课程尚未开始
请大家耐心等待

关注微信公共账号，获得最新面试题信息及解答

关注微信
ninechapter
Weibo:
http://www.weibo.com/ninechapter

获得最新面试题、面经、题解

5. Dynamic Programming

高级算法班IT求职面试培训 第5章
www.ninechapter.com

Overview

1. 空间优化
2. 记忆化搜索
3. 循环引用的状态数组
4. 区间动态规划

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

a) 最优解/Maximum/Minimum

b) Yes/No

c) Count(*)

2. 方程 Function

状态之间的联系，怎么通过小的状态，来求得大的状态

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

1.动态规划的空间优化

House Robber

http://www.lintcode.com/en/problem/house-robber/

House Robber

1. 状态 State
   f[i] 表示前i个房子中，偷了第i个房子的，偷到的最大价值
2. 方程 Function
   f[i] = max(f[i-2], f[i-3])+ A[i];
3. 初始化 Intialization
   f[0] = A[0];
   f[1] = Math.max(A[0], A[1]);
   f[2] = Math.max(A[0]+A[2], A[1]);
4. 答案 Answer
   max{f[i]}

一维优化的题目

这类题目特点
f[i] = 由 f[i-2],f[i-3] 来决定状态
可以转化为
f[i%3] =由f[(i-2)%3]和 f[(i-3)%3] 来决定状态
观察我们需要保留的状态来确定模数

比如
Climbing Stairs
Maximum Subarray

Maximal Square

http://www.lintcode.com/en/problem/maximal-square/

Maximal Square

1. 状态 State

   f[i][j] 表示以i和j作为正方形右下角可以拓展的最大边长

2. 方程 Function

```
if matrix[i][j] == 1
    f[i][j] = min(f[i - 1][j], f[i][j-1], f[i-1][j-1]) + 1;
if matrix[i][j] == 0
    f[i][j] = 0
```

3. 初始化 Intialization

```
f[i][0] = matrix[i][0];
f[0][j] = matrix[0][j];
```

4. 答案 Answer

```
max{f[i][j]}
```

Maximal Square

1. 状态 State
   f[i][j] 表示以i和j作为正方形右下角可以拓展的最大边长
2. 方程 Function
   if matrix[i][j] == 1
   f[i%2][j] = min(f[(i - 1)%2][j], f[i%2][j-1], f[(i-1)%2][j-1]) + 1;
   if matrix[i][j] == 0
   f[i%2][j] = 0
3. 初始化 Intialization
   f[i][0] = matrix[i][0];
   f[0][j] = matrix[0][j];
4. 答案 Answer
   max{f[i][j]}

Backpack II

http://www.lintcode.com/en/problem/backpack-ii/

Backpack II

1. 状态 State
   f[i][j] 表示前i个物品当中选一些物品组成容量为j的最大价值
2. 方程 Function
   f[i][j] = max(f[i-1][j], f[i-1][j-A[i-1]] + V[i-1]);
3. 初始化 Intialization
   dp[0][0]=0;
4. 答案 Answer
   dp[n][m]

类似二维动态规划空间优化

这类题目特点
f[i][j] = 由f[i-1]行 或者 f[i][k](k<j) 来决定状态
第i行跟 i-2行之前毫无关系
状态转变为
f[i%2][j] = 由f[(i-1)%2]行 或者 f[i%2][k](k<j) 来决定状态

相关的题目
Unique Paths
Minimum Path Sum
Edit Distance

2.记忆化搜索

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. 循环（从小到大递推）
2. 记忆化搜索(从大到小搜索)

记忆化搜索

```
// 循环求所有状态
For I = 1 ->n
   dp[i] = search (i)

// 搜索
int search(int i)
{
                if(visit[i] == 1)
                                return dp[i];
                if(smallest state)
                {
                                set smallest state
                } else {
                                // to update (i,) , we might need other state
                                // such as (i-1), (i+1)
                                for other state
                                  update dp[i] = max(search(i-1)  , search(i+1))
                }
                visit[i] = 1;
                return  dp[i];
}
```

Longest Increasing Subsequence

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

[4, 2, 5, 4, 3, 9,8,10]

Longest Increasing continuous Subsequence 2D

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

10	2	7
2	3	6
11	4	5

什么时候用记忆化搜索?

1. 状态转移特别麻烦，不是顺序性。

2. 初始化状态不是很容易找到

博弈类
Coins in a line

http://www.lintcode.com/en/problem/coins-in-a-line/

state: f[x] 现在还剩x个硬币，现在先手取硬币的人最后输赢状况

function:

f[n] = (f[n-2]&& f[n-3]) | (f[n-3]&&f[n-4])

intialize:

f[1] = true

f[2] = true

f[3] = false

f[4] = true

f[5] = true

Answer: f[n]

Coins in a Line II

http://www.lintcode.com/en/problem/coins-in-a-line-ii/

[5,1,2,10]

state: f[x] 现在还剩x个硬币，现在先手取硬币的人最后最多取硬币价值

function:

f[n] = max(min(f[n+2], f[n+3])+x[n]) | (min(f[n+3],f[n+4])+x[n]+x[n+1])

intialize:

f[n] = 0

f[n-1] = a[n-1]

f[n-2] = a[n-2]

f[n-3] = a[n-3]+ a[n-2]

Answer:f[n]

Coins in a Line III

http://www.lintcode.com/en/problem/coins-in-a-line-iii/

state: f[x][y] 现在还第x到第y的硬币，现在先手取硬币的人最后最多取硬币价值
function:

f[x][y] = max(min(f[x+2][y], f[x+1][y-1])+a[x]) ,
(min(f[x][y-2], f[x+1][y-1])+a[y])

intialize:

f[x][x] = a[x],
f[x][x+1] = max(a[x],a[x+1]),

Answer:f[n][m]

3.用循环引用的状态数组解决高难度的动态规划问题

Maximum Subarray

http://www.lintcode.com/en/problem/minimum-subarray/

Maximum Subarray

1. 状态 State
   local[i] 表示包括第i个元素能够找到的 最大值
   global[i] 表示全局前i个元素中能够找到的最大值
2. 方程 Function
   local[i] = Max(nums[i],local[i-1]+nums[i]);
   global[i] = Max(local[i],global[i-1]);
3. 初始化 Intialization
   local [0] = global [0] = nums[0];
4. 答案 Answer
   max{global[n-1]}

Maximum Product Subarray

http://www.lintcode.com/en/problem/maximum-product-subarray/

Maximum Product Subarray

1. 状态 State
   min[i] 表示前i个数包括第i个数找到的最小乘积
   max[i] 表示前i个数包括第i个数找到的最大乘积
2. 方程 Function
   min[i] = Min(nums[i], Min(min[i - 1] * nums[i], max[i - 1] * nums[i]));
   max[i] = Max(nums[i], Max(max[i - 1] * nums[i], min[i - 1] * nums[i]));

3. 初始化 Intialization
   min[0] = max[0] = nums[0];
4. 答案 Answer
   max{max[i]}

Best Time to Buy and Sell Stock IV

http://www.lintcode.com/en/problem/best-time-to-buy-and-sell-stock-iv/

Maximum Product Subarray

简单的做法
k transcations
state: f[i][j]表示前i天进行j次交易，能够获得的最大收益
function: f[i][j] = max{f[x][j-1] + profit(x+1, i)}
answer: f[n][k]
intialize: f[i][0] = 0, f[0][i] = -MAXINT (i>0)

Maximum Product Subarray

1. 状态 State
   mustSell[i][j] 表示前i天，至多进行j次交易，第i天必须sell的最大获益
   globalbest[i][j] 表示前i天，至多进行j次交易，第i天可以不sell的最大获益
2. 方程 Function

```
gainorlose = prices[i] - prices[i - 1];
mustsell[i][j] = Max(globalbest[(i - 1)][j - 1] + gainorlose,
                     mustsell[(i - 1)][j] + gainorlose);
globalbest[i][j] = Math.max(globalbest[(i - 1)][j], mustsell[i ][j]);
```

3. 初始化 Intialization

```
mustsell[0][i] = globalbest[0][i] = 0;
```

4. 答案 Answer

```
globalbest[(n - 1)][k]
```

4.区间动态规划

Copy Books

http://www.lintcode.com/en/problem/copy-books/

1. 状态 State

    dp[i][nk] 表示前i本书用nk个人写的最小花费

2. 方程 Function

    dp[i][nk] = max{dp[j][nk-1], w[j+1][i]}

3. 初始化 Intialization

    dp[i][1] = w[1][i];

4. 答案 Answer

    dp[n][k]

Post Office Problem

http://www.lintcode.com/en/problem/post-office-problem/

Post Office Problem

1. 状态 State
   dp[i][nk] 表示前i个房子用nk个邮局的最小花费
2. 方程 Function
   dp[i][nk] = max{dp[j][nk-1] + dis[j+1][i]}
3. 初始化 Intialization
   dp[i][1] = dis[1][i];
4. 答案 Answer
   dp[n][k]

Summary

1. 空间优化
   重点： 滚动数组
2. 记忆化搜索
   重点： 博弈类问题
3. 循环引用的状态数组
   重点： global 和 local 最优
4. 区间动态规划

   重点： 区间的划分