动态规划

瞿绍军

什么是动态规划

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。

例题1 凑硬币

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

例题1凑硬币

- 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)。

例题1凑硬币

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。

例题1凑硬币

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。

例题1凑硬币

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。

例题1凑硬币

- 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)。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}。

例题1凑硬币

- 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”
- 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

例题1凑硬币

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

```
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]
```

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)

例题1凑硬币

- 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。

例题2最长非降子序列的长度

- 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)

- 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。

例题2最长非降子序列的长度

- 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。
- 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。

例题2最长非降子序列的长度

- 如果我们要求的这N个数的序列是：

  5，3，4，8，6，7

- 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)

例题2最长非降子序列的长度

- 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]

- 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。

例题2最长非降子序列的长度

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

- 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

例题2最长非降子序列的长度

```
#include <iostream>
using namespace std;
 intlis(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;
}
```

例题3 收集苹果

路径经过的最值

[题意简述]

平面上有N*M个格子，每个格子中放着一定数量的苹果。从左上角的格子开始，每一步只能向下走或是向右走，每次走到一个格子就把格子里的苹果收集起来，这样一直走到右下角，问最多能收集到多少个苹果。

例题3 收集苹果

[输入格式]

第一行为两个整数N，M(N，M<=10000)，表示棋盘；

第2..N+1行：每行有M个整数，表示每个格子里放置的苹果。

[输出格式]

输出仅一行为一个数，表示最多能收集到多少个苹果。

例题3 收集苹果

[输入样例]

6 6

5 8 5 7 1 8

1 3 2 8 7 9

7 8 6 6 8 7

9 9 8 1 6 3

2 4 10 2 6 2

5 5 2 1 8 8

[输出样例]

76

例题3 收集苹果

根据题目的规则“每一步只能向下走或是向右走”，可以从左上角开始将到达第一行和第一列中各点所能收集到的最大苹果数量填成一张表格。如下图：

	1	2	3	4	5	6
1	5	8	5	7	1	8
2	1	3	2	8	7	9
3	7	8	6	6	8	7
4	9	9	8	1	6	3
5	2	4	10	2	6	2
6	5	5	2	1	8	8

	1	2	3	4	5	6
1	5	13	18	25	26	34
2	6					
3	13					
4	22					
5	24					
6	29					

例题3 收集苹果

同理，填所在格能获得的最大苹果数就是看它左面一格和上面一格哪个值更大，就取哪个值再加上自己格子里面的苹果数，就是到达此格能获得的最大苹果数。依此填完所有格子，最后得到下图：

	1	2	3	4	5	6
1	5	8	5	7	1	8
2	1	3	2	8	7	9
3	7	8	6	6	8	7
4	9	9	8	1	6	3
5	2	4	10	2	6	2
6	5	5	2	1	8	8

	1	2	3	4	5	6
1	5	13	18	25	26	34
2	6	16	20	33	40	49
3	13	24	30	39	48	56
4	22	33	41	42	54	59
5	24	37	51	53	60	62
6	29	42	53	54	68	76

例题3 收集苹果

- 第一步找到问题的“状态”
- 第二步找到“状态转移方程”
- 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)

例题3 收集苹果

- 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。

例题3 收集苹果

- 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。

例题3 收集苹果

①划分阶段：以每一个格子为阶段。

②确立状态及状态变量：到达每一个格子时所能收集到的最多的苹果数

③决策：选择左面一格还是上面一格

④状态转移方程：

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

例题3 收集苹果

```
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]
```

例题3 收集苹果

参考程序片段

```
//C++代码
b[1][1]=a[1][1];  //初始化
for (i=2;i<=n;i++) b[i][1]=b[i-1][1]+a[i][1];
for (i=2;i<=m;i++) b[1][i]=b[1][i]+a[1][i];
for (i=2;i<=n;i++)  //dp
  for (i=2;i<=m;i++)
    if (b[i-1][j]>b[i][j-1])
        b[i][j]=b[i-1][j]+a[i][j];
    else
        b[i][j]=b[i][j-1]+a[i][j];
printf("%d\n",b[n][m]);
```

小结

动态规划和一般递推的不同点？

（1）递推的边界条件一般很明显，而动态规划的边界条件比较隐蔽，容易被忽视；

（2）递推的数学性一般较强，而动态规划的数学性相对较弱；

（3）递推一般不划分阶段，而动态规划一般有较为明显的阶段；

（4）动态规划往往是用来求一个最优值，而一般的递推往往是用来计数或是求一个值。

动态规划和一般递推的相同点？

无后效性和有边界条件。