动态规划：从新手到专家

March 26, 2013

作者：Hawstein
出处：http://hawstein.com/posts/dp-novice-to-advanced.html

前言

本文翻译自TopCoder上的一篇文章：Dynamic Programming: From novice to advanced，并非严格逐字逐句翻译，其中加入了自己的一些理解。水平有限，还望指摘。

前言_

我们遇到的问题中，有很大一部分可以用动态规划(简称DP)来解。解决这类问题可以很大地提升你的能力与技巧，我会试着帮助你理解如何使用DP来解题。这篇文章是基于实例展开来讲的，因为干巴巴的理论实在不好理解。

注意：如果你对于其中某一节已经了解并且不想阅读它，没关系，直接跳过它即可。

简介(入门)

什么是动态规划，我们要如何描述它?

动态规划算法通常基于一个递推公式及一个或多个初始状态。当前子问题的解将由上一次子问题的解推出。使用动态规划来解题只需要多项式时

间复杂度， 因此它比回溯法、暴力法等要快许多。

现在让我们通过一个例子来了解一下DP的基本原理。

首先，我们要找到某个状态的最优解，然后在它的帮助下，找到下一个状态的最优解。

“状态”代表什么及如何找到它?

“状态”用来描述该问题的子问题的解。原文中有两段作者阐述得不太清楚，跳过直接上例子。

如果我们有面值为1元、3元和5元的硬币若干枚，如何用最少的硬币凑够11元？（表面上这道题可以用贪心算法，但贪心算法无法保证可以求出解，比如1元换成2元的时候）

首先我们思考一个问题，如何用最少的硬币凑够i元(i<11)？为什么要这么问呢？ 两个原因：1.当我们遇到一个大问题时，总是习惯把问题的规模变小，这样便于分析讨论。 2.这个规模变小后的问题和原来的问题是同质的，除了规模变小，其它的都是一样的， 本质上它还是同一个问题(规模变小后的问题其实是原问题的子问题)。

好了，让我们从最小的i开始吧。当i=0，即我们需要多少个硬币来凑够0元。 由于1，3，5都大于0，即没有比0小的币值，因此凑够0元我们最少需要0个硬币。（这个分析很傻是不是？别着急，这个思路有利于我们理清动态规划究竟在做些什么。）这时候我们发现用一个标记来表示这句“凑够0元我们最少需要0个硬币。”会比较方便， 如果一直用纯文字来表述，不出一会儿你就会觉得很绕了。那么， 我们用d(i)=j来表示凑够i元最少需要j个硬币。于是我们已经得到了d(0)=0， 表示凑够0元最小需要0个硬币。当i=1时，只有面值为1元的硬币可用， 因此我们拿起一个面值为1的硬币，接下来只需要凑够0元即可，而这个是已经知道答案的， 即d(0)=0。所以，d(1)=d(1-1)+1=d(0)+1=0+1=1。当i=2时， 仍然只有面值为1的硬币可用，于是我拿起一个面值为1的硬币， 接下来我只需要再凑够2-1=1元即可(记得要用最小的硬币数量)，而这个答案也已经知道了。 所以

d(2)=d(2-1)+1=d(1)+1=1+1=2。一直到这里，你都可能会觉得，好无聊，感觉像做小学生的题目似的。因为我们一直都只能操作面值为1的硬币！耐心点， 让我们看看i=3时的情况。当i=3时，我们能用的硬币就有两种了：1元的和3元的(5元的仍然没用，因为你需要凑的数目是3元！5元太多了亲)。 既然能用的硬币有两种，我就有两种方案。如果我拿了一个1元的硬币，我的目标就变为了： 凑够3-1=2元需要的最少硬币数量。即d(3)=d(3-1)+1=d(2)+1=2+1=3。 这个方案说的是，我拿3个1元的硬币；第二种方案是我拿起一个3元的硬币， 我的目标就变成：凑够3-3=0元需要的最少硬币数量。即d(3)=d(3-3)+1=d(0)+1=0+1=1. 这个方案说的是，我拿1个3元的硬币。好了，这两种方案哪种更优呢？ 记得我们可是要用最少的硬币数量来凑够3元的。所以， 选择d(3)=1，怎么来的呢？具体是这样得到的：d(3)=min{d(3-1)+1, d(3-3)+1}。

OK，码了这么多字讲具体的东西，让我们来点抽象的。从以上的文字中， 我们要抽出动态规划里非常重要的两个概念：状态和状态转移方程。

上文中d(i)表示凑够i元需要的最少硬币数量，我们将它定义为该问题的"状态"， 这个状态是怎么找出来的呢？我在另一篇文章 动态规划之背包问题(一)中写过： 根据子问题定义状态。你找到子问题，状态也就浮出水面了。 最终我们要求解的问题，可以用这个状态来表示：d(11)，即凑够11元最少需要多少个硬币。 那状态转移方程是什么呢？既然我们用d(i)表示状态，那么状态转移方程自然包含d(i)， 上文中包含状态d(i)的方程是：d(3)=min{d(3-1)+1, d(3-3)+1}。没错， 它就是状态转移方程，描述状态之间是如何转移的。当然，我们要对它抽象一下，

$d(i)=\min\{ d(i-v_j)+1 \}$，其中$i-v_j >=0$，$v_j$表示第j个硬币的面值;

有了状态和状态转移方程，这个问题基本上也就解决了。当然了，Talk is cheap,show me the code!

伪代码如下：

```
Set Min[i] equal to Infinity for all of i
Min[0]=0

For i = 1 to S
For j = 0 to N - 1
   If (Vj<=i AND Min[i-Vj]+1<Min[i])
Then Min[i]=Min[i-Vj]+1

Output Min[S]
```

下图是当i从0到11时的解：

Sum	Min. nr. of coins	Coin value added to a smaller sum to obtain this sum (it is displayed in brackets)
0	0	-
1	1	1 (0)
2	2	1 (1)
3	1	3 (0)
4	2	1 (3)
5	1	5 (0)
6	2	3 (3)
7	3	1 (6)
8	2	3 (5)
9	3	1 (8)
10	2	5 (5)
11	3	1 (10)

从上图可以得出，要凑够11元至少需要3枚硬币。

此外，通过追踪我们是如何从前一个状态值得到当前状态值的， 可以找到每一次我们用的是什么面值的硬币。比如，从上面的图我们可以看出，最终结果d(11)=d(10)+1(面值为1)，而d(10)=d(5)+1(面值为5)，最后d(5)=d(0)+1 (面值为5)。所以我们凑够11元最少需要的3枚硬币是：1元、5元、5元。

注意：原文中这里本来还有一段的，但我反反复复读了几遍， 大概的意思我已经在上文从i=0到i=3的分析中有所体现了。作者本来想讲的通俗一些， 结果没写好，反而更不好懂，所以这段不翻译了。

初级

上面讨论了一个非常简单的例子。现在让我们来看看对于更复杂的问题，如何找到状态之间的转移方式(即找到状态转移方程)。 为此我们要引入一个新词叫递推关系来将状态联系起来(说的还是状态转移方程)

OK，上例子，看看它是如何工作的。

一个序列有N个数：A[1],A[2],…,A[N]，求出最长非降子序列的长度。（讲DP基本都会讲到的一个问题LIS：longest increasing subsequence）

正如上面我们讲的，面对这样一个问题，我们首先要定义一个“状态”来代表它的子问题， 并且找到它的解。注意，大部分情况下，某个状态只与它前面出现的状态有关， 而独立于后面的状态。

让我们沿用“入门”一节里那道简单题的思路来一步步找到“状态”和“状态转移方程”。 假如我们考虑求A[1],A[2],…,A[i]的最长非降子序列的长度，其中i<N， 那么上面的问题变成了原问题的一个子问题(问题规模变小了，你可以让i=1,2,3等来分析) 然后我们定义d(i)，表示前i个数中以A[i]结尾的最长非降子序列的长度。OK， 对照“入门”中的简单题，你应该可以估计到这个d(i)就是我们要找的状态。 如果我们把d(1)到d(N)都计算出来，那么最终我们要找的答案就是这里面最大的那个。 状态找到了，下一步找出状态转移方程。

为了方便理解我们是如何找到状态转移方程的，我先把下面的例子提到前面来讲。 如果我们要求的这N个数的序列是：

根据上面找到的状态，我们可以得到：（下文的最长非降子序列都用LIS表示）

- 前1个数的LIS长度d(1)=1(序列：5)
- 前2个数的LIS长度d(2)=1(序列：3；3前面没有比3小的)
- 前3个数的LIS长度d(3)=2(序列：3，4；4前面有个比它小的3，所以d(3)=d(2)+1)
- 前4个数的LIS长度d(4)=3(序列：3，4，8；8前面比它小的有3个

数，所以 d(4)=max{d(1),d(2),d(3)}+1=3)

OK，分析到这，我觉得状态转移方程已经很明显了，如果我们已经求出了d(1)到d(i-1)， 那么d(i)可以用下面的状态转移方程得到：

```
d(i) = max{1, d(j)+1},其中j<i,A[j]<=A[i]
```

用大白话解释就是，想要求d(i)，就把i前面的各个子序列中， 最后一个数不大于A[i]的序列长度加1，然后取出最大的长度即为d(i)。 当然了，有可能i前面的各个子序列中最后一个数都大于A[i]，那么d(i)=1， 即它自身成为一个长度为1的子序列。

分析完了，上图：（第二列表示前i个数中LIS的长度， 第三列表示，LIS中到达当前这个数的上一个数的下标，根据这个可以求出LIS序列）

I	The length of the longest non-decreasing sequence of first i numbers	The last sequence i from which we "arrived" to this one
1	1	1 (first number itself)
2	1	2 (second number itself)
3	2	2
4	3	3
5	3	3
6	4	5

Talk is cheap, show me the code:

```
#include <iostream>
using namespace std;

int lis(int A[], int n){
    int *d = new int[n];
    int len = 1;
    for(int i=0; i<n; ++i){
        d[i] = 1;
        for(int j=0; j<i; ++j)
            if(A[j]<=A[i] && d[j]+1>d[i])
```

```
                d[i] = d[j] + 1;
        if(d[i]>len) len = d[i];
    }
    delete[] d;
    return len;
}
int main(){
    int A[] = {
        5, 3, 4, 8, 6, 7
    };
    cout<<lis(A, 6)<<endl;
    return 0;
}
```

该算法的时间复杂度是O(n^2)，并不是最优的解法。 还有一种很巧妙的算法可以将时间复杂度降到O(nlogn)，网上已经有各种文章介绍它， 这里就不再赘述。传送门： LIS的O(nlogn)解法。 此题还可以用“排序+LCS”来解，感兴趣的话可自行Google。

练习题

无向图G有N个结点(1<N<=1000)及一些边，每一条边上带有正的权重值。 找到结点1到结点N的最短路径，或者输出不存在这样的路径。

提示：在每一步中，对于那些没有计算过的结点， 及那些已经计算出从结点1到它的最短路径的结点，如果它们间有边， 则计算从结点1到未计算结点的最短路径。

尝试解决以下来自topcoder竞赛的问题：

- ZigZag - 2003 TCCC Semifinals 3
- BadNeighbors - 2004 TCCC Round 4
- FlowerGarden - 2004 TCCC Round 1

中级

接下来，让我们来看看如何解决二维的DP问题。

平面上有N*M个格子，每个格子中放着一定数量的苹果。你从左上角的格子开始， 每一步只能向下走或是向右走，每次走到一个格子上就把格子里的苹果收集起来， 这样下去，你最多能收集到多少个苹果。

解这个问题与解其它的DP问题几乎没有什么两样。第一步找到问题的“状态”， 第二步找到“状态转移方程”，然后基本上问题就解决了。

首先，我们要找到这个问题中的“状态”是什么？我们必须注意到的一点是， 到达一个格子的方式最多只有两种：从左边来的(除了第一列)和从上边来的(除了第一行)。 因此为了求出到达当前格子后最多能收集到多少个苹果， 我们就要先去考察那些能到达当前这个格子的格子，到达它们最多能收集到多少个苹果。 (是不是有点绕，但这句话的本质其实是DP的关键：欲求问题的解，先要去求子问题的解)

经过上面的分析，很容易可以得出问题的状态和状态转移方程。 状态S[i][j]表示我们走到(i, j)这个格子时，最多能收集到多少个苹果。那么， 状态转移方程如下：

```
S[i][j]=A[i][j] + max(S[i-1][j], if i>0 ; S[i][j-1], if j>0)
```

其中i代表行，j代表列，下标均从0开始；A[i][j]代表格子(i, j)处的苹果数量。

S[i][j]有两种计算方式：1.对于每一行，从左向右计算，然后从上到下逐行处理；2. 对于每一列，从上到下计算，然后从左向右逐列处理。 这样做的目的是为了在计算S[i][j]时，S[i-1][j]和S[i][j-1]都已经计算出来了。

伪代码如下：

```
For i = 0 to N - 1
   For j = 0 to M - 1
   S[i][j] = A[i][j] +
      max(S[i][j-1], if j>0 ; S[i-1][j], if i>0 ; 0)

Output S[n-1][m-1]
```

以下两道题来自topcoder，练习用的。

- [AvoidRoads]() - 2003 TCO Semifinals 4
- [ChessMetric]() - 2003 TCCC Round 4

中高级

这一节要讨论的是带有额外条件的DP问题。

以下的这个问题是个很好的例子。

无向图G有N个结点，它的边上带有正的权重值。

你从结点1开始走，并且一开始的时候你身上带有M元钱。如果你经过结点i， 那么你就要花掉S[i]元(可以把这想象为收过路费)。如果你没有足够的钱， 就不能从那个结点经过。在这样的限制条件下，找到从结点1到结点N的最短路径。 或者输出该路径不存在。如果存在多条最短路径，那么输出花钱数量最少的那条。 限制：1<N<=100 ; 0<=M<=100 ; 对于每个i, 0<=S[i]<=100；正如我们所看到的， 如果没有额外的限制条件(在结点处要收费，费用不足还不给过)，那么， 这个问题就和经典的迪杰斯特拉问题一样了(找到两结点间的最短路径)。 在经典的迪杰斯特拉问题中， 我们使用一个一维数组来保存从开始结点到每个结点的最短路径的长度， 即M[i]表示从开始结点到结点i的最短路径的长度。然而在这个问题中， 我们还要保存我们身上剩余多少钱这个信息。因此，很自然的， 我们将一维数组扩展为二维数组。M[i][j]表示从开始结点到结点i的最短路径长度，且剩余j元。通过这种方式，我们将这个问题规约到原始的路径寻找问题。 在每一步中，对于已经找到的最短路径，我们找到它所能到达的下一个未标记状态(i,j)， 将它标记为已访问(之后不再访问这个结点)，并且在能到达这个结点的各个最短路径中， 找到加上当前边权重值后最小值对应的路径，即为该结点的最短路径。(写起来真是绕，建议画个图就会明了很多)。不断重复上面的步骤， 直到所有的结点都访问到为止(这里的访问并不是要求我们要经过它， 比如有个结点收费很高，你没有足够的钱去经过它，但你已经访问过它) 最后Min[N-1][j]中的最小值即是问题的答案(如果有多个最小值， 即有多条最短路径，那么选择j最大的那条路

径，即，使你剩余钱数最多的最短路径)。

伪代码：

```
Set states(i,j) as unvisited for all (i,j)
Set Min[i][j] to Infinity for all (i,j)

Min[0][M]=0

While(TRUE)

Among all unvisited states(i,j) find the one for which Min[i][j]
is the smallest. Let this state found be (k,l).

If there wasn't found any state (k,l) for which Min[k][l] is
less than Infinity - exit While loop.

Mark state(k,l) as visited

For All Neighbors p of Vertex k.
   If (l-S[p]>=0 AND
    Min[p][l-S[p]]>Min[k][l]+Dist[k][p])
      Then Min[p][l-S[p]]=Min[k][l]+Dist[k][p]
   i.e.
If for state(i,j) there are enough money left for
going to vertex p (l-S[p] represents the money that
will remain after passing to vertex p), and the
shortest path found for state(p,l-S[p]) is bigger
than [the shortest path found for
state(k,l)] + [distance from vertex k to vertex p)],
then set the shortest path for state(i,j) to be equal
to this sum.
End For

End While

Find the smallest number among Min[N-1][j] (for all j, 0<=j<=M);
if there are more than one such states, then take the one with greater
j. If there are no states(N-1,j) with value less than Infinity - then
such a path doesn't exist.
```

下面有几道topcoder上的题以供练习：

- Jewelry - 2003 TCO Online Round 4
- StripePainter - SRM 150 Div 1
- QuickSums - SRM 197 Div 2
- ShortPalindromes - SRM 165 Div 2

高级

以下问题需要仔细的揣摩才能将其规约为可用DP解的问题。

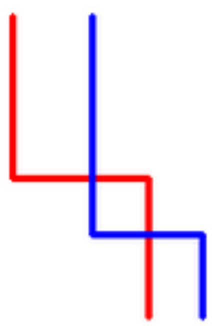
问题：StarAdventure - SRM 208 Div 1:

给定一个M行N列的矩阵(M*N个格子)，每个格子中放着一定数量的苹果。你从左上角的格子开始，只能向下或向右走，目的地是右下角的格子。你每走过一个格子，就把格子上的苹果都收集起来。然后你从右下角走回左上角的格子， 每次只能向左或是向上走，同样的，走过一个格子就把里面的苹果都收集起来。 最后，你再一次从左上角走到右下角，每过一个格子同样要收集起里面的苹果 (如果格子里的苹果数为0，就不用收集)。求你最多能收集到多少苹果。

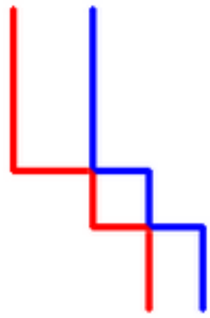
注意：当你经过一个格子时，你要一次性把格子里的苹果都拿走。

限制条件：1 < N, M <= 50；每个格子里的苹果数量是0到1000(包含0和1000)。

如果我们只需要从左上角的格子走到右下角的格子一次，并且收集最大数量的苹果， 那么问题就退化为“中级”一节里的那个问题。将这里的问题规约为“中级”里的简单题， 这样一来会比较好解。让我们来分析一下这个问题，要如何规约或是修改才能用上DP。 首先，对于第二次从右下角走到左上角得出的这条路径， 我们可以将它视为从左上角走到右下角得出的路径，没有任何的差别。 (即从B走到A的最优路径和从A走到B的最优路径是一样的)通过这种方式， 我们得到了三条从顶走到底的路径。对于这一点的理解可以稍微减小问题的难度。 于是，我们可以将这3条路径记为左，中，右路径。对于两条相交路径(如下图)：



在不影响结果的情况下，我们可以将它们视为两条不相交的路径：



这样一来，我们将得到左，中，右3条路径。此外，如果我们要得到最优解， 路径之间不能相交(除了左上角和右下角必然会相交的格子)。因此对于每一行y(除了第一行和最后一行)，三条路径对应的x坐标要满足：x1[y] < x2[y] < x3[y]。 经过这一步的分析，问题的DP解法就进一步地清晰了。让我们考虑行y， 对于每一个x1[y-1]，x2[y-1]和x3[y-1]，我们已经找到了能收集到最多苹果数量的路径。 根据它们，我们能求出行y的最优解。现在我们要做的就是找到从一行移动到下一行的方式。 令Max[i][j][k]表示到第y-1行为止收集到苹果的最大数量， 其中3条路径分别止于第i,j,k列。对于下一行y，对每个Max[i][j][k] 都加上格子(y,i)，(y,j)和(y,k)内的苹果数量。因此，每一步我们都向下移动。 我们做了这一步移动之后，还要考虑到，一条路径是有可能向右移动的。（对于每一个格子，我们有可能是从它上面向下移动到它， 也可能是从它左边向右移动到它)。为了保证3条路径互不相交， 我们首先要考虑左边的路径向右移动的情况，然后是中间，最后是右边的路径。 为了更好的理解，让我们来考虑左边的路径向右移动的情况，对于每一个可能的j,k对(j<k)， 对每个i(i<j)，考虑从位置(i-1,j,k)移动到位置(i,j,k)。处理完左边的路径， 再处理中间的路径，最后处理右边的路径。方法都差不多。

用于练习的topcoder题目：

- MiniPaint - SRM 178 Div 1

其它

当阅读一个题目并且开始尝试解决它时，首先看一下它的限制。 如果要求在多项式时间内解决，那么该问题就很可能要用DP来解。遇到这种情况， 最重要的就是找到问题的“状态”和“状态转移方程”。(状态不是随便定义的， 一般定义完状态，你要找到当前状态是如何从前面的状态得到的， 即找到状态转移方程)如果看起来是个DP问题，但你却无法定义出状态， 那么试着将问题规约到一个已知的DP问题(正如“高级”一节中的例子一样)。

后记

看完这教程离DP专家还差得远，好好coding才是王道。