Dynamic Programming

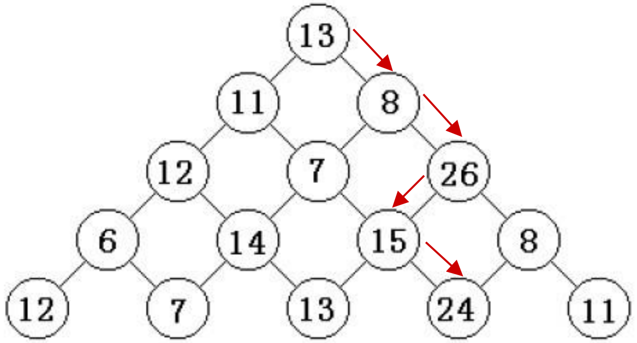
基础动态规划

第1课 概念、术语、基本模型

模型1：数塔

引例：数字金字塔（USACO1.5.1　NKOJ1796）

下面有一数字金字塔。写一个程序来查找从**最高点**到**底部**任意处结束的路径，使路径经过数字的和最大。每一步可以走到**左下方**的点也可以到达**右下方**的点。



在上面的样例中,从13 到 8 到 26 到 15 到 24 的路径产生了最大和

Input

第一个行包含 n(1<= n<=1000) ,表示行的数目。

后面每行为这个数字金字塔特定行包含的整数。

Output

单独的一行,包含那个可能得到的最大的和。

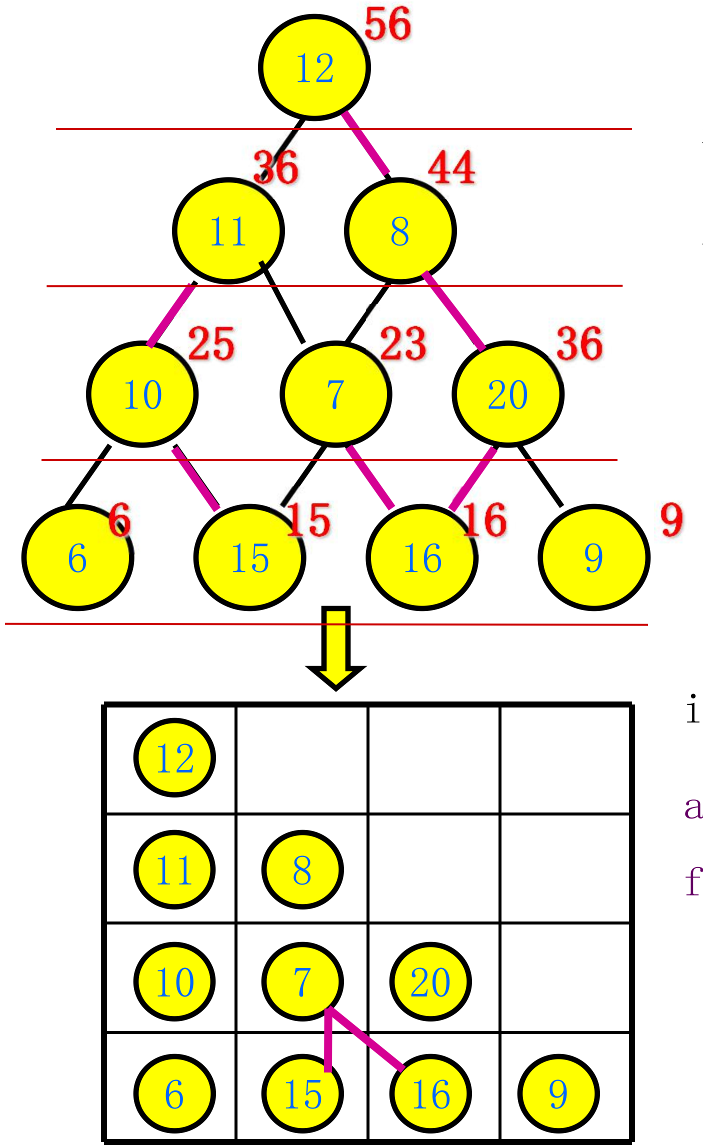
Sample Input

5
13
11 8
12 7 26
6 14 15 8
12 7 13 24 11

Sample Output

86

一个n（n<=1000）层的数字三角形， 请找出从顶层至底层的一条路径，使该路径所经过的数字的总和最大。



分析：

层数n	路径数p
n=1	p=1
n=2	p=2
n=3	p=4
n=4	p=8
…	…
n=k	$p=2^{k-1}$

效率太低

不可取

换个角度思考：

我们由下往上找一条数字总和最大的道路。

由分析得出：求从点t到底层的最大路径值f(t)，

f(t)=t号点的值+max(f(左下)，f(右下))

```
int  a[1001][1001],f[1001][1001];
```

a数组：a[x][y]存点(x,y)本身的数值

f数组：f[x][y]存从底层往上达到点(x,y)能得到的最大数字和

f[x][y]= a[x][y]+max{ f[x+1][y],f[x+1][y+1] }

数字金字塔 参考代码:

```
int  f[1001][1001],a[1001][1001],n,x,y;
int main()
{      //输入数据
       scanf("%d",&n);
       for(x=1;x<=n;x++)
           for(y=1;y<=x;y++)scanf("%d",&a[x][y]);

       //初始化将f数组，全部置为0
       for(x=0;x<=1000;x++)
           for(y=0;y<=1000;y++)f[x][y]=0;

       //动规
       for(x=n;x>=1;x- -)//从最底层开始向上一层一层讨论
          for(y=1;y<=x;y++)
                 f[x][y]=a[x][y]+max(f[x+1][y],f[x+1][y+1]);

      //输出结果
      printf("%d\n",f[1][1]);  //坐标为(1,1)的点位于最顶层
      return 0;
}
```

时间复杂度？$O(n^2)$

什么是“动态规划”？

20世纪50年代初美国数学家R.E.Bellman(理查德 .贝尔曼)**等人在研究多阶段决策过程**(multistep decision process) **的优化问题时，提出了著名的最优化原理**(principle of optimality)**，把多阶段过程转化为一系列单阶段问题，逐个求解，创立了解决这类过程优化问题的新方法——动态规划(Dynamic Programming)。**

南加州大学教授
美国艺术与科学研究院研究员
美国国家工程院院士
美国国家科学院院士

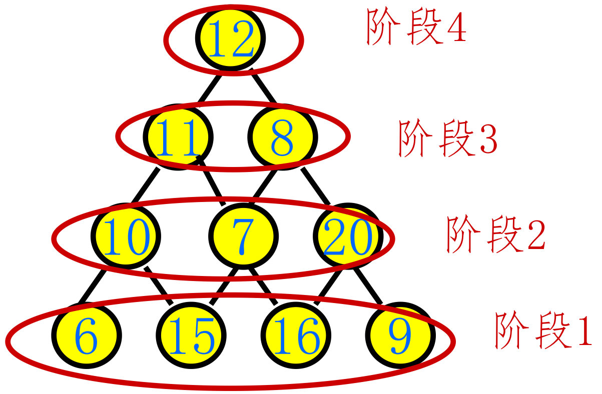
“决策过程和控制系统理论方面做出突出贡献，特别是动态规划的发明和应用。”

还有图论最短路的“bellman-ford算法”

Richard Bellman 1920—1984

1957年出版了他的名著《Dynamic Programming》，这是该领域的第一本著作

数字金字塔的动规过程



解题总结：

1. 首先将问题分成了n个有次序的阶段（多阶段）；

2. 计算每个阶段中，从底层到每个点的最优值（逐个求解）。

3. 在计算底层到某个点的最优值时，是从前一个阶段的两个值中选取的最大的一个（最优化原理）。

(第i阶段的点的最优值是从第i-1阶段的点得到的，与i+1，i+2...阶段无关)

动态规划的特点：

把求最底层到顶点的最优值这个大问题(主问题)分解成了：求最底层到达每个点的最优值。

把一个大的问题分解成了若干个相似的小问题(也称为子问题)。按阶段逐个求解子问题的最优解，最终得到主问题的最优解。

动态规划的优势

动态规划比穷举具有较少的计算次数

从数塔问题可以看出，层数为k时，

穷举算法求路径的条数2^{k-1}

动态规划计算的次数为：$\dfrac{k(k+1)}{2}$

该题k最大为1000

模型2：最大连续和

例1：最大连续和　NKOJ1043

　　有一条很长的面包，该面包被分成了n段(1<=n<=100,000)，每段的味道可能不同。有的味道何老板很喜欢，有的味道何老板很讨厌，于是，何老板事先给每段面包的味道做了美味值打分。

　　美味值越大，表示何老板越喜欢。

　　现在何老板要切下连续一段面包，要求这些面包的美味值总和尽可能大。请你求出这个最大的美味和。

　　例如有一条长度为6的面包，每段的美味值分别是

　　{ -2，11，-4，13，-5，-2 }，

　　其美味值总和最大的一段为红色数字所示

　　{ -2，11，-4，13，-5，-2 }，

　　最大和为20。

例1：最大连续子序列(子串) NKOJ1043

我们可以把问题抽象为下列数学模型：

给定K个整数的序列{ A_1, A_2, ..., A_n }，求其中一段任意长度的连续序列，要求该序列的数字之和最大。

解法一，暴力枚举：

依次讨论从第i个数开始，能得到的最大连续和：

```
for(i=1;i<=n;i++)
{
      Sum=a[i];
      for(j=i+1;j<=n;j++)
      {
            Sum=Sum+a[j];
            if(Sum>ans)ans=Sum;
      }
}
cout>>Sum;
```

时间复杂度$O(n^2)$

n<=100,000 会严重超时

例1：最大连续子序列(子串) NKOJ1043

我们可以把问题抽象为下列数学模型：
给定K个整数的序列{ A_1, A_2, ..., A_n }，求其中一段任意长度的连续序列，要求该序列的数字之和最大。

{ -2, 11, -4, 13, -5, -2 },

方法二，动态规划：

事先把这n个整数依次存到a数组中

确定要求解的状态：f[i]表示以第i个数字作为开头*(包含a[i])***的连续序列的最大和**

按阶段讨论问题：从右到左依次讨论每个数字

做出最佳决策： 如果 f[i+1]<=0，那么f[i]=a[i];
如果 f[i+1]>0， 那么f[i]=f[i+1]+a[i];

最后得出方程：

$$f[i]=\begin{cases} f[i+1]+a[i]; & (\ f[i+1]>0\ ) \\ a[i]; & (\ f[i+1]<=0\ ) \end{cases}$$

边界条件：1<=i<=n

例1：最大连续子序列(子串)　参考代码

方程：$f[i]=\begin{cases} f[i+1]+a[i]; & (f[i+1]>0) \\ a[i]; & (f[i+1]<=0) \end{cases}$

```
f[n]=a[n];                      //DP初始化
for(i=n-1;i>=1;i--)             //依次讨论n个阶段
    if(f[i+1]>0)f[i]=a[i]+f[i+1];
    else f[i]=a[i];
```

时间复杂度O(n)

相似题目：最佳游览(NKOJ 1049)

例1：最大连续子序列(子串) NKOJ1043

你也可以换个方向讨论：

按阶段讨论问题：从左到右依次讨论每个数字

确定要求解的状态：f[i]表示以第i个数字作为结尾的连续序列的最大和

做出最佳决策： 如果 f[i-1]<=0，那么f[i]=a[i];
                如果 f[i-1]>0， 那么f[i]=f[i-1]+a[i];

最后得出方程：

$$f[i]=\begin{cases} f[i-1]+a[i]; & (\ f[i-1]>0\ ) \\ a[i]; & (\ f[i-1]<=0\ ) \end{cases}$$

边界条件：1<=i<=n

模型3：最长上升子序列

例2：最长上升子序列

有N个整数构成的序列,请找出其中长度最长的上升子序列.例如有N=8的序列 3，18，7，14，10，55，12，23

其上升子序列有:

18,55

3,18,23

3,7,10,12,23

……

f[1]=5	f[2]=2	f[3]=4	f[4]=2	f[5]=3	f[6]=1	f[7]=2	f[8]=1
3,	18,	7,	14,	10,	55,	12,	23
1	2	3	4	5	6	7	8

要讨论的**状态**：f[i]表示从第i个数开始能够得到的最长上升序列的长度

按**阶段**讨论问题：从右到左依次讨论每个数字

最佳**决策**：第i个数与它右侧大于i的数中，f[]值最大那一个组合成上升序列

动规**方程**：$f[i]=1+\max\{f[j]\}$　　**边界条件**：i<j<=n,且a[i]<a[j]

例2：最长上升子序列 参考代码

f[i]=1+max(f[j]);

i<j<=n,且a[i]<=a[j]

```
for(i=1;i<=n;i++)f[i]=1;      //初始化DP

for(i=n;i>=1;i--)             //依次讨论每个阶段
    for(k=i+1;k<=n;k++)
        if(a[i]<a[k])&&(f[i]<1+f[k])f[i]=1+f[k];
```

典型例题：拦截导弹(NKOJ 1004)
　　　　　合唱队形(NKOJ 1042)

模型4：分组

例3：最小乘车费用 nkoj1001

Description

某条街上每一公里就有一汽车站，乘车费用如下表：

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

费用 12-- 21-- 31-- 40-- 49-- 58-- 69-- 79-- 90-- 101

而在一辆车上一次最多只能坐10站。某人想行驶n公里，假设他可以任意次换车，请你帮他找到一种乘车方案使费用最小。 编一程序,算出所需最小费用；

Input

第一行为10个不超过100的整数，依次表示行驶1～10公里的费用，相邻两数间用空格隔开；

第二行为某人想要行驶的公里数（1～100公里）。

Output

输出文件仅一行包含一个整数，表示该测试点的最小费用

Sample Input

12 21 31 40 49 58 69 79 90 101

15

Sample Output

147

动规分析：

公里 1 ---2---- 3--- 4----5--- 6 --- 7--- 8 --- 9----10
费用 12-- 21-- 35-- 40-- 49-- 58-- 69-- 79-- 90-- 101

1. 阶段： 按每公里划分阶段

2. 状态： f[i]表示行驶i公里所需最小费用

3. 决策： f[1]=a[1]=12

$$f[2]=\min\begin{cases} f[1]+f[1]=24 \\ a[2]=21 \end{cases} =21$$

$$f[3]=\min\begin{cases} f[1]+f[2]=33 \\ a[3]=35 \end{cases} =33$$

$$f[4]=\min\begin{cases} f[1]+f[3]=45 \\ f[2]+f[2]=42 \\ a[4]=40 \end{cases} =40$$

$$f[5]=\min\begin{cases} f[1]+f[4]=52 \\ f[2]+f[3]=54 \\ a[5]=49 \end{cases} =49$$

$$f[6]=\min\begin{cases} f[1]+f[5]=61 \\ f[2]+f[4]=61 \\ f[3]+f[3]=70 \\ a[6]=58 \end{cases} =58$$

$$f[i]=\min\begin{cases} f[1]+f[i-1] \\ f[2]+f[i-2] \\ f[3]+f[i-3] \\ \ldots\ldots \\ f[i/2]+f[i-i/2] \\ a[i] \qquad i<=10 \end{cases}$$

动规方程：

$$f[i]=\min\{\ a[i]\ ,\ f[j]+f[i-j]\ \}$$

边界： 1<=i<=10， 1<=j<=i/2

$$f[i]=\min\{\ f[j]+f[i-j]\ \}$$

边界： i>10， 1<=j<=i/2

再谈什么是动态规划

将待求解的问题分解为若干个子问题（阶段），按顺序求解子问题，前一子问题的解，为后一子问题的求解提供了有用的信息。

在求解任一子问题时，列出各种可能的局部解，通过决策保留最优的局部解，丢弃其他局部解。依次解决各子问题，最后一个子问题就是初始问题的解。

由于动态规划解决的问题多数有**重叠子问题**这个特点，为减少重复的计算，对每一个子问题只解一次，将其不同阶段的不同状态保存在一个数组中。

动态规划的四大关键字：

1. 阶段：将问题按时间或空间划分成若干个相互联系的阶段，以便按次序求解每个阶段的最优解。 *(数塔问题中，是按层划分的阶段)*

2. 状态：把大的问题分解从若干个相似的子问题，每个子问题称为一个状态。

数塔问题中，把“求从底层到达顶点的最优值”这个大问题分解成“求从底层到达每个点的最优值”，每一个点的最优值，对应了一个子问题。

从底层到点(x,y)的最优值这就是状态，我们用数组f[x][y]来记录这个状态的值。

每个阶段通常包含若干个状态点。

第i阶段的某个状态是由前面阶段的哪个状态转移而来，需要做出决策。

3. 决策：每个阶段的状态都是由前面阶段的状态以某种方式“转化”而来，这种转化往往需要在多种方案中做出最优的选择，我们称之为决策。

数塔问题中，从底层到(x,y)的最优值是从底层到(x+1,y)和底层到(x+1,y+1)的这两个值中，选择最大者转化得到的。

4. 方程： 当确定好解决问题的阶段、状态和决策后，将相邻两个阶段状态转移的规律数学化的表示出来，称为**状态转移方程**。

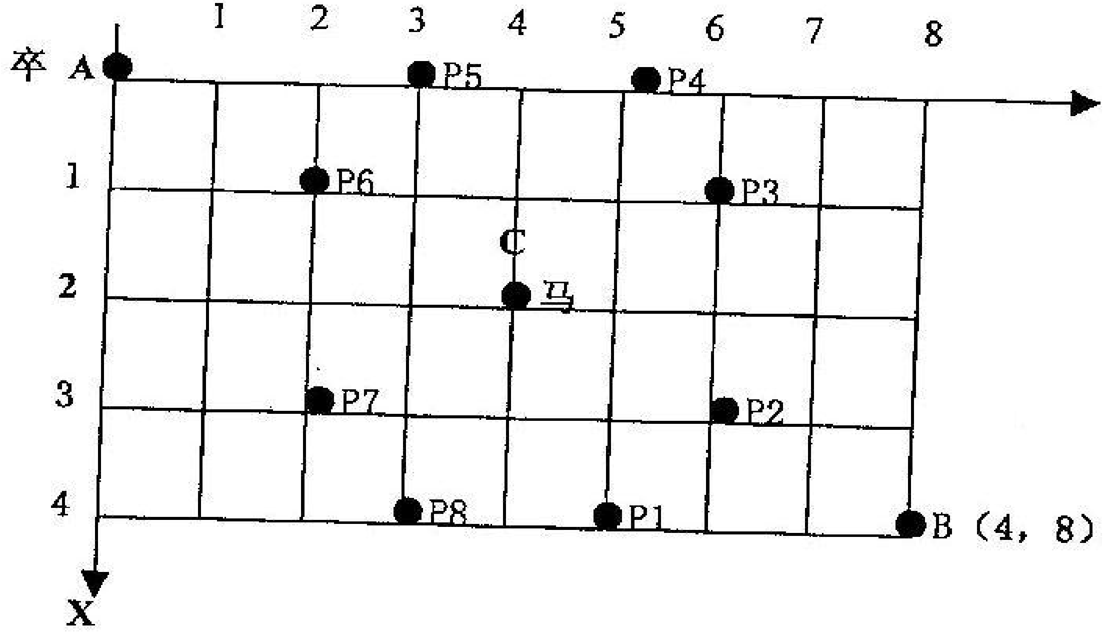
比如：数塔问题中的方程为： f[x][y]=m[x][y]+max(f[x+1][y],f[x+1][y+1])

当前阶段的一个状态　　决策　　前一个阶段的状态

边界条件：给出的状态转移方程是一个递推式，需要一个递推的开始和终止的边界条件。

例4：马拦过河卒（NOIP2002　NKOJ1050）

如图，A点有一个卒，需要走到B点。卒行走规则：可以**向下**、或者**向右**。同时在棋盘上有一个对方的马（如上图的C点），该马所在的点和所有跳跃一步可达的点称为马的控制点。例如上图C点上的马可以控制 9个点（图中的P1，P2 … P8和 C）。卒不能通过对方马的控制点。



棋盘用坐标表示，A点（0，0）、B点（n,m）(n,m为不超过 17的整数)，同样马的位置坐标是题目给出的（约定：C!=A，同时C!=B）。现在要求你计算出卒从 **A点能够到达 B点的路径的条数**。

申明数组 int f[18][18];

f[x][y]记录卒从起点走到坐标点(x,y)可行的路径总数。

初始化：memset(f,0,sizeof(f));　　f[0][0]=1;

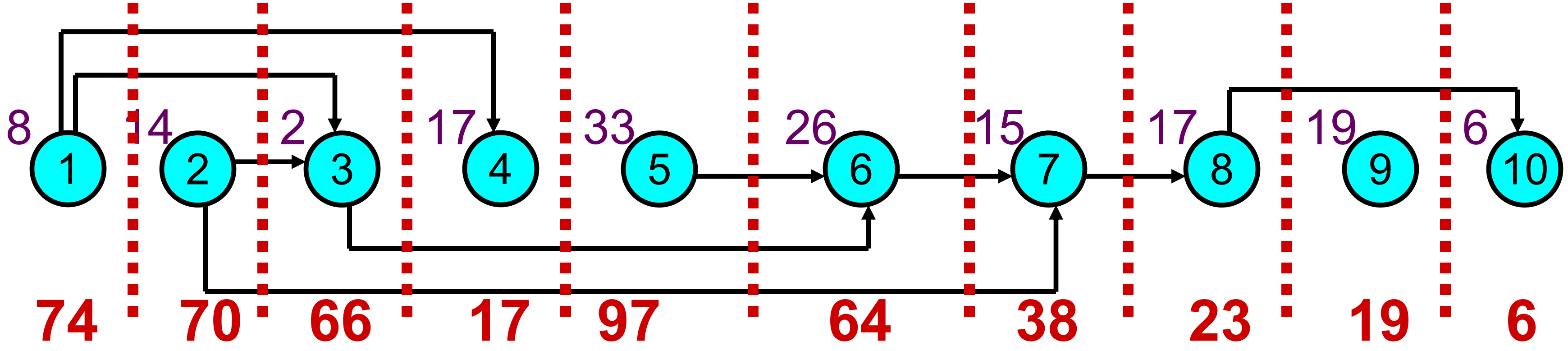
讨论:若点(x,y)不可通过(被马控制)，那么f[x][y]=0。

若点(x,y)可通过，那么**f[x][y]= f[x-1][y]+f[x][y-1]**

注意边界，讨论时，坐标(x,y),(x,y-1),(x-1,y)不能越处棋盘的边界

是动规？　不是！缺乏动规三要素中的"决策",只能算是"递推"

例5. 挖地雷：公路上有n堆（<=200）地雷，从左往右编号1到n。假如n=10时，每堆地雷的情况如下图，图中的箭头表明了地雷间的联系，比如①有箭头指向③和④，表明从①开始，挖完后可继续挖③或④的地雷（存在多种方案时只能选其中一种），只能选一堆地雷开始挖，求最多可以挖多少颗地雷。



分析：

阶段：以每一堆雷划分一个阶段。

状态：f[i]表示从第i堆开始，最多能挖的地雷个数。

决策：当有多条路可走始，选择已挖出地雷最多的那一条。

int a[n],f[n];

f[x]=a[x]+max{ f[y] }　y为x直接连接到的下一堆地雷的编号

思考：上述解法是合理？

必须增加条件：箭头只能从编号小的指向编号大的！

动态规划需满足的基本原则：**无后效性**

第i个阶段的状态只能由前面阶段的状态转移得来，与后面阶段无关。——状态无后效性

比如数塔问题中，状态f[x][y]，中x代表了层数，也就是阶段。f[x][y]是由f[x+1][y]和f[x+1][y+1]两个状态决策而来的，而这两个状态都是位于之前已经计算过第x+1层。而第x-1,x-2,...,1这些层的情况是不会影响到已经算好的f[x][y]。

换言之：一旦当前阶段的状态的值确定了以后，不会受后面阶段的状态的影响。

（若要受后面阶段的影响，则有后效性，不能用动规来解决）

动态规划适用的情况：

适用的情况能采用动态规划求解的问题的一般要具有3个性质：

(1) 最优化原理：如果问题的最优解所包含的子问题的解也是最优的，就称该问题具有最优子结构，即满足最优化原理。

(2) 无后效性：即某阶段状态一旦确定，就不受这个状态以后决策的影响。也就是说，某状态以后的过程不会影响以前的状态，只与当前状态有关。

(3)有重叠子问题：即子问题之间是不独立的，一个子问题在下一阶段决策中可能被多次使用到。（该性质并不是动态规划适用的必要条件，但是如果没有这条性质，动态规划算法同其他算法相比就不具备优势）

今天你学到的动规模型有：

1.数塔
2最大连续和
3最长上升子序列
4分组

本课习题：

1796数字金字塔，
1043最大连续子序列
1001最小乘车费用

1042合唱队形
1049最佳游览