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

Facebook: http://www.facebook.com/ninechapter

Weibo: http://www.weibo.com/ninechapter

Outline

复习上一节课的内容

双序列动态规划

背包问题的动态规划

如何想到使用DP

1. One of the following three
    a) Maximum/Minimum
    b) Yes/No
    c) Count(*)

2. Can not sort / swap
    http://www.lintcode.com/en/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 (10%)
2. Sequence (40%)
3. Two Sequences DP (40%)
4. Backpack (10%)

3. Two Sequences Dp

state: f[i][j]代表了第一个sequence的前i个数字/字符 配上第二个sequence的前j个...

function: f[i][j] = 研究第i个和第j个的匹配关系

intialize: f[i][0] 和 f[0][i]

answer: f[s1.length()][s2.length()]

Longest Common Subsequence

http://www.lintcode.com/en/problem/longest-common-subsequence/
http://www.ninechapter.com/solutions/longest-common-subsequence/

Longest Common Subsequence

state: f[i][j]表示前i个字符配上前j个字符的LCS的长度

function: f[i][j] = f[i-1][j-1] + 1 // a[i] == b[j]

= MAX(f[i-1][j], f[i][j-1]) // a[i] != b[j]

intialize: f[i][0] = 0

f[0][j] = 0

answer: f[a.length()][b.length()]

Longest Common Substring

http://www.lintcode.com/en/problem/longest-common-substring/
http://www.ninechapter.com/solutions/longest-common-substring/

Longest Common Substring

state: f[i][j]表示前i个字符配上前j个字符的LCS‘的长度
(一定以第i个和第j个结尾的LCS’)

function: f[i][j] = f[i-1][j-1] + 1 // a[i] == b[j]
= 0 // a[i] != b[j]

intialize: f[i][0] = 0
f[0][j] = 0

answer: MAX(f[0..a.length()][0..b.length()])

Edit Distance

http://www.lintcode.com/en/problem/edit-distance/

http://www.ninechapter.com/solutions/edit-distance/

Edit Distance

state: f[i][j]a的前i个字符“配上”b的前j个字符最少要用几次编辑使得他们相等

function:

f[i][j] = MIN(f[i-1][j-1], f[i-1][j]+1, f[i][j-1]+1) // a[i] == b[j]

= MIN(f[i-1][j], f[i][j-1], f[i-1][j-1]) + 1 // a[i] != b[j]

intialize: f[i][0] = i, f[0][j] = j

answer: f[a.length()][b.length()]

其他题目

Distinct Subsequence

Interleaving String

4. Backpack DP

背包问题

Backpack

http://www.lintcode.com/en/problem/backpack/
http://www.ninechapter.com/solutions/backpack/

Backpack

n个整数a[1..n], 装m的背包

state: f[i][j] “前i”个数, 取出一些能否组成和为j

function: f[i][j] = f[i-1][j - a[i]] or f[i-1][j]

intialize: f[X][0] = true; f[0][1..m] = false

answer: 能够使得f[n][X]最大的X(0<=X<=m)

Backpack II

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

Backpack II

n个物品，背包为m，体积a[1..n]，价值v[1..n]

state: f[i][j]表示前i个物品中，取出“若干”物品后，体积“正好”为j的最大价值。

function: f[i][j] = max{f[i-1][j], f[i-1][j - a[i]] + v[i]}

intialize: f[X][0] = 0, f[0][1..m] = -oo

answer: f[n][1..m]中最大值

k Sum

http://www.lintcode.com/en/problem/k-sum/

k Sum

state: f[i][j][t]前i个数取j个数出来能否和为t

function: f[i][j][t] = f[i - 1][j - 1][t - a[i]] or

f[i - 1][j][t]

1. 问是否可行 (DP) - f[x][0][0] = true
2. 问方案总数 (DP) - f[x][0][0] = 1
3. 问所有方案 (递归/搜索)

Minimum Adjustment Cost

http://www.lintcode.com/en/problem/minimum-adjustment-cost/

Minimum Adjustment Cost

n个数，可以对每个数字进行调整，使得相邻的两个数的差都<=target，调整的费用为

Sigma(|A[i]-B[i]|)

A[i]原来的序列 B[i]是调整后的序列

A[i] < 200, target < 200

让代价最小

B[woB[]B[]B[B

最小调整代价

state: f[i][v] 前i个数，第i个数调整为v，满足相邻两数<=target，所需要的最小代价

function: f[i][v] = min(f[i-1][v’] + |A[i]-v|, |v-v’| <= target)

intialize: f[1][A[1]] = 0, f[1][A[1] +- X] = X

answer: f[n][X]

O(n * A * T)

Conclusion

4 key points of DP:

1. State
2. Function
3. Initialize / start
4. Answer / end

Recursive VS DP

递归是一种程序的实现方式:函数的自我调用

```
Function(x) {
    …
    Funciton(x-1);
    …
}
```

动态规划是一种解决问题的思想:大规模问题的结果，是由小规模问题的结果运算得来的。

动态规划可以用递归来实现(Memorization Search)