程序设计实习

郭炜 微博 http://weibo.com/guoweiofpku
http://blog.sina.com.cn/u/3266490431
刘家瑛 微博 http://weibo.com/pkuliujiaying

动态规划

数字三角形

例题一、数字三角形(POJ1163)

```
        7
      3   8
    8   1   0
  2   7   4   4
4   5   2   6   5
```

在上面的数字三角形中寻找一条从顶部到底边的路径，使得路径上所经过的数字之和最大。路径上的每一步都只能往左下或右下走。只需要求出这个最大和即可，不必给出具体路径。

三角形的行数大于1小于等于100，数字为 0 - 99

输入格式：

5　//三角形行数。下面是三角形
7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

要求输出最大和

解题思路：

用二维数组存放数字三角形。

D(r, j) ：第r行第 j 个数字(r,j从1开始算)
MaxSum(r, j) ： 从D(r,j)到底边的各条路径中，
最佳路径的数字之和。

问题：求 MaxSum(1,1)

典型的递归问题。

D(r, j)出发，下一步只能走D(r+1,j)或者D(r+1, j+1)。故对于N行的三角形：

```
if ( r == N)
        MaxSum(r,j) = D(r,j)
else
        MaxSum( r, j) = Max{ MaxSum(r＋1,j), MaxSum(r+1,j+1) } + D(r,j)
```

数字三角形的递归程序:

```
#include <iostream>
#include <algorithm>
#define MAX 101
using namespace std;
int D[MAX][MAX];
int n;
int MaxSum(int i, int j){
  if(i==n)
     return D[i][j];
  int x = MaxSum(i+1,j);
  int y = MaxSum(i+1,j+1);
  return max(x,y)+D[i][j];
}
```

```
int main(){
  int i,j;
  cin >> n;
  for(i=1;i<=n;i++)
        for(j=1;j<=i;j++)
              cin >> D[i][j];
  cout << MaxSum(1,1) << endl;
}
```

为什么超时？

7_1

$3_1 \quad 8_1$

$8_1 \quad 1_2 \quad 0_1$

$2_1 \quad 7_3 \quad 4_3 \quad 4_1$

$4_1 \quad 5_4 \quad 2_6 \quad 6_4 \quad 5_1$

- 回答：重复计算

如果采用递规的方法，深度遍历每条路径，存在大量重复计算。则时间复杂度为 2^n，对于 n = 100 行，肯定超时。

改进

如果每算出一个MaxSum(r,j)就保存起来，下次用到其值的时候直接取用，则可免去重复计算。那么可以用$O(n^2)$时间完成计算。因为三角形的数字总数是n(n+1)/2

数字三角形的记忆递归型动归程序：

```
#include <iostream>
#include <algorithm>
using namespace std;
#define MAX 101
int D[MAX][MAX];    int n;
int maxSum[MAX][MAX];
int MaxSum(int i, int j){
    if( maxSum[i][j] != -1 )
        return maxSum[i][j];
    if(i==n)   maxSum[i][j] = D[i][j];
    else {
        int x = MaxSum(i+1,j);
        int y = MaxSum(i+1,j+1);
        maxSum[i][j] = max(x,y)+ D[i][j];
    }
    return maxSum[i][j];
}
```

```
int main(){
  int i,j;
  cin >> n;
  for(i=1;i<=n;i++)
        for(j=1;j<=i;j++) {
              cin >> D[i][j];
              maxSum[i][j] = -1;
        }
   cout << MaxSum(1,1) << endl;
}
```

递归转成递推

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

4	5	2	6	5

递归转成递推

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

7				
4	5	2	6	5

递归转成递推

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

7	12			
4	5	2	6	5

递归转成递推

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

7	12	10		
4	5	2	6	5

递归转成递推

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

7	12	10	10	
4	5	2	6	5

递归转成递推

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

20				
7	12	10	10	
4	5	2	6	5

递归转成递推

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

20	13			
7	12	10	10	
4	5	2	6	5

递归转成递推

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

20	13	10		
7	12	10	10	
4	5	2	6	5

递归转成递推

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

30				
23	21			
20	13	10		
7	12	10	10	
4	5	2	6	5

“人人为我”递推型动归程序

```
#include <iostream>
#include <algorithm>
using namespace std;
#define MAX 101
int D[MAX][MAX];   int n;
int maxSum[MAX][MAX];
int main()  {
  int i,j;
  cin >> n;
  for(i=1;i<=n;i++)
            for(j=1;j<=i;j++)
                        cin >> D[i][j];
  for( int i = 1;i <= n; ++ i )
             maxSum[n][i] = D[n][i];
  for( int i = n-1; i>= 1;  --i )
             for( int j = 1; j <= i; ++j )
                 maxSum[i][j] = max(maxSum[i+1][j],maxSum[i+1][j+1]) + D[i][j]
  cout << maxSum[1][1] << endl;
}
```

空间优化

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

4	5	2	6	5

没必要用二维maxSum数组存储每一个MaxSum(r,j),只要从底层一行行向上递推，那么只要一维数组maxSum[100]即可,即只要存储一行的MaxSum值就可以。

空间优化

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

7	5	2	6	5

没必要用二维maxSum数组存储每一个MaxSum(r,j),只要从底层一行行向上递推，那么只要一维数组maxSum[100]即可,即只要存储一行的MaxSum值就可以。

空间优化

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

7	12	2	6	5

没必要用二维maxSum数组存储每一个MaxSum(r,j),只要从底层一行行向上递推，那么只要一维数组maxSum[100]即可,即只要存储一行的MaxSum值就可以。

空间优化

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

7	12	10	6	5

没必要用二维maxSum数组存储每一个MaxSum(r,j),只要从底层一行行向上递推，那么只要一维数组maxSum[100]即可,即只要存储一行的MaxSum值就可以。

空间优化

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

7	12	10	10	5

没必要用二维maxSum数组存储每一个MaxSum(r,j),只要从底层一行行向上递推，那么只要一维数组maxSum[100]即可,即只要存储一行的MaxSum值就可以。

空间优化

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

20	12	10	10	5

没必要用二维maxSum数组存储每一个MaxSum(r,j),只要从底层一行行向上递推，那么只要一维数组maxSum[100]即可,即只要存储一行的MaxSum值就可以。

空间优化

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

20	13	10	10	5

没必要用二维maxSum数组存储每一个MaxSum(r,j),只要从底层一行行向上递推，那么只要一维数组maxSum[100]即可,即只要存储一行的MaxSum值就可以。

空间优化

进一步考虑，连maxSum数组都可以不要，直接用D的第n行替代maxSum即可。

节省空间，时间复杂度不变

空间优化

```
#include <iostream>
#include <algorithm>
using namespace std;
#define MAX 101
int D[MAX][MAX];
int n; int * maxSum;
int main(){
  int i,j;
  cin >> n;
  for(i=1;i<=n;i++)
        for(j=1;j<=i;j++)
              cin >> D[i][j];
  maxSum = D[n]; //maxSum指向第n行
  for( int i = n-1; i>= 1;  --i )
        for( int j = 1; j <= i; ++j )
          maxSum[j] = max(maxSum[j],maxSum[j+1]) + D[i][j];
   cout << maxSum[1] << endl;
}
```

递归到动规的一般转化方法

- 递归函数有n个参数，就定义一个n维的数组，数组的下标是递归函数参数的取值范围，数组元素的值是递归函数的返回值，这样就可以从边界值开始，逐步填充数组，相当于计算递归函数值的逆过程。

动规解题的一般思路

1. 将原问题分解为子问题

- 把原问题分解为若干个子问题，子问题和原问题形式相同或类似，只不过规模变小了。子问题都解决，原问题即解决(数字三角形例）。
- 子问题的解一旦求出就会被保存，所以每个子问题只需求解一次。

动规解题的一般思路

2. 确定状态

- 在用动态规划解题时，我们往往将和子问题相关的各个变量的一组取值，称之为一个“状态”。一个“状态”对应于一个或多个子问题，所谓某个“状态”下的“值”，就是这个“状态”所对应的子问题的解。

动规解题的一般思路

2. 确定状态

所有“状态”的集合，构成问题的“状态空间”。“状态空间”的大小，与用动态规划解决问题的时间复杂度直接相关。在数字三角形的例子里，一共有N×(N+1)/2个数字，所以这个问题的状态空间里一共就有N×(N+1)/2个状态。

整个问题的时间复杂度是状态数目乘以计算每个状态所需时间。

在数字三角形里每个“状态”只需要经过一次，且在每个状态上作计算所花的时间都是和N无关的常数。

动规解题的一般思路

2. 确定状态

用动态规划解题，经常碰到的情况是，K个整型变量能构成一个状态（如数字三角形中的行号和列号这两个变量构成“状态”）。如果这K个整型变量的取值范围分别是N1, N2, ……Nk，那么，我们就可以用一个K维的数组array[N1] [N2]……[Nk]来存储各个状态的“值”。这个“值”未必就是一个整数或浮点数，可能是需要一个结构才能表示的，那么array就可以是一个结构数组。一个“状态”下的“值”通常会是一个或多个子问题的解。

动规解题的一般思路

3. 确定一些初始状态（边界状态）的值

以“数字三角形”为例，初始状态就是底边数字，值就是底边数字值。

动规解题的一般思路

4. 确定状态转移方程

定义出什么是“状态”，以及在该“状态”下的“值”后，就要找出不同的状态之间如何迁移——即如何从一个或多个“值”已知的“状态”，求出另一个“状态”的“值”(“人人为我”递推型)。状态的迁移可以用递推公式表示，此递推公式也可被称作“状态转移方程”。

数字三角形的状态转移方程:

$$
\text{MaxSum}[r][j] = \begin{cases} D[r][j] & r = N \\ \text{Max}\{\text{MaxSum}[r+1][j],\ \text{MaxSum}[r+1][j+1]\} + D[r][j] & \text{其他情况} \end{cases}
$$

能用动规解决的问题的特点

1) 问题具有最优子结构性质。如果问题的最优解所包含的子问题的解也是最优的，我们就称该问题具有最优子结构性质。

2) 无后效性。当前的若干个状态值一旦确定，则此后过程的演变就只和这若干个状态的值有关，和之前是采取哪种手段或经过哪条路径演变到当前的这若干个状态，没有关系。