程序设计与算法(二)

郭 炜

动态规划(一)

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

```
#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。

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。

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:

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

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。

例题二：最长上升子序列(百练2757)

问题描述

一个数的序列ai，当$a_1 < a_2 < ... < a_S$的时候，我们称这个序列是上升的。对于给定的一个序列(a_1, a_2, ..., a_N)，我们可以得到一些上升的子序列(a_{i1}, a_{i2}, ..., a_{iK})，这里1 <= i1 < i2 < ... < iK <= N。比如，对于序列(1, 7, 3, 5, 9, 4, 8)，有它的一些上升子序列，如(1, 7), (3, 4, 8)等等。这些子序列中最长的长度是4，比如子序列(1, 3, 5, 8).

你的任务，就是对于给定的序列，求出最长上升子序列的长度。

输入数据

输入的第一行是序列的长度N (1 <= N <= 1000)。第二行给出序列中的N个整数，这些整数的取值范围都在0到10000。

输出要求

最长上升子序列的长度。

输入样例

7

1 7 3 5 9 4 8

输出样例

4

解题思路

1. 找子问题

“求序列的前n个元素的最长上升子序列的长度”是个子问题，但这样分解子问题，不具有“无后效性”

假设F(n) = x，但可能有多个序列满足F(n) = x。有的序列的最后一个元素比 a_{n+1}小，则加上a_{n+1}就能形成更长上升子序列；有的序列最后一个元素不比a_{n+1}小……以后的事情受如何达到状态n的影响，不符合“无后效性”

解题思路

1. 找子问题

“求以a_k（k=1，2，3…N）为终点的最长上升子序列的长度”

一个上升子序列中最右边的那个数，称为该子序列的“终点”。

虽然这个子问题和原问题形式上并不完全一样，但是只要这N个子问题都解决了，那么这N个子问题的解中，最大的那个就是整个问题的解。

2. 确定状态：

子问题只和一个变量-- 数字的位置相关。因此序列中数的位置k 就是“状态”，而状态 k 对应的“值”，就是以a_k做为“终点”的最长上升子序列的长度。

状态一共有N个。

3. 找出状态转移方程：

maxLen（k）表示以a_k做为“终点”的最长上升子序列的长度那么：

初始状态：maxLen（1）= 1

maxLen（k）= max｛maxLen（i）：1<=i < k 且 a_i < a_k且 k≠1 ｝+ 1

若找不到这样的i，则maxLen(k) = 1

maxLen(k)的值，就是在a_k左边，“终点”数值小于a_k，且长度最大的那个上升子序列的长度再加1。因为a_k左边任何“终点”小于a_k的子序列，加上a_k后就能形成一个更长的上升子序列。

“人人为我”递推型动归程序

```
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;
const int  MAXN =1010;
int a[MAXN];   int maxLen[MAXN];
int main() {
       int N;          cin >> N;
       for( int i = 1;i <= N;++i) {
              cin >> a[i];  maxLen[i] = 1;
       }
       for( int i = 2; i <= N; ++i) {
       //每次求以第i个数为终点的最长上升子序列的长度
              for( int j = 1; j < i; ++j)
              //察看以第j个数为终点的最长上升子序列
                     if( a[i] > a[j] )
                            maxLen[i] = max(maxLen[i],maxLen[j]+1);
       }
       cout << * max_element(maxLen+1,maxLen + N + 1 );
       return 0;
} //时间复杂度O(N²)
```

动归的常用两种形式

1）递归型

优点：直观，容易编写

缺点：可能会因递归层数太深导致爆栈，函数调用带来额外时间开销。无法使用滚动数组节省空间。总体来说，比递推型慢。

1）递推型

效率高，有可能使用滚动数组节省空间

例三、最长公共子序列（POJ1458)

给出两个字符串，求出这样的一个最长的公共子序列的长度：子序列中的每个字符都能在两个原串中找到，而且每个字符的先后顺序和原串中的先后顺序一致。

最长公共子序列

Sample Input

abcfbc abfcab
programming contest
abcd mnp

Sample Output

4
2
0

最长公共子序列

输入两个串s1,s2,

设MaxLen(i,j)表示:

s1的左边i个字符形成的子串，与s2左边的j个字符形成的子串的最长公共子序列的长度(i,j从0开始算）

MaxLen(i,j) 就是本题的“状态”

假定 len1 = strlen(s1),len2 = strlen(s2）

那么题目就是要求 MaxLen(len1,len2)

最长公共子序列

显然：

MaxLen(n,0) = 0 (n= 0…len1)

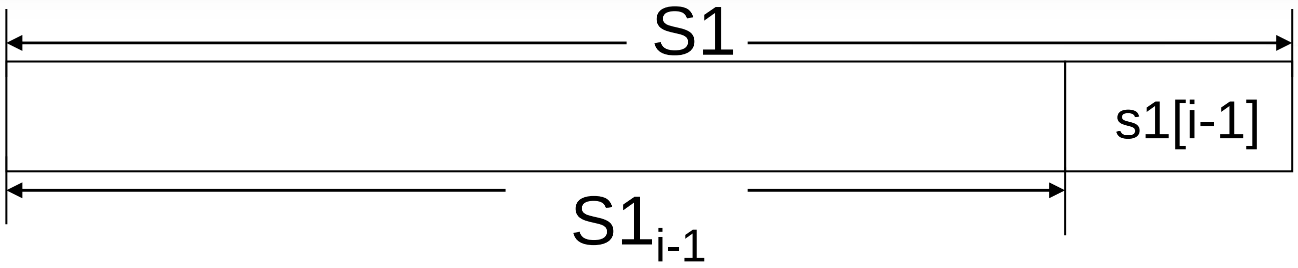
MaxLen(0,n) = 0 (n=0…len2)

递推公式：

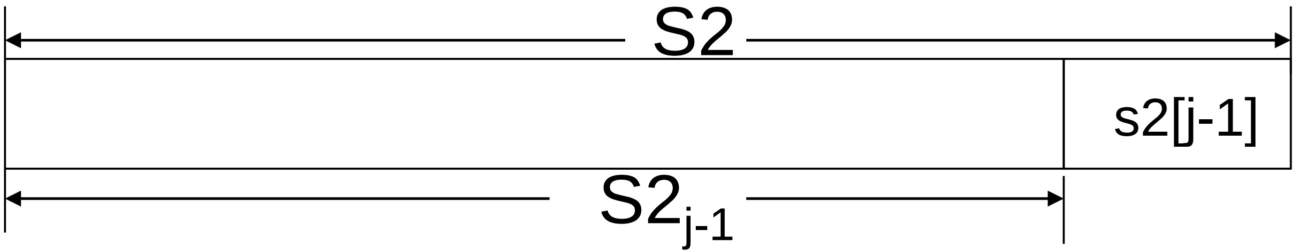
```
if ( s1[i-1] == s2[j-1] ) //s1的最左边字符是s1[0]
    MaxLen(i,j) = MaxLen(i-1,j-1) + 1;
else
    MaxLen(i,j) = Max(MaxLen(i,j-1),MaxLen(i-1,j) );
```

时间复杂度O(mn) m,n是两个字串长度

S1长度为 i



S2长度为 j



S1[i-1]!= s2[j-1]时，MaxLen(S1,S2)不会比MaxLen(S1,$S2_{j-1}$)和MaxLen($S1_{i-1}$,S2)两者之中任何一个小，也不会比两者都大。

```
#include <iostream>
#include <cstring>
using namespace std;
char sz1[1000];
char sz2[1000];
int maxLen[1000][1000];
int main() {
        while( cin >> sz1 >> sz2 ) {
                int length1 = strlen( sz1);
                int length2 = strlen( sz2);
                int nTmp;
                int i,j;
                for( i = 0;i <= length1; i ++ )
                        maxLen[i][0] = 0;
                for( j = 0;j <= length2; j ++ )
                        maxLen[0][j] = 0;
```

```
        for( i = 1;i <= length1;i ++ )  {
            for( j = 1; j <= length2; j ++ ) {
                if( sz1[i-1] == sz2[j-1] )
                     maxLen[i][j] = maxLen[i-1][j-1] + 1;
                else
                    maxLen[i][j] = max(maxLen[i][j-1],
                       maxLen[i-1][j]);
                }
        }
        cout <<  maxLen[length1][length2] << endl;
    }
    return 0;
}
```

活学活用

- 掌握递归和动态规划的思想，解决问题时灵活应用

例四、最佳加法表达式

有一个由1..9组成的数字串.问如果将m个加号插入到这个数字串中,在各种可能形成的表达式中，值最小的那个表达式的值是多少

解题思路

假定数字串长度是n，添完加号后,表达式的最后一个加号添加在第 i 个数字后面，那么整个表达式的最小值，就等于在前 i 个数字中插入 m－1 个加号所能形成的最小值，加上第 i＋1到第 n 个数字所组成的数的值（i从1开始算）。

解题思路

设V(m,n)表示在n个数字中插入m个加号所能形成的表达式最小值，那么：

if m = 0

  V(m,n) = n个数字构成的整数

else if n < m + 1

  V(m,n) = ∞

else

  V(m,n) = Min{ V(m-1,i) + Num(i+1,n) } (i = m … n-1)

Num(i,j)表示从第i个数字到第j个数字所组成的数。数字编号从1开始算。此操作复杂度是O(j-i+1)，可以预处理后存起来。

总时间复杂度：$O(mn^2)$.

解题思路

总时间复杂度：$O(mn^2)$.

若 n 比较大，long long 不够存放运算过程中的整数，则需要使用高精度计算（用数组存放大整数，模拟列竖式做加法），复杂度为$O(mn^3)$