Dynamic Programming I

现在是课程答疑时间

扫描二维码关注微信/微博
获取最新面试题及权威解答

微信: ninechapter
微博: http://www.weibo.com/ninechapter
官网: www.jiuzhang.com

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
- 坐标（矩阵）动态规划

Triangle

http://www.lintcode.com/problem/triangle/

http://www.jiuzhang.com/solutions/triangle/

解决方法：

- DFS: Traverse
- DFS: Divide Conquer
- Divide Conquer + Memorization
- Traditional Dynamic Programming

DFS: Traverse

时间复杂度？

- A - O(n^2)
- B - O(2^n)
- C - O(n!)
- D - I don’t know

```
void traverse(int x, int y, int sum) {
    if (x == n) {
        // found a whole path from top to bottom
        if (sum < best) {
            best = sum;
        }
        return;
    }

    traverse(x + 1, y, sum + A[x][y]);
    traverse(x + 1, y + 1, sum + A[x][y]);
}

best = MAXINT;
traverse(0, 0, 0);
// best is the answer
```

DFS: Divide Conquer

时间复杂度？

- A - O(n^2)
- B - O(2^n)
- C - O(n!)
- D - I don’t know

```
// return minimum path from (x, y) to bottom
int divideConquer(int x, int y) {
    // x: 0 .. n-1
    if (x == n) {
        return 0;
    }
    return A[x][y] + Math.min(dfs(x + 1, y),
                              dfs(x + 1, y + 1));
}

// answer
divideConquer(0, 0);
```

DFS: Divide Conquer + Memorization

九章算法

时间复杂度？

- A - O(n^2)
- B - O(2^n)
- C - O(n!)
- D - I don't know

```
// return minimum path from (x, y) to bottom
int divideConquer(int x, int y) {
    if (x == n) {
        return 0;
    }
    // Integer.MAX_VALUE 代表没有访问过x, y这个点
    if (hash[x][y] != Integer.MAX_VALUE) {
        return hash[x][y];
    }
    hash[x][y] = A[x][y] + Math.min(dfs(x + 1, y),
                                    dfs(x + 1, y + 1));
    return hash[x][y];
}

// answer
hash[*][*] = Integer.MAX_VALUE;
divideConquer(0, 0);
```

Memorization Search

记忆化搜索的本质:动态规划

动态规划为什么会快？

动态规划与分治的区别？

重复计算！

九章算法

多重循环 vs 记忆化搜索

优点：
正规，大多数面试官可以接受，存在空间优化可能性。

缺点：
思考有难度。

优点：
容易从搜索算法直接转化过来。有的时候可以节省更多的时间。

缺点：
递归。

多重循环：自底向上

九章算法

时间复杂度？

空间复杂度？

```
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]
```

多重循环：自顶向下

九章算法 九章算法

时间复杂度？

空间复杂度？

```
// 初始化，起点
f[0][0] = A[0][0];

// 初始化三角形的左边和右边
for (int i = 1; i < n; i++) {
    f[i][0] = f[i - 1][0] + A[i][0];
    f[i][i] = f[i - 1][i - 1] + A[i][i];
}

// top down
for (int i = 1; i < n; i++) {
    for (int j = 1; j < i; j++) {
        f[i][j] = Math.min(f[i - 1][j], f[i - 1][j - 1]) + A[i][j];
    }
}

Math.min(f[n - 1][0], f[n - 1][1], f[n - 1][2] ...);
```

自底向上 vs 自顶向下

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**使用**动态规划？

满足下面三个条件之一：

- 求最大值最小值
- 判断是否可行
- 统计方案个数

则 **极有可能** 是使用动态规划求解

什么情况下**不使用**动态规划？

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/

则 **极不可能** 使用动态规划求解

动态规划的四点要素

九章算法

状态 State

灵感，创造力，存储小规模问题的结果

方程 Function

状态之间的联系，怎么通过小的状态，来算大的状态

初始化 Initialization

最极限的小状态是什么，起点

答案 Answer

最大的那个状态是什么，终点

面试中常见的动态规划类型

坐标型动态规划 15%

序列型动态规划 30%

双序列动态规划 30%

划分型动态规划 10%

背包型动态规划 10%

区间型动态规划 5%

Take a break

5 Minutes Break

坐标型动态规划

state:

f[x] 表示我从起点走到坐标x……

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[i][0] = sum(0,0 ~ i,0)

f[0][i] = sum(0,0 ~ 0,i)

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

独孤九剑 之 破枪式

初始化一个二维的动态规划时

就去初始化**第0行**和**第0列**

Unique Path

http://www.lintcode.com/problem/unique-paths/

http://www.jiuzhang.com/solutions/unique-paths/

Unique Path

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][i] = 1

f[i][0] = 1

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

Related Question: Unique Path II

Climing Stairs

http://www.lintcode.com/problem/climbing-stairs/

http://www.jiuzhang.com/solutions/climbing-stairs/

Climing Stairs

state: f[i]表示跳到第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/

Jump Game

最优算法：贪心法，时间复杂度 O(n)

次优算法：动态规划，时间复杂度 O(n^2)

state: f[i]代表我能否跳到第i个位置

function: f[i] = OR(f[j], j < i && j能够跳到i)

initialize: f[0] = true;

answer: f[n-1]

Jump Game II

http://www.lintcode.com/problem/jump-game-ii/

http://www.jiuzhang.com/solutions/jump-game-ii/

Jump Game II

state: f[i]代表我跳到第i个位置最少需要几步

function: f[i] = MIN{f[j]+1, j < i && j能够跳到i}

initialize: f[0] = 0;

answer: f[n-1]

Conclusion

Why DP?

When DP?

How DP?