数据结构和算法
（Python描述）

郭 炜

微信公众号

微博：http://weibo.com/guoweiofpku

学会程序和算法，走遍天下都不怕!

讲义照片均为郭炜拍摄

动态规划

例题
数字三角形

加拿大班芙国家公园

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:

```
n = int(input())
D = []
def MaxSum(i,j):
    if i == n-1:
        return D[i][j]
    x = MaxSum(i+1,j)
    y = MaxSum(i+1,j+1)
    return max(x,y) + D[i][j]
def main():
    for i in range(n):
        lst = list(map(int,input().split()))
        D.append(lst)
    print(MaxSum(0,0))
main()
```

为什么超时?

- 回答：重复计算

7_{1}

3_{1} 8_{1}

8_{1} 1_{2} 0_{1}

2_{1} 7_{3} 4_{3} 4_{1}

4_{1} 5_{4} 2_{6} 6_{4} 5_{1}

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规程序:


```
n = int(input())
D = []
maxSum = [[-1 for j in range(i+1)] for i in range(n)]
def MaxSum(i,j):
    if i == n-1:
        return D[i][j]
    if maxSum[i][j] != -1:
        return maxSum[i][j]
    x = MaxSum(i+1,j)
    y = MaxSum(i+1,j+1)
    maxSum[i][j] = max(x,y) + D[i][j]
    return maxSum[i][j]
for i in range(n):
        lst = list(map(int,input().split()))
        D.append(lst)
print(MaxSum(0,0))
```

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

递推程序


```
n = int(input())
D = []
maxSum = [[-1 for j in range(i+1)] for i in range(n)]
def main():
    for i in range(n):
        lst = list(map(int,input().split()))
        D.append(lst)
    for i in range(n):
        maxSum[n-1][i] = D[n-1][i]
    for i in range(n-2,-1,-1):
        for j in range(0,i+1):
            maxSum[i][j] =
          max(maxSum[i+1][j],maxSum[i+1][j+1]) + D[i][j]
    print(maxSum[0][0])

main()
```

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

空间优化后的程序

```
n = int(input())
D = []
def main():
    for i in range(n):
        lst = list(map(int,input().split()))
        D.append(lst)
    maxSum = D[n-1]
    for i in range(n-2,-1,-1):
        for j in range(0,i+1):
            maxSum[j] = max(maxSum[j],maxSum[j+1]) + D[i][j]
    print(maxSum[0])

main()
```

动规解题的一般思路

加拿大班芙国家公园

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:

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

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
最长上升子序列

加拿大班芙国家公园

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$a_1 < a_2 < \ldots < a_S$的时候，我们称这个序列是上升的。对于给定的一个序列($a_1, a_2, \ldots, a_N$)，我们可以得到一些上升的子序列($a_{i1}, a_{i2}, \ldots, a_{iK}$)，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。

最长上升子序列

```
N = int(input())
maxLen = [1 for i in range(N+10)]
a = list(map(int,input().split()))
for i in range(1,N):
#每次求以第i个数为终点的最长上升子序列的长度
    for j in range(0,i):
        #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)
print(max(maxLen))

#时间复杂度O(N²)
```

动规的常用两种形式

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

例题
最长公共子序列

荷兰阿姆斯特丹库肯霍立夫公园

例三、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)

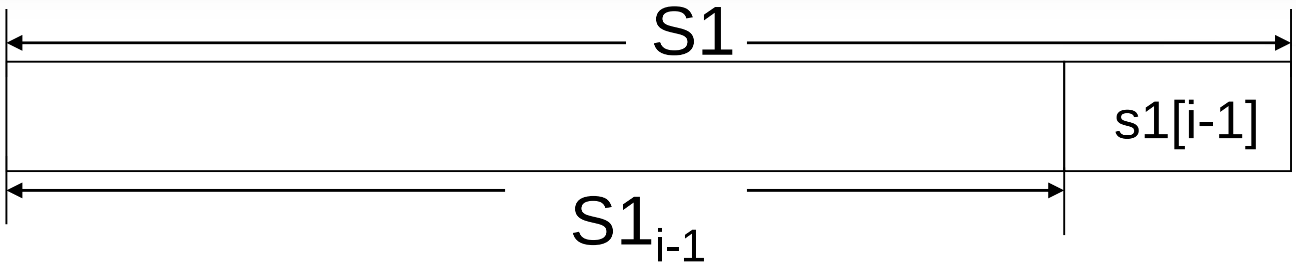
MaxLen(0,n) = 0 (n=0…len2)

递推公式:

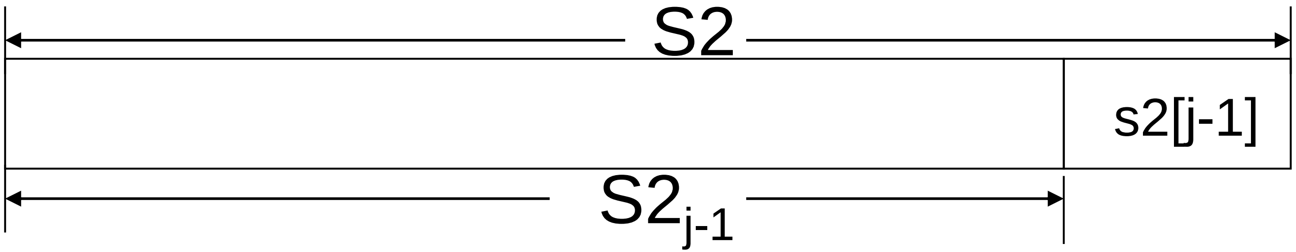
```
if ( s1[i-1] == s2[j-1] ) //s1的最左边字符是s1[0]
    MaxLen(i,j) = MaxLen(i-1,j-1) + 1;
else
    MaxLen(i,j) = Max(MaxLen(i,j-1),MaxLen(i-1,j) );
```

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。

```
mx = 1010
maxLen = [[0 for j in range(mx)] for i in range(mx)]
while True:
    try:
        [s1,s2] = input().split()
        length1,length2 = len(s1),len(s2)
        for i in range(length1+1):
            maxLen[i][0] = 0
        for j in range(length2+1):
            maxLen[0][j] = 0
        for i in range(1,length1+1):
            for j in range(1,length2+1):
                if s1[i-1] == s2[j-1]:
                    maxLen[i][j] = maxLen[i-1][j-1] + 1
                else:
                    maxLen[i][j] = max(maxLen[i][j-1],maxLen[i-1][j])
        print( maxLen[length1][length2])
    except:
        break
```

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

例题
最佳加法表达式

波兰华沙老城

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(否则总复杂度变为O(mn^3)。

总时间复杂度：O(mn^2) .

解题思路

总时间复杂度：$O(mn^2)$

若运算过程中生成的整数比较大，64个bit(假设电脑是64位）不够存放运算过程中的整数，则需要使用高精度计算（用列表存放大整数，模拟列竖式做加法），复杂度为$O(mn^2\times$整数的二进制表示法的bit数$)$

例题
神奇的口袋

圣彼得堡阿芙乐尔号巡洋舰

例五、神奇的口袋(百练2755)

- 有一个神奇的口袋，总的容积是40，用这个口袋可以变出一些物品，这些物品的总体积必须是40。
- John现在有n（$1 \le n \le 20$）个想要得到的物品，每个物品的体积分别是a_1，a_2……a_n。John可以从这些物品中选择一些，如果选出的物体的总体积是40，那么利用这个神奇的口袋，John就可以得到这些物品。现在的问题是，John有多少种不同的选择物品的方式。

- **输入**

输入的第一行是正整数n (1 <= n <= 20)，表示不同的物品的数目。接下来的n行，每行有一个1到40之间的正整数，分别给出a_1，a_2……a_n的值。

- **输出**

输出不同的选择物品的方式的数目。

输入样例

3
20
20
20

输出样例

3

枚举的解法：

枚举每个物品是选还是不选，共2^{20}种情况

递归解法


```
def Ways( w , k ):  # 从前k种物品中选择一些，凑成体积w的做法数目
        if w == 0:
                return 1
        if k == 0:
                return 0
        if w >= a[k]:
                return Ways(w, k -1 ) + Ways(w - a[k], k - 1 )
        else:
                return Ways(w, k -1 )

N = int( input() )
a = [ 0 ]
for i in range(N):
        a.append( int(input()))
print( Ways(40,N) )
```

动规解法

```
N = int( input() )
Ways = [ [ 0 for i in range(50) ] for i in range(50) ]
# Ways[i][j]表示从前j种物品里凑出体积i的方法数
a = [ 0 ]
for i in range( 1, N+1 ):
    a.append( int(input()) )
    Ways[0][i] = 1
Ways[0][0] = 1
for w in range( 1, 41 ):
    for k in range( 1, N+1 ):
        Ways[w][k] = Ways[w][k-1]
        if w-a[k] >= 0:
            Ways[w][k] += Ways[w-a[k]][k-1]
print( Ways[40][N] )
```

例题
Charm Bracelet

美国圣地亚哥中途岛号航母

例六、 Charm Bracelet 0-1背包问题(POJ3624)

有N件物品和一个容积为M的背包。第i件物品的体积w[i]，价值是d[i]。求解将哪些物品装入背包可使价值总和最大。每种物品只有一件，可以选择放或者不放
(N<=3500,M <= 13000)。

0-1背包问题(POJ3624)

用 F[i][j] 表示取前i种物品，使它们总体积不超过j的最优取法取得的价值总和。要求F[N][M]

边界：if (w[1] <= j)

F[1][j] = d[1];

else

F[1][j] = 0;

0-1背包问题(POJ3624)

用 F[i][j] 表示取前i种物品，使它们总体积不超过j的最优取法取得的价值总和

递推： F[i][j] = max(F[i-1][j],F[i-1][j-w[i]]+d[i])

取或不取第 i种物品，两者选优

(j-w[i] >= 0才有第二项)

0-1背包问题(POJ3624)

F[i][j] = max(F[i-1][j],F[i-1][j-w[i]]+d[i])

本题如用记忆型递归，需要一个很大的二维数组，会超内存。注意到这个二维数组的下一行的值，只用到了上一行的正上方及左边的值，因此可用滚动数组的思想，只要一行即可。即可以用一维数组递推实现。

例题:滑雪

圣彼得堡炮兵博物馆

例七、滑雪(百练1088)

Michael喜欢滑雪百这并不奇怪， 因为滑雪的确很刺激。

可是为了获得速度，滑的区域必须向下倾斜，而且当你滑到坡底，

你不得不再次走上坡或者等待升降机来载你。

Michael想知道载一个区域中最长的滑坡。区域由一个二维数组给出。数组的每个数字代表点的高度。下面是一个例子

1 2 3 4 5
16 17 18 19 6
15 24 25 20 7
14 23 22 21 8
13 12 11 10 9

一个人可以从某个点滑向上下左右相邻四个点之一，当且仅当高度减小。在上面的例子中，一条可滑行的滑坡为24-17-16-1。当然25-24-23-...-3-2-1更长。事实上，这是最长的一条。输入输入的第一行表示区域的行数R和列数C(1 <= R,C <= 100)。下面是R行，每行有C个整数，代表高度h，0<=h<=10000。输出输出最长区域的长度。

输入

输入的第一行表示区域的行数R和列数C(1 <= R,C <= 100)。下面是R行，每行有C个整数，代表高度h，0<=h<=10000。

输出

输出最长区域的长度。

样例输入

```
5 5
1  2  3  4  5
16 17 18 19 6
15 24 25 20 7
14 23 22 21 8
13 12 11 10 9
```

样例输出

```
25
```

解题思路

L(i,j)表示从点(i,j)出发的最长滑行长度。
一个点(i,j), 如果周围没有比它低的点，L(i,j) = 1

否则

递推公式： L(i,j) 等于(i,j)周围四个点中,比(i,j)低，且L值最大的那个点的L值，再加1

复杂度：$O(n^2)$

解题思路

解法

L(i,j)表示从点(i,j)出发的最长滑行长度。
一个点(i,j), 如果周围没有比它低的点，L(i,j) = 1

将所有点按高度从小到大排序。每个点的 L 值都初始化为1

从小到大遍历所有的点。经过一个点(i,j)时，用递推公式求L(i,j)