)

则“极有可能”是使用动态规划求解



# 什么情况下可能不是动态规划？

如果题目需要求出所有“具体”的方案而非方案“个数”

<http://www.lintcode.com/problem/palindrome-partitioning/>

输入数据是一个“集合”而不是“序列”

<http://www.lintcode.com/problem/longest-consecutive-sequence/>



# 动态规划的4点要素

## 1. 状态 State

灵感, 创造力, 存储小规模问题的结果

## 2. 方程 Function

状态之间的联系, 怎么通过小的状态, 来算大的状态

## 3. 初始化 Intialization

最极限的小状态是什么, 起点

## 4. 答案 Answer

最大的那个状态是什么, 终点



# 面试最常见的四种类型

1. Matrix DP (15%)
2. Sequence (40%)
3. Two Sequences DP (40%)
- \*4. Others (5%)



# 10 minutes break





# 1. Matrix DP

state:  $f[x][y]$  表示我从起点走到坐标 $x,y$ .....

function: 研究走到 $x,y$ 这个点之前的一步

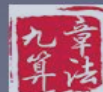
intialize: 起点

answer: 终点



# Minimum Path Sum

<http://www.lintcode.com/problem/minimum-path-sum/>  
<http://www.jiuzhang.com/solutions/minimum-path-sum/>



# Minimum Path Sum

state:  $f[x][y]$  从起点走到  $x, y$  的最短路径

function:  $f[x][y] = \min(f[x-1][y], f[x][y-1]) + A[x][y]$

intialize:  $f[0][0] = A[0][0]$

//  $f[i][0] = \text{sum}(0, 0 \rightarrow i, 0)$

//  $f[0][i] = \text{sum}(0, 0 \rightarrow 0, i)$

answer:  $f[n-1][m-1]$



# Unique Paths

<http://www.lintcode.com/problem/unique-paths/>  
<http://www.jiuzhang.com/solutions/unique-paths/>



# Unique Paths

state:  $f[x][y]$  从起点到  $x, y$  的路径数

function: (研究倒数第一步)

$$f[x][y] = f[x - 1][y] + f[x][y - 1]$$

intialize:  $f[0][0] = 1$

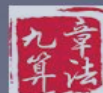
$$// f[0][i] = 1, f[i][0] = 1$$

answer:  $f[n-1][m-1]$



# Unique Paths II

<http://www.lintcode.com/problem/unique-paths-ii/>  
<http://www.jiuzhang.com/solutions/unique-paths-ii/>



## 2. Sequence Dp

state:  $f[i]$ 表示“前 $i$ ”个位置/数字/字母,(以第 $i$ 个为)...

function:  $f[i] = f[j] \dots j$  是 $i$ 之前的一个位置

intialize:  $f[0]..$

answer:  $f[n-1]..$



# Climbing Stairs

<http://www.lintcode.com/problem/climbing-stairs/>  
<http://www.jiuzhang.com/solutions/climbing-stairs/>





# Climbing Stairs

state:  $f[i]$ 表示前 $i$ 个位置, 跳到第 $i$ 个位置的方案总数

function:  $f[i] = f[i-1] + f[i-2]$

intialize:  $f[0] = 1$

answer:  $f[n]$



# Jump Game

<http://www.lintcode.com/problem/jump-game/>  
<http://www.jiuzhang.com/solutions/jump-game/>

note: Dynamic Programming is not the best solution for this problem



# Jump game

state:  $f[i]$ 代表我能否从起点跳到第 $i$ 个位置

function:  $f[i] = \text{OR}(f[j], j < i \ \&\& \ j \text{能够跳到} i)$

initialize:  $f[0] = \text{true};$

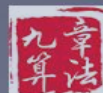
answer:  $f[n-1]$



# Jump Game II

<http://www.lintcode.com/problem/jump-game-ii/>  
<http://www.jiuzhang.com/solutions/jump-game-ii/>

note: Dynamic Programming is not the best solution for this problem



# Jump game II

state:  $f[i]$ 代表我跳到这个位置最少需要几步

function:  $f[i] = \text{MIN}(f[j]+1, j < i \ \&\& \text{j能够跳到i})$

initialize:  $f[0] = 0;$

answer:  $f[n-1]$



# Related Problems

<http://www.lintcode.com/tag/dynamic-programming/>

