动态规划(上)

九章算法强化班 第5章

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
知乎: http://zhuanlan.zhihu.com/jiuzhang
官网: http://www.jiuzhang.com

Overview

- 滚动数组
    - House Robber I/II

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
     - 最优解/Maximum/Minimum
     - Yes/No
     - 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

滚动数组优化

滚动数组优化

f[i] = max(f[i-1], f[i-2] + A[i]);
转换为
f[i%2] = max(f[(i-1)%2]和 f[(i-2)%2])

House Robber

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

公主追王子
For循环 -----> DP

House Robber

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

House Robber II

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

滚动数组优化一维

- 这类题目特点
  - f[i] = max(f[i-1], f[i-2] + A[i]); 由 f[i-1],f[i-2] 来决定状态
- 可以转化为
  - f[i%2] = max(f[(i-1)%2]和 f[(i-2)%2]) 由f[(i-1)%2]和 f[(i-2)%2] 来决定状态
- 观察我们需要保留的状态来确定模数

- 其他一维滚动数组的题目
  - http://www.lintcode.com/en/problem/climbing-stairs/

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(LEFT[i - 1][j], UP[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]}

二维矩阵小技巧:
用左下角定位

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]}

二维矩阵小技巧：
用左下角定位

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]}

Follow up

01矩阵里面找一个，对角线全为1， 其他为0的矩阵

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[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

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：
  - 循环（从小到大递推）
  - 记忆化搜索(从大到小搜索)
    - 画搜索树
    - 万金油

Longest Increasing Subsequence

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

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

10	2	7
2	3	6
11	4	5

Longest Increasing continuous Subsequence 2D

- 多重循环DP遇到的困难：
  - 从上到下循环不能解决问题
  - 初始状态找不到

- 那我们有没有可以比较暴力解决的方法呢？
  - 有搜索， 我们从大的往小的搜索

Longest Increasing continuous Subsequence 2D

- 普通搜索

```
// 循环求所有状态
For i = 1 -> n
  For j = 1 -> n
   search (i,j) // 直接求i,j最为结尾最长子序列


int search(int x, int y, int[][] A)   {
    for(int i = 0; i < 4; i++) {
        nx = x + dx[i];
        ny = y + dy[i];
        if( A[x][y] > A[nx][ny]) {
            // 通过 search( nx, ny, A) 更新最长子序列
        }
    }
    //返回答案
}
```

- 普通搜索

```
// 循环求所有状态
For i = 1 -> n
  For j = 1 -> n
    search (i,j) // 直接求i,j最为结尾最长子序列


int search(int x, int y, int[][] A)   {
    for(int i = 0; i < 4; i++) {
        nx = x + dx[i];
        ny = y + dy[i];
        if( A[x][y] > A[nx][ny]) {
            // 通过 search( nx, ny, A) 更新最长子序列
        }
    }
    //返回答案
}
```

- 记忆化搜索

```
// 循环求所有状态
For i = 1 -> n
  For j = 1 -> n
    dp[i][j] = search (i,j) // 直接求i,j最为结尾最长子序列


int search(int x, int y, int[][] A)   {
    if(flag[x][y] != 0) // 遍历过直接返回
        return dp[x][y];

    for(int i = 0; i < 4; i++) {
        nx = x + dx[i];
        ny = y + dy[i];
        if( A[x][y] > A[nx][ny]) {
            // 通过 search( nx, ny, A) 更新最长子序列
        }
    }
    //返回答案
}
```

Longest Increasing continuous Subsequence 2D

- 那怎么根据DP四要素转化为记忆化搜索呢?
- State: dp[x][y] 以x,y作为结尾的最长子序列
- Function:
  - 遍历x,y 上下左右四个格子
  - dp[x][y] = dp[nx][ny] + 1
    - (if dp[x][y] > dp[nx][ny])
- Intialize:
  - dp[x][y] 是极小值时，初始化为1
- Answer: dp[x][y]中最大值

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


int search(int x, int y, int[][] A)   {
    if(flag[x][y] != 0)
        return dp[x][y];


    dp[x][y] = 0;  // Intialize
    for(int i = 0; i < 4; i++) { // Intialize
        nx = x + dx[i];
        ny = y + dy[i];
        if( A[x][y] > A[nx][ny]) { // Function
            dp[x][y] = Math.max(dp[i][j],  search( nx, ny, A) + 1);
        }
    }

    return dp[x][y]; //Answer
}
```

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DP

Coins in a line

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

博弈类DP

博弈有先后手

- State:
  - 定义一个人的状态
- Function:
  - 考虑两个人的状态做状态更新
- Intialize:
- Answer:

先思考最小状态

然后思考大的状态-> 往小的递推，那么非常适合记忆化搜索

Coins in a line

- State:
  - dp[i] 现在还剩i个硬币，现在先手取硬币的人最后输赢状况
- Function:
  - dp[n] = !dp[n-1] || !dp[n-2]
- Intialize:
  - dp[0] = false
  - dp[1] = true
  - dp[2] = true
- Answer:
  - dp[n]

Coins in a Line II

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

[5,1,2,10]

Coins in a Line II

- State:
  - dp[i] 现在还剩i个硬币，现在先手取硬币的人最后最多取硬币价值
- Function:
  - n 是所有硬币数目
  - sum[i] 是后i个硬盘的总和
  - dp[i] = sum[i]-min(dp[i-1], dp[i-2])
- Intialize:
  - dp[0] = 0
  - dp[1] = coin[i-1]
  - dp[2] = coin[i-2] + coin[i-1]
- Answer:
  - dp[n]

Coins in a Line III

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

www.jiuzhang.com/solutions/coins-in-a-line-iii

Coins in a Line III

- State:
  - dp[i][j] 现在还第i到第j的硬币，现在先手取硬币的人最后最多取硬币价值
- Function:
  - Sum[i][j]第i到第j的硬币价值总和
  - dp[i][j] = sum[i][j] - min(dp[i-1][j], dp[i][j-1]);
- Intialize:
  - dp[i][i] = coin[i],
  - dp[i][i+1] = max(coin[i], coin[i+1]),
- Answer:
  - dp[0][n-1]

区间类Dp

特点：

1. 求一段区间的解max/min/count
2. 转移方程通过区间更新
3. 从大到小的更新

Stone-Game

http://www.lintcode.com/en/problem/stone-game/
http://www.jiuzhang.com/solutions/stone-game/

[3,4,5,6]

Stone-Game

- 死胡同：容易想到的一个思路从小往大，枚举第一次合并是在哪?
- 记忆化搜索的思路，从大到小，先考虑最后的0-n-1 合并的总花费
- State:
  - dp[i][j] 表示把第i到第j个石子合并到一起的最小花费
- Function:
  - 预处理sum[i,j] 表示i到j所有石子价值和
  - dp[i][j] = min(dp[i][k]+dp[k+1][j]+sum[i,j]) 对于所有k属于{i,j}
- Intialize:
  - for each i
    - dp[i][i] = 0
- Answer:
  - dp[0][n-1]

Burst Ballons

http://www.lintcode.com/en/problem/burst-balloons/

http://www.jiuzhang.com/solutions/burst-ballons/

Burst Ballons

- 死胡同：容易想到的一个思路从小往大，枚举第一次在哪吹爆气球?
- 记忆化搜索的思路，从大到小，先考虑最后的0-n-1 合并的总价值

- State:
  - dp[i][j] 表示把第i到第j个气球打爆的最大价值
- Function:
  - 对于所有k属于{i,j}
    - midValue = arr[i- 1] * arr[k] * arr[j+ 1];
    - dp[i][j] = min(dp[i][k-1]+d[k+1][j]+midvalue)
- Intialize:
  - for each i
    - dp[i][i] = 0
- Answer:
  - dp[0][n-1]

Scramble String

http://www.lintcode.com/en/problem/scramble-string/
http://www.jiuzhang.com/solutions/scramble-string/
[abcd, dcab]

Scramble String

- 看 f[great][rgreat] 这个参考例子
- f[gr|eat][rgreat] =
  - f[gr][rg] && f[eat][eat]
  - f[gr][at] && f[eat][rgr]

Scramble String

- State:
  - dp[x][y][k] 表示是从s1串x开始，s2串y开始，他们后面k个字符组成的substr是Scramble String
- Function:
  - 对于所有i属于{1,k}
  - s11 = s1.substring(0, i); s12 = s1.substring(i, s1.length());
  - s21 = s2.substring(0, i); s22 = s2.substring(i, s2.length());
  - s23 = s2.substring(0, s2.length() - i); s24 = s2.substring(s2.length() - i, s2.length());
  - for i = x -> x+k
    - dp[x][y][k] = (dp[x][y][i] && dp[x+i][y+i][k-i]) || dp[x][y+k-i][i] && dp[x+i][y][k-i])
- Intialize:
  - dp[i][j][1] = s1[i]==s[j].
- Answer:
  - dp[0][0][len]

区间DP

- coin in a line III
- stone game
- scramble string

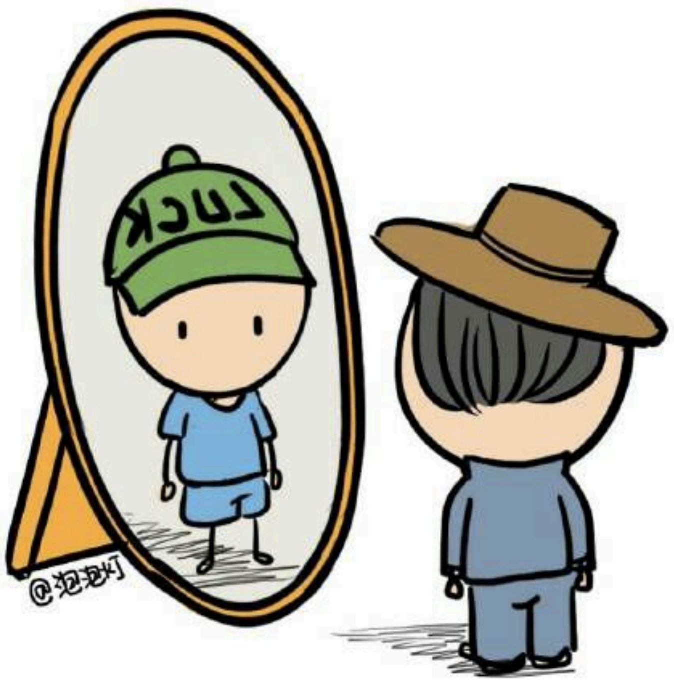
- 这种题目共性就是区间最后求[0,n-1] 这样一个区间
- 逆向思维分析 从大到小就能迎刃而解

- 逆向=》 分治类似

什么时候用记忆化搜索?

- 状态转移特别麻烦，不是顺序性。
  - **Longest Increasing continuous Subsequence 2D**
    - 遍历x,y 上下左右四个格子 dp[x][y] = dp[nx][ny]
  - **Coins in a Line III**
    - pick_left = min(dp[i+2][j], dp[i+1][j-1]) + coin[i];
    - pick_right = min(dp[i][j-2], dp[i+1][i-1])+coin[j];
    - dp[i][j] = max(pick_left, pick_right);

- 初始化状态不是很容易找到
  - **Stone Game**
    - **初始化dp[i][i] = 0**
  - **Longest Increasing continuous Subsequence 2D**
    - **初始化极小值**



The only person you should compare yourself to,

is the person you were yesterday.

唯一能够和你相比较的，就是那个曾经的自己。

今日重点三题

- **House Robber**
  - 滚动数组优化最简单的入门。
- **Longest Increasing continuous Subsequence 2D**
  - 记忆化搜索的经典题，此题只有记忆化搜索才能最优。
- **Coins in a Line III**
  - 博弈问题和记忆化搜索的结合

Thank You