目　录

第一部分　常用算法

第二部分　数据结构

第一部分　常用算法

在计算机程序设计中讨论算法的目的则是将其作为编写程序的依据，它是软件设计的基础。算法的好坏，将影响着软件的质量，因此研究算法对提高软件的质量起着很重要的作用。本章将就程序设计中常见的典型算法做一些介绍。

第一章　基础题

奥林匹克信息学竞赛十分强调基础知识。每年的分区联赛（NOIP）都含一些直接考核选手是否会编程的基础题，而每年的全国赛（NOI）、组队赛（CTSC）或国际赛（IOI）对灵活应用基础知识的要求也愈来愈高。因此，在平日训练中既不要因为畏难而放弃大题难题，更不要因为轻视而不屑做基础题。自信心和自知之明、夯实基础和破解难题是对立的统一。我们在做大题难题的时候，为什么效果经常不尽如人意，除了采用算法错误的原因外，更多的时候是因为细节上的一些瑕疵而导致算法的关键地方时效低下，甚至导致整个算法的时间复杂度远远高于正常情况，最为严重的后果是导致正确的算法无法实现。出现“因小失大，功亏一篑”的主观原因是好高骛远、轻视基础。因此我们应该注意从小题、中题练起，积累经验，强化内功，夯实基础。

【例 1】计算多项式

设多项式 exp(x)=1+x+$\frac{x^2}{2!}+\frac{x^3}{3!}+\cdots\cdots+\frac{x^i}{i!}$ ($\left|\frac{x^i}{i!}\right|\leqslant 10^{-10}$)

输入 x

输出 exp(x)的值（保留小数点后四位）

题解

设 s—和；t—当前项；i—x 的幂次。即 $s=t_0+t_1+t_2+\cdots\cdots+t_i$

1. 确定重复条件

   abs（t）>1e-10；

2. 确定重复体

   由 $t_i=\frac{x^{i-1}}{(i-1)!}*\frac{x}{i}=t_{i-1}*\frac{x}{i}$ 得出重复体

   i←i+1；

   t←t*$\frac{x}{i}$；

   s←s+t；

3. 设定初值

   i←0；t←1；s←1；

由此得出程序：

```
var
```

```
 i，X：integer;                                          {x 的幂次和自变量}
 t，s：real;                                             {当前项和多项式的值}
 begin
  readln(x);                                             {读自变量的值}
  i：=0;  t：=1;  s：=1;                                 {赋初值}
  while abs(t)>1e-10 do                                  {若当前项未达到精度要求,则新增一项}
   begin
    i：=i+1;
    t：=t*x/i;
    s：=s+t;
   end；{while}
  writeln(' EXP(', X,' )= ', S：0：4);                   {输出多项式的值}
  readln;
 END.{main}
```

【例 2】计算灯的开关状态

有 N 个灯放在一排，从 1 到 N 依次顺序编号。有 N 个人也从 1 到 N 依次编号。1 号将灯全部关闭，2 将凡是 2 的倍数的灯打开；3 号将凡是 3 的倍数的灯作相反处理(该灯如为打开的， 则将它关闭；如关闭的，则将它打开)。以后的人都和 3 号一样，将凡是自己编号倍数的灯作相反处理。试计算第 N 个操作后，哪几盏灯是点亮的。(0-表示灯打开 1-表示灯关闭)

题解

设布尔数组 a 为 n 盏灯的状态。初始时,所有的灯打开,即 a 数组的每一个元素设为 false。然后依次将每盏灯序号的倍数作取反处理，由此得出的 a 数组的序数值即为最后的灯状态。

```
var  k，n，i，j：integer;
    a：array[1..100]of boolean;                          {N 盏灯的状态}
begin
  readln (n);                                            {读入灯的数目}
  for i：=1 to n do a[i]：=false;                        {初始时所有灯打开}
  for i：=1 to n do                                      {依次进行 n 次操作}
   begin
    j：=i;                                               {从第 i 号队员出发进行第 i 次操作}
    while j<=n do
      begin
       a[j]：=not(a[j]); j：=j+i;                        {将凡是 i 的倍数的灯作取反处理}
      end；{while}
    end；{for}
   for i：=1 to n do write(ord(a[i]));                   {输出 n 次操作后的结果}
   writeln;
end.{main}
```

【例 3】计算今天星期几

按照年 月 日的格式输入今天的日期。计算和输出今天是星期几的信息。

题解

设年、月、日为 y、m、d。按照余数公式

$(a_1+a_2+\cdots a_n) \bmod k=(a_1 \bmod k+a_2 \bmod k+\cdots a_n \bmod k) \bmod k$

我们首先计算公元 0000 至去年（y-1）间每年天数对 7 的余数，累计它们的和 y'；然后计算今年 1 月至上月（m-1）的天数对 7 的余数 m'；最后得出(y'+m'+d)mod 7 为今天的星期信息。

1.计算公元 0000 至 y-1 年对 7 的余数和 y'

闰年的天数为 366，对 7 的余数为 2；平年的天数为 365，对 7 的余数为 1。公元 0000 年至去年（y-1）含 $\left\lfloor\frac{y-1}{4}\right\rfloor-\left\lfloor\frac{y-1}{100}\right\rfloor+\left\lfloor\frac{y-1}{400}\right\rfloor$ 个闰年。由此得出公元 0000 至 y-1 年对 7 的余数和为 $y'=y-1+\left\lfloor\frac{y-1}{4}\right\rfloor-\left\lfloor\frac{y-1}{100}\right\rfloor+\left\lfloor\frac{y-1}{400}\right\rfloor$。

2. 计算今年 1 月至上月(m-1)的总天数对 7 的余数 m'

$$m'=\begin{cases}0 & \text{上月为1,10月}\\ 1 & \text{上月为5月}\\ 2 & \text{上月为8月}\\ 3 & \text{上月为2,3,11月}\\ 4 & \text{上月为6月}\\ 5 & \text{上月为9,12月}\\ 6 & \text{上月为4,7月}\end{cases}$$

若今年为闰年(((y mod 4=0) and(y mod 100<>0))or(y mod 400=0))且月份大于 2(m>2)，则 m'=m'+1；

3.根据表达式(y'+m'+d)mod 7 的值，分情形输出今天是星期几的信息

```
var
 y, y0, m, d, m0, s, week: integer;
begin
 readln(y, m, d);                                          {读今天的日期}
 y0:=y-1;                                                  {计算去年的年号}
 y0:=y0+y0 div 4-y0 div 100+y0 div 400;        {计算公元 0000 至去年对 7 的余数和}
 case m of                                  {计算 1 月至 m-1 月的总天数对 7 的余数 m0}
   1, 10:  m0:=0;
      5:  m0:=1;
      8:  m0:=2;
 2, 3, 11:  m0:=3;
      6:  m0:=4;
   9, 12:  m0:=5;
    4, 7:  m0:=6;
```

```
 end; {case}
if(((y mod 4=0)and(y mod 100<>0))or(y mod 400=0))and(m>2) then m0: =m0+1;
case (y0+m0+d) mod 7 of  {根据(y'+m'+d)mod 7 的值，分情形输出今天是星期几的信息}
   0: writeln(' Sunday' );
   1:  writeln(' Monday' );
   2:  writeln(' Tuesday' );
   3:  writeln(' Wednesday' );
   4:  writeln(' Thursday' );
   5:  writeln(' Friday);
   6:  writeln(' Saturday' );
 end; {case}
readln;
end.{main}
```

【例 4】放球

把 m 个球放入编号为 0，1，2，…，k-1 的 k 个盒中($m<2^k$)要求第 i 个盒内必须放 2^i 只球。如果无法满足这一条件，就一个不放，求出放球的具体方案。

题解：

将十进制数 M 化为对应的二进制数。将第 i 位的数码乘以 2^i 即为第 i($0\leqslant i\leqslant \log_2 m$)个盒子应放的球数。例如 M=29，化为二进制数是 11101，$29=2^4+2^3+2^2+1$，即第 0 号盒放 1 只球，1 号盒不放，2 号盒放 4 只球，3 号盒放 8 只球，4 号盒放 16 只球。设

$(A)_2=a_ka_{k-1}\cdots a_1a_0$ $(B)_2=b_kb_{k-1}\cdots b_1b_0$ $(C)_2=c_kc_{k-1}\cdots c_1c_0$ ($0\leqslant a_i$，b_i，$c_i\leqslant 1$，$0\leqslant i\leqslant k$)

turbo.Pascal 提供了五种位运算：

C=not A	按位取反（一元运算），即 $c_i=\overline{a_i}$；
C=A and B	按位与，即 $a_i=b_i=1$ 时，$c_i=1$；否则 $c_i=0$；
C=A or B	按位或，即 a_i 和 b_i 中至少有一个为 1 时，$c_i=1$；否则 $c_i=0$；
C=A xor B	按位异或，即 a_i 和 b_i 相反时，$c_i=1$；否则 $c_i=0$；
C=A shl l	$(A)_2$ 左移 l 位，相当于乘 2^l；
C=A shr l	$(A)_2$ 右移 l 位，相当于整除 2^l；

注意，为了保证位运算的正确性，参与位运算的变量一般为无符号整数（byte 或 word）。如果使用有符号整数（longint 或 integer），则最高位的符号位不能参与运算。例如，A=10，B=7，L=2。

C=not A=$(\overline{1010})_2=(0101)_2=5$，$C=AandB=and\begin{matrix}1010\\0111\\0010\end{matrix}=2$，$C=AorB=or\begin{matrix}1010\\0111\\1111\end{matrix}=15$，

$C=AxorB=xor\begin{matrix}1010\\0111\\1101\end{matrix}=13$。

C=A shl 1=40

0	0	1	0	1	0	0	0	←

C=A shr 1=2

0	0	0	0	0	0	1	0	→1 0

表 3.3.3

显然，我们可以通过(m shr 1)and 1 取出 m 的下一位二进制数码。但问题是 Pascal 不能直接计算 2^i， 我们通过两个函数和相应的换底公式

EXP（x）—计算 e 的 x 次幂 e^x；

Ln（x）—计算以 e 为底的对数；

$2^i=e^{i*\ln(2)}$ =EXP(i*ln(2))；

计算出 2^i。按照由底位至高位的顺序逐位取出$(m)_2$的二进制数码，乘上 2^i（i 为位序号），便可以得出每一个盒子放的球数。计算过程如图 3.3.2 所示：

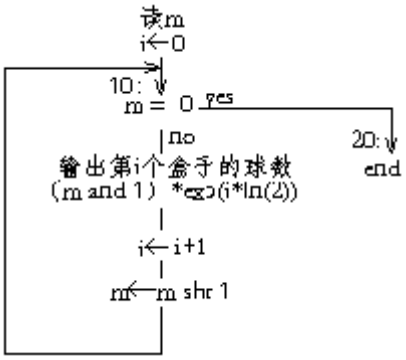


图 3.3.2

```
var
 m，i：longint;                                          {小球数和位序号}
begin
  readln(m);                                                {读小球数}
  i:=0;                                                  {从第 0 个盒子开始计算}
  while m<>0 do                                  {若目前 i 个盒子未放入所有小球，则循环}
   begin
    writeln(i,':', (m and 1)*exp(i*ln(2))：0：0);       {输出第 i 个盒子的球数}
    i:=i+1;                                                  {增加一个盒子}
    m：=m shr 1;                                             {右移一位}
   end; {while}
  readln;
end.{main}
```

【例 5】代码翻译

输入一个以’@’结束的字符串，从左至右翻译。若下一个字符是数字 n(0≤n≤9)，表示后一个字符重复 n+1 次，不论后一个字符是否为数字；若下一个字符非数字，则表示自己。翻译后，以 3 个字符为一组输出，组与组之间用空格分开。例如’A2B5E34FG0ZYWPQ59R@’，翻译成’ABB_BEE_EEE_E44_44F_GZY_WPQ_999_999_R@ ’。

输入

字符串（串长≤255）；

输出

翻译后的字符串；

题解

设 ch 为输入字符；

n 为组内计数器。按照三个译码为一组、组与组之间以空格分隔的输出要求，每输出一个译码，n←(n+1)mod 3。当 n=0 时，一组译码输出完毕，应输出一个空格；

c 为字符重复输出的次数。当 c>'0'时，说明字符 ch 作为译码还应继续输出。c←pred(c)，输出 ch；当 c='0'时，说明译码 ch 输出完毕，重新输入一个字符 ch。若 ch 为数字符，则表示 ch 的后一个字符作为译码，该译码应重复 ord(ch)-ord('0')次，c←ch；若 ch 为字母，则表示 ch 作为译码应该输出一次；

我们在输入字符的同时，按照上述翻译规则进行译码：

```
var   c，ch：char;                                          {重复次数和输入字符}
      n：integer;                                           {组内计数器}
begin
 n：=0;  c：='0';                                          {组内计数器和重复次数初始化}
 read(ch);                                                  {输入字符}
 if ch<>'@'                                                 {若当前字符非结束符}
  then repeat
        if c>'0'                                {若当前字符需要重复输出，则重复次数-1}
         then c：=pred(c)
         else if (ch>='0')and(ch<='9')
                  then begin                    {若输入的字符表示重复次数，则记下}
                       c：=ch;
                       read(ch);                            {输入被重复的字符}
                       end; {then}
        write(ch);                                          {输出译码}
        n：=(n+1)mod 3;                                     {计算组内计数器}
        if n=0 then write('  ');                {若一组输出完毕，则以空格间隔}
        if c='0' then read(ch);{若当前字符完成了重复输出的次数，则读下一个字符}
       until ch='@';                                        {直至读入结束符为止}
 writeln('@');
end.{main}
```

【例 6】字符加密

加密规则是将输入的英文字母下推 K 个顺序后输出。加密工作直至输入一个非英文字母为止。例如 K=5

输入	输出
'a'	'f'
'b'	'g'
……	
'y'	'd'

’Z’ ’E’
’0’ 结束

题解

设输入字符为 ch。首先我们通过布尔表达式（ch in[’a’..’z’,’A’..’Z’]）判断字符 ch 是否为英文字母。如果是，则通过布尔表达式（ch=upcase(ch)）确定该英文字母的大小写，因为下推 K 个顺序的计算只能在所属的大小写范围内进行。ch 下推 K 个顺序后的字母在 26 个英文字母中的顺序为 ord(ch)-ord(’A’)+k)mod 26（或 ord(ch)-ord(’a’)+k）mod 26），加上’A’（或’a’）的 ascll 码（ord(’A’)或 ord(’a’)）即为译码的 ascll 码，通过 chr 函数将其还原为译符。

```
var
 ch:char;                                                    {输入字符}
 k:integer;                                                  {下推的顺序数}
begin
 readln(k);                                                  {读下推的顺序数}
 readln(ch);                                                 {读第 1 个字符}
 while ch in[’a’..’z’,’A’..’Z’]do          {若该字符为英文字母，则循环}
  begin
   write(ch,’:’);
   if ch=upcase(ch)                        {根据 ch 的大小写计算下推 K 个顺序后的译符}
    then ch:=chr((ord(ch)-ord(’A’)+k)mod 26+ord(’A’))
    else ch:=chr((ord(ch)-ord(’a’)+k)mod 26+ ord(’a’));
   writeln(ch);                                              {输出译符}
   readln(ch);                                               {读下一个字符}
  end; {while}
 end.{main}
```

【例 7】计算路程

一次军事演习，A、B 两队约好同一时间从相距 s(50≤s≤100)公里的各自驻地出发相向运动，A 队行进速度为 va 公里/小时(5≤va≤10)，B 队为 vb 公里/小时(4≤vb≤8)。一通迅员骑摩托车从 A 队的驻地也在同一时间为行进中的两队传递信息。摩托车的速度为 vm(30≤vm≤60)60 公里/小时，往返于两队之间，每遇一队立即折回驶向另一队。当两队距离小于 0.5 公里时，摩托车停下来不再传递信息。

输入

s,va,vb,vm

输出

通讯员跑了多少趟。（从一队驶向另一队叫一趟）

题解

s——A.B 两队之间的距离，初值为两地的距离；

va，vb，vm——A 队，B 队和摩托车的行进速度；

j——通讯员所跑的趟数；

t—当前一趟摩托车的费时；

i——通讯员往返于两队的标志。$i=\begin{cases}1 & \text{通讯员从}A\text{队折回驶向}B\text{队}\\ -1 & \text{通讯员从}B\text{队折回驶向}A\text{队}\end{cases}$

i=1，通讯员从 A 队折回驶向 B 队相遇时，满足 s-vb*t=vm*t.即 t=$\frac{s}{vb+vm}$；

i=-1，通讯员从 B 队折回驶向 A 队相遇时，满足 s-va*t=vm*t.即 t=$\frac{s}{va+vm}$；

由上式可以看出，由于 t 为除法运算的结果，而 s 的计算过程中有 t 参与，因此 s 和 t 为实数类型。

```
var    va:5..10;                                  { A 队，B 队，摩托车的行进速度}
       vb:4..8;     vm:30..60;
       i，j：integer;                                {往返标志和趟数}
       s，t：real;                                {两队的距离和当前一趟摩托车的费时}
 begin
  readln(s,va,vb,vm);              {输入两队的距离和 A 队、B 队、摩托车的行进速度}
  i：=1；j：=0;                                {往返标志和趟数初始化}
  repeat
    if i=1 then t：=s/(vb+vm)                {根据往返标志计算当前一趟的时间}
           else t：=s/(va+vm);
    s:=s-(va+vb)*t;                                {计算两队距离}
    j：=j+1;                                {累计趟数}
    i：=i*(-1);                                {往返标志取反}
  until s<=0.5;                              {直至两队距离小于 0.5 公里为止}
  writeln (’ j=’, j);                              {输出趟数}
 end.{main}
```

【例 8】计算数列

求 1/1+1/2+2/3+3/5+5/8+…前 n 项（n≤50）的和（保留小数点后 2 位）。

题解

设 e=$t_1+t_2+\cdots t_i+\cdots t_n$，其中 $t_i=\frac{a_i}{b_i}$（1≤i≤n）。由数列的特征可以看出，$a_i=b_{i-1}$，$b_i=a_{i-1}+b_{i-1}$。由于 n≤50，因此 a_i，b_i 和 e 的数值超过了标准的整数类型和实数类型的范围。不得不通过{$n+}启动浮点运算，并将 a_i，b_i 和 e 的数据类型设为 extended，使得其精度保持在 19 位。

```
{$N+}                                                {启动浮点运算}
var i，n：interger;                                {循环变量和项数}
    a，b，c，e：extended;  {数列的和为 e；当前项的分子为 a，分母为 b；c 为辅助变量}
begin
  readln(n);                                                {读项数}
  a：=0；b：=1；e：=0;                        { 当前项的分子、分母和数列的和初始化}
  for i：=1 to n do                                {累计每一项}
```

```
    begin
     c：=b；b：=a+b；a：=c；                          {计算第 i 项的分子和分母}
     e：=e+a/b；                                      {将第 i 项计入数列和}
    end；{for}
  writeln(e：0：2)；                                  {输出数列和}
end.{main}
```

【例 9】从 m 个不同数中取 n 个不同数的全部组合

输入 m，n(1≤n≤m≤100)

输出所有组合

题解

设组合为 $c_1 \cdots c_n$。要求组合中的 n 个数按照递增顺序排列。c_n 最大为 m，c_{n-1} 最大为 m-1，…,c_1 最大为 m-n+1,即 $c_j \leqslant m-n+j$。计算组合 c 的过程如下：

初始时，$c_1=1$，$c_2=2$，…，$c_n=n$，即 $c_1 \cdots c_n$ 为最小。

从 c_n 出发向右扫描。若 $c_{i+1} \cdots c_n$ 都达到了最大值（$c_j=m-n+j$，j=i+1…n）,$c_i<m-n+i$,则 $c_1 \cdots c_{i-1}$ 保持不变，$c_i=c_i+1$，$c_{i+1}=c_i+1$，$c_{i+2}=c_{i+1}+1$，…,$c_n=c_{n-1}+1$。输出组合 $c_1 \cdots c_n$。例如 1…6 中取 3 个数的组合 256。由于 $c_1=2<4$,因此可按照上述方法计算和输出下一个组合 345。

返回 2、计算下一个组合，直至 $c_1 \cdots c_n$ 都达到了最大值为止。

例如从 1、2、3、4、5 中取 3 个不同数，按照上述算法可得出十个组合：

1 2 3　1 2 4　1 2 5　1 3 4　1 3 5

1 4 5　2 3 4　2 3 5　2 4 5　3 4 5　（$c_i<m-n+i$）

显然当 $c_1 \cdots c_n$ 都达到了最大值时，$c_1=m-n+1$。由此得出算法的终止标志 $c_1=m-n+1$。

```
Var  i,j,k:integer;
     m,n:1..1000;                                    {问题规模}
     b:array[0..100] of integer;                     {组合}
begin
 repeat
    readln(m,n)；                                    {读问题规模}
 until m>n;
 for i:=1 to n do b[i]:=i;                           {设置和输出初始组合}
for i:=1 to n do write(b[i]:5);
 while b[1]<m-n+1 do                                 {若 c1…cn 未全部达到最大值，则循环}
  begin
   j:=n;                                             {由右而左搜索第 1 个未达到最大值的元素 j}
   while b[j]=m-n+j do j:=j-1;
   b[j]:=b[j]+1;                                     {调整元素 j…元素 n}
   for i:=j+1 to n do b[i]:=b[i-1]+1;
   for i:=1 to n do write(b[i]:5);                   {输出当前组合}
   writeln;
  end；{while}
end.{main}
```

【例 10】计算成等差数列的素数

求出 2~n（n≤50）之间长度最长、成等差数列的素数。
例如：2~50 之间的奇数
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
公差为 1： 2， 3 长度为 2
公差为 2： 3 5 7 长度为 3
输入 n
输出：成等差数列的素数

题解

我们首先通过筛选法计算素数集合。设 s 为筛；primes 为素数集合。初始时，s=[2..n]，primes=[]。选取 s 筛中的最小数，作为素数进入 primes 集合，并将该素数的倍数从 s 筛中划去；然后再选取 s 筛中最小数为素数，并将 s 筛中该素数的倍数划去，…，直至 s 筛空为止。此时 primes 集合即存储了 2..n 中的素数。例如表 3．4．1 给出了计算 2—10 间素数的过程：

动作	S 集合	Primes 集合
初始状态	[2，3，4，5，6，7，8，9，10]	[]
2 进入 Primes 集合，将 2 的倍数从 s 筛中划去	[3，5，7，9]	[2]
3 进入 Primes 集合，将 3 的倍数从 s 筛中划去	[5，7]	[2，3]
5，7 进入 Primes 集合	[ ]	[2，3，5，7]

表 3．4．1

然后搜索 primes 集合中的每一个素数 i，分别作为等差数列的第 1 个元素，计算以素数 i 后的每一个元素与素数 i 的间距为公差的数列。显然，按照上述算法，可以得出所有可能的等差数列。其中长度最长的等差数列即为问题的解。

```
var
 s,primes:set of 1..255;                          {s—筛；primes—素数集合}
 f,f1,ff,c:string;
        {f-数列；f1-数列的当前元素；f-f 数列的第一个元素；c-长度最长的等差数列}
 d,p,n,i,j,delta,k,max:integer;
          {P—筛中最小数；d—P 的倍数，应从筛中去掉；delta—公差；max—最大长度}
begin
 readln(n);                                       {读问题规模}
 s:=[2..n]; p:=1; primes:=[];                     {筛、筛中最小数和素数集合初始化}
 repeat
  repeat p:=p+1;  until p in s;                   {选取筛中最小数为素数}
  primes:=primes+[p];                             {P 放入素数表}
  d:=p;
  repeat
   s:=s-[d];d:=d+p;                               {去掉筛中所有 P 的倍数}
  until d>n;
```

```
until s=[];                                  {直到筛空,即 2..N 上的所有素数求出为止}
for i:=2 to n do                             {搜索素数集合中的每一个素数}
 if i in primes
  then begin
       str(i,ff); f:=ff;       {将素数 i 转换成数串 ff,作为等差数列的第一个元素}
       for j:=i+1 to n do                    {顺序搜索 i 后的每一个素数}
        if j in primes
         then begin
              delta:=j-i;  k:=delta;                              {计算公差}
              while(i+k<=n)and((i+k) in primes)do
               begin  {将公差为 delta 的素数(i+k,对应的数串为 f1)送入等差数列}
                str(i+k,f1); f:=f+ ' ' +f1; k:=k+delta;
               end; {while}
              if length(f)>max                {若等差数列的长度最长，则记下}
                 then begin max:=length(f);  c:=f;  end; {then}
              f:=ff;                  {继续搜索由素数 i 开始的、公差更大的数列}
              end; {then}
        end;
 writeln(' The max length ' ,max);                          {输出最佳长度}
 writeln(c);                                    {输出最佳的等差数列}
end.{main}
```

【上机练习】

1、高度与宽度(文件名：aa.pas)

输入杂乱排列的 N 个正整数，求出所有的最大平台（即出现次数最多的数）的平台宽度（最大平台数的个数）和平台高度（即最大平台中数的值）。例如：

输入：(aa.in)

10　　（正整数的个数 N，N ≤ 30）

1 2 3 1 2 3 1 2 1 2　　（N 个正整数，每两个间空一格）

输出：(aa.out)

1: its width is 4,its height is 1.

2: its width is 4,its height is 2.

2、求级数和（SERIES.PAS）

问题描述：

我们让计算机来做一道数学题：计算 S = 1/1!+1/2!+1/3!+...+1/ N!

(其中 N! = 1*2*3*...* N)

键盘输入整数 N (1≤N≤1000)。屏幕输出 S，四舍五入到 15 位小数。

输入输出样例：

INPUT：N = 3

OUTPUT：S = 1.666666666666667

3、密码破译(PASSWORD.PAS)

问题描述：

某组织欲破获一个外星人的密码，密码由一定长度的字串组成。此组织拥有一些破译此密码的长度不同的钥匙，若两个钥匙的长度之和恰好为此密码的长度，则此密码被成功破译。现在就请你编程找出能破译此密码的两个钥匙。

输入文件（PASSWORD.IN）：

输入文件第一行为钥匙的个数 N（1≤N≤5000）

输入文件第二行为密码的长度

以下 N 行为每个钥匙的长度

输出文件（PASSWORD.OUT）：

若无法找到破译此密码的钥匙，则输出文件仅 1 行 0 。

若找到两把破译的钥匙，则输出文件有两行，分别为两把钥匙的编号。

若有多种破译方案，则只输出一种即可。

输入输出样例：

PASSWORD.IN

10
80
27
9
4
73
23
68
12
64
92
24

PASSWORD.OUT

6
7

题目要求：

所有长度、计算均在 Longint 范围之内。程序运行限定时间：2 秒

4、彩票摇奖(LOTTERY.PAS)

问题描述：

为了丰富人民群众的生活、支持某些社会公益事业，北塔市设置了一项彩票。该彩票的规则是：

(1) 每张彩票上印有7 个各不相同的号码，且这些号码的取指范围为1~33。

(2) 每次在兑奖前都会公布一个由七个各不相同的号码构成的中奖号码。

(3) 共设置7 个奖项，特等奖和一等奖至六等奖。兑奖规则如下：

特等奖：要求彩票上7 个号码都出现在中奖号码中。

一等奖：要求彩票上有6 个号码出现在中奖号码中。

二等奖：要求彩票上有5 个号码出现在中奖号码中。

三等奖：要求彩票上有4 个号码出现在中奖号码中。

四等奖：要求彩票上有3 个号码出现在中奖号码中。

五等奖：要求彩票上有2 个号码出现在中奖号码中。

六等奖：要求彩票上有1 个号码出现在中奖号码中。

注：兑奖时并不考虑彩票上的号码和中奖号码中的各个号码出现的位置。例如，中奖号码为23 31 1 14 19 17 18，则彩票12 8 9 23 1 16 7 由于其中有两个号码(23 和1)出现在中奖号码中，所以该彩票中了五等奖。

现已知中奖号码和小明买的若干张彩票的号码，请你写一个程序帮助小明判断他买的彩票的中奖情况。

输入文件(LOTTERY.IN)：

输入文件的第一行只有一个自然数 N ≤ 1000，表示小明买的彩票张数；第二行存放了7 个介于1 和33 之间的自然数，表示中奖号码；在随后的 N 行中每行都有 7 个介于 1 和33 之间的自然数，分别表示小明所买的 N 张彩票。

输出文件(LOTTERY.OUT)：

依次输出小明所买的彩票的中奖情况(中奖的张数)，首先输出特等奖的中奖张数，然后依次输出一等奖至六等奖的中奖张数。

输入输出样例：

LOTTERY.IN

2

23 31 1 14 19 17 18

12 8 9 23 1 16 7

11 7 10 21 2 9 31

LOTTERY.OUT

0 0 0 0 0 1 1

5、周期串（PERIODIC.PAS）

问题描述：

如果一个字符串是以一个或者一个以上的长度为 k 的重复字符串所连接成的，那么这个字符串就被称为周期为 k 的字符串。例如，字符串”abcabcabcabc”周期为 3，因为它是由 4 个循环”abc”组成的。它同样是以 6 为周期（两个重复的”abcabc”）和以 12 为周期（一个循环”abcabcabcabc”）。

写一个程序，读入一个字符串，并测定它的最小周期。

输入文件（PERIODIC.IN）：

一个最长为 100 的没有空格的字符串。

输出文件（PERIODIC.OUT）：

一个整数表示输入的字符串的最小周期。

输入输出样例：

PERIODIC.IN

HoHoHo

PERIODIC.OUT

2

6、生日日期(BIRTHDAY.PAS)

问题描述：

小甜甜的生日是 YY 年 MM 月 DD 日，他想知道自己出生后第一万天纪念日的日期（出生日算第 0 天）。

输入文件（BIRTHDAY.IN）：

从文件的第一行分别读入 YY，MM，DD 其中 1949 <= YY <= 2002，日期绝对合法。

输出文件（BIRTHDAY.OUT）：

输出文件只有一行，即小甜甜生日第一万天以后的日期，格式为 “YY-MM-DD”。

输入输出样例：

BIRTHDAY.IN	BIRTHDAY.OUT
1975 7 15	2002-11-30

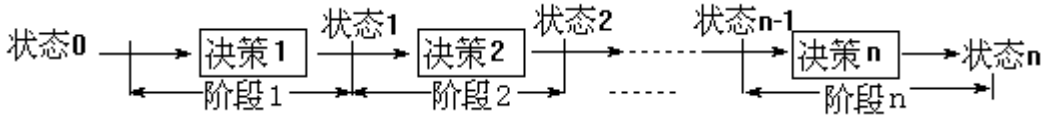
第十四章　动态规划

动态规划程序设计是对解最优化问题的一种途径、一种方法，而不是一种特殊算法。不象前面所述的那些搜索或数值计算那样，具有一个标准的数学表达式和明确清晰的解题方法。动态规划程序设计往往是针对一种最优化问题，由于各种问题的性质不同，确定最优解的条件也互不相同，因而动态规划的设计方法对不同的问题，有各具特色的解题方法，而不存在一种万能的动态规划算法，可以解决各类最优化问题。因此读者在学习时，除了要对基本概念和方法正确理解外，必须具体问题具体分析处理，以丰富的想象力去建立模型，用创造性的技巧去求解。我们也可以通过对若干有代表性的问题的动态规划算法进行分析、讨论，逐渐学会并掌握这一设计方法。

第一节 动态规划的基本模型

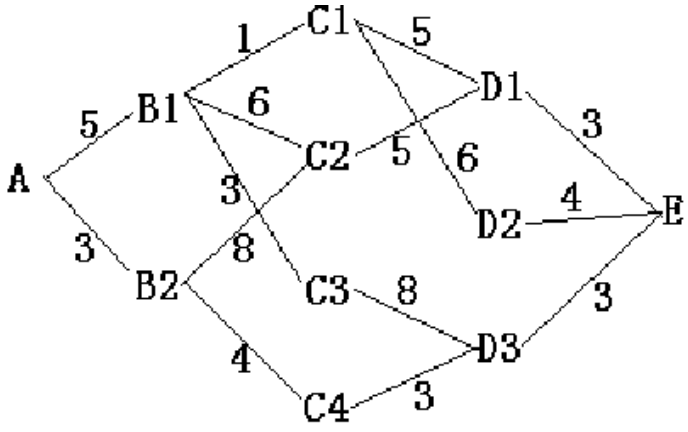
一、多阶段决策过程的最优化问题

在现实生活中，有一类活动的过程，由于它的特殊性，可将过程分成若干个互相联系的阶段，在它的每一阶段都需要作出决策，从而使整个过程达到最好的活动效果。当然，各个阶段决策的选取不是任何确定的，它依赖于当前面临的状态，又影响以后的发展，当各个阶段决策确定后，就组成一个决策序列，因而也就确定了整个过程的一条活动路线，这种把一个问题看作是一个前后关联具有链状结构的多阶段过程就称为多阶段决策过程，这种问题就称为多阶段决策问题。如下图所示：



多阶段决策过程，是指这样的一类特殊的活动过程，问题可以按时间顺序分解成若干相互联系的阶段，在每一个阶段都要做出决策，全部过程的决策是一个决策序列。要使整个活动的总体效果达到最优的问题，称为**多阶段决策问题**。

例 1：最短路径问题。下图给出了一个地图，地图中的每个顶点代表一个城市，两个城市间的一条连线代表道路，连线上的数值代表道路的长度。现在想从城市 A 到达城市 E，怎样走路程最短？最短路程的长度是多少？



【算法分析】把 A 到 E 的全过程分成四个阶段，用 K 表示阶段变量，第 1 阶段有一个初始状态 A，有两条可供选择的支路 A-B1、A-B2；第 2 阶段有两个初始状态 B1、B2，B1 有三条可供选择的支路，B2 有两条可供选择的支路……。用 DK（X_I，$X+1_J$）表示在第 K 阶段由初始状态 X_I 到下阶段的初始状态 $X+1_J$ 的路径距离，FK（X_I）表示从第 K 阶段的 X_I 到终点 E 的最短距离，利用倒推的方法，求解 A 到 E 的最短距离。具体计算过程如下：

S1 ： K = 4 有

F4（D1）= 3，

F4（D2）= 4，

F4（D3）= 3；

S2 ： K = 3 有

F3（C1）= MIN{ D3（C1，D1）+ F4（D1），D3（C1，D2）+ F4（D2）}

= MIN{ 5+3，6+4 } = 8，

F3（C2）= D3（C2，D1）+ F4（D1）= 5+3 = 8；

F3（C3）= D3（C3，D3）+ F4（D3）= 8+3 = 11；

F3（C4）= D3（C4，D3）+ F4（D3）= 3+3 = 6；

S3 ： K = 2 有

F2（B1）= MIN{ D2（B1，C1）+ F3（C1），D2（B1，C2）+ F3（C2），

D2（B1，C3）+ F3(C3)} = MIN{ 1+8,6+8,3+11} = 9,

F2（B2）= MIN{ D2（B2，C2）+ F3（C2），D2（B2，C4）+ F3（C4）}

= MIN{ 8+8，4+6 } = 10；

S4 ： K = 1 有

F1（A）= MIN{ D1（A，B1）+ F2（B1），D1（A，B2）+ F2（B2）}

= MIN{ 5+9，3+10} = 13。

因此由 A 点到 E 点的全过程最短路径为 A→B2→C4→D3→E；最短路程长度为 13。

从以上过程可以看出，每个阶段中，都求出本阶段的各个初始状态到终点 E 的最短距离，当逆序倒推到过程起点 A 时，便得到了全过程的最短路径和最短距离。

在上例的多阶段决策问题中，各个阶段采取的决策，一般来说是与阶段有关的，决策依赖于当前状态，又随即引起状态的转移，一个决策序列就是在变化的状态中产生出来的，故有“动态”的含义，我们称这种解决多阶段决策最优化的过程为动态规划程序设计方法。

二、动态规划的基本概念和基本模型构成

现在我们来介绍动态规划的基本概念。

1. 阶段和阶段变量：

用动态规划求解一个问题时，需要将问题的全过程恰当地分成若干个相互联系的阶段，以便按一定的次序去求解。描述阶段的变量称为阶段变量，通常用 K 表示，阶段的划分一般是根据时间和空间的自然特征来划分，同时阶段的划分要便于把问题转化成多阶段决策过程，如例题 2 中，可将其划分成 4 个阶段，即 K = 1，2，3，4。

2. 状态和状态变量：

某一阶段的出发位置称为状态，通常一个阶段包含若干状态。一般地，状态可由变量来描述，用来描述状态的变量称为状态变量。如例题 2 中，C3 是一个状态变量。

3. 决策、决策变量和决策允许集合：

在对问题的处理中作出的每种选择性的行动就是决策。即从该阶段的每一个状态出发，通过一次选择性的行动转移至下一阶段的相应状态。一个实际问题可能要有多次决策和多个决策点，在每一个阶段的每一个状态中都需要有一次决策，决策也可以用变量来描述，称这种变量为决策变量。在实际问题中，决策变量的取值往往限制在某一个范围之内，此范围称为允许决策集合。如例题 2 中，F3（C3）就是一个决策变量。

4．策略和最优策略：

所有阶段依次排列构成问题的全过程。全过程中各阶段决策变量所组成的有序总体称为策略。在实际问题中，从决策允许集合中找出最优效果的策略成为最优策略。

5. 状态转移方程

前一阶段的终点就是后一阶段的起点，对前一阶段的状态作出某种决策，产生后一阶段的状态，这种关系描述了由 k 阶段到 k+1 阶段状态的演变规律，称为状态转移方程。

三、最优化原理与无后效性

上面已经介绍了动态规划模型的基本组成，现在需要解决的问题是：什么样的“多阶段决策问题”才可以采用动态规划的方法求解。

一般来说，能够采用动态规划方法求解的问题，必须满足**最优化原理**和**无后效性原则**：

1、动态规划的最优化原理。作为整个过程的最优策略具有：无论过去的状态和决策如何，对前面的决策所形成的状态而言，余下的诸决策必须构成最优策略的性质。也可以通俗地理解为子问题的局部最优将导致整个问题的全局最优，即问题具有最优子结构的性质，也就是说一个问题的最优解只取决于其子问题的最优解，而非最优解对问题的求解没有影响。在例题 2 最短路径问题中，A 到 E 的最优路径上的任一点到终点 E 的路径，也必然是该点到终点 E 的一条最优路径，即整体优化可以分解为若干个局部优化。

2、动态规划的无后效性原则。所谓无后效性原则，指的是这样一种性质：某阶段的状态一旦确定，则此后过程的演变不再受此前各状态及决策的影响。也就是说，“未来与过去无关”，当前的状态是此前历史的一个完整的总结，此前的历史只能通过当前的状态去影响过程未来的演变。在例题 2 最短路径问题中，问题被划分成各个阶段之后，阶段 K 中的状态只能由阶段 K+1 中的状态通过状态转移方程得来，与其它状态没有关系，特别与未发生的状态没有关系，例如从 Ci 到 E 的最短路径，只与 Ci 的位置有关，它是由 Di 中的状态通过状态转移方程得来，与 E 状态，特别是 A 到 Ci 的路径选择无关，这就是无后效性。

由此可见，对于不能划分阶段的问题，不能运用动态规划来解；对于能划分阶段，但不符合最优化原理的，也不能用动态规划来解；既能划分阶段，又符合最优化原理的，但不具备无后效性原则，还是不能用动态规划来解；误用动态规划程序设计方法求解会导致错误的结果。

四、动态规划设计方法的一般模式

动态规划所处理的问题是一个多阶段决策问题，一般由初始状态开始，通过对中间阶段决策的选择，达到结束状态；或倒过来，从结束状态开始，通过对中间阶段决策的选择，达到初始状态。这些决策形成一个决策序列，同时确定了完成整个过程的一条活动路线，通常

是求最优活动路线。

动态规划的设计都有着一定的模式，一般要经历以下几个步骤：

1、划分阶段：按照问题的时间或空间特征，把问题划分为若干个阶段。在划分阶段时，注意划分后的阶段一定是有序的或者是可排序的，否则问题就无法求解。

2、确定状态和状态变量：将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来。当然，状态的选择要满足无后效性。

3、确定决策并写出状态转移方程：因为决策和状态转移有着天然的联系，状态转移就是根据上一阶段的状态和决策来导出本阶段的状态。所以如果确定了决策，状态转移方程也就可以写出。但事实上常常是反过来做，根据相邻两段的各个状态之间的关系来确定决策。

4、寻找边界条件：给出的状态转移方程是一个递推式，需要一个递推的终止条件或边界条件。

5、程序的设计实现：一旦设计完成，实现就会非常简单。下面我们给出从初始状态开始，通过对中间阶段决策的选择，达到结束状态，按照阶段、状态和决策的层次关系，写出的程序流程的一般形式：

```
所有状态费用的初始化；
for i := 阶段最大值-1 downto 1 do     {倒推每一个阶段}
  for j := 状态最小值 to 状态最大值 do  {枚举阶段 i 的每一个状态}
    for k := 决策最小值 to 决策最大值 do {枚举阶段 i 中状态 j 可选择的每一种决策}
      begin
        f[i,j] ← min{d[i,j,k] + f[i+1,k]}
      end;
输出 f[1,1];
```

例 1 对应的 Pascal 程序如下：

```
var  d : array[1..4,1..4,1..4] of byte;
     f : array[1..5,1..4] of byte;
     i,j,k,min : byte;
begin
  fillchar(d,sizeof(d),0);
  d[1,1,1] := 5; d[1,1,2] := 3;
  d[2,1,1] := 1; d[2,1,2] := 6; d[2,1,3] := 3;
  d[2,2,2] := 8; d[2,2,4] := 4;
  d[3,1,1] := 5; d[3,1,2] := 6;
  d[3,2,1] := 5;
  d[3,3,3] := 8;
  d[3,4,3] := 3;
  d[4,1,1] := 3;
  d[4,2,1] := 4;
  d[4,3,1] := 3;
  fillchar(f,sizeof(f),255);
  f[5,1] := 0;
  for i := 4 downto 1 do
```

```
    for j := 1 to 4 do
      for k := 1 to 4 do
        if d[i,j,k] <> 0 then
          if f[i,j] > d[i,j,k]+f[i+1,k] then f[i,j] := d[i,j,k]+f[i+1,k];
  writeln(f[1,1]);
  readln;
end.
```

第二节　动态规划与递推

——动态规划是最优化算法

由于动态规划的“名气”如此之大，以至于很多人甚至一些资料书上都往往把一种与动态规划十分相似的算法，当作是动态规划。这种算法就是递推。实际上，这两种算法还是很容易区分的。

按解题的目标来分，信息学试题主要分四类：判定性问题、构造性问题、计数问题和最优化问题。我们在竞赛中碰到的大多是最优化问题，而动态规划正是解决最优化问题的有力武器，因此动态规划在竞赛中的地位日益提高。而递推法在处理判定性问题和计数问题方面也是一把利器。下面分别就两个例子，谈一下递推法和动态规划在这两个方面的联系。

一、逆推法：

例 2：数塔问题（IOI94）：有形如图 1.3-8 所示的数塔，从顶部出发，在每一结点可以选择向左走或是向右走，一起走到底层，要求找出一条路径，使路径上的值最大。

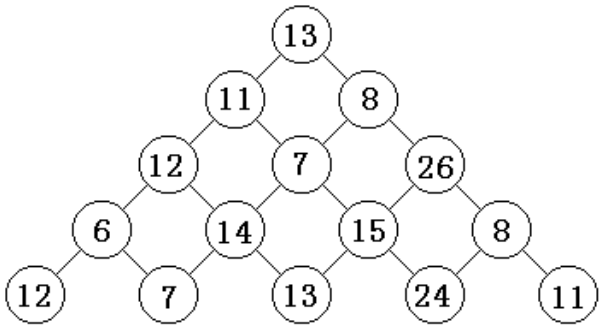


图 1.3-8

这道题如果用枚举法，在数塔层数稍大的情况下（如 40），则需要列举出的路径条数将是一个非常庞大的数目。如果用贪心法又往往得不到最优解。在用动态规划考虑数塔问题时可以自顶向下的分析，自底向上的计算。从顶点出发时到底向左走还是向右走应取决于是从左走能取到最大值还是从右走能取到最大值，只要左右两道路径上的最大值求出来了才能作出决策。同样的道理下一层的走向又要取决于再下一层上的最大值是否已经求出才能决策。这样一层一层推下去，直到倒数第二层时就非常明了。所以实际求解时，可从底层开始，层层递进，最后得到最大值。实际求解时应掌握其编程的一般规律，通常需要哪几个关键数组来存储变化过程这一点非常重要。

一般说来，很多最优化问题都有着对应的计数问题；反过来，很多计数问题也有着对应的最优化问题。因此，我们在遇到这两类问题时，不妨多联系、多发展，举一反三，从比较中更深入地理解动态规划的思想。

其实递推和动态规划这两种方法的思想本来就很相似，也不必说是谁借用了谁的思想。关键在于我们要掌握这种思想，这样我们无论在用动态规划法解最优化问题，或是在用递推法解判定型、计数问题时，都能得心应手、游刃有余了。

【算法分析】

①贪心法往往得不到最优解：本题若采用贪心法则：13-11-12-14-13，其和为 63

但存在另一条路：13-8-26-15-24，其和为 86。

贪心法问题所在：眼光短浅。

②动态规划求解：动态规划求解问题的过程归纳为：自顶向下的分析，自底向上计算。

其基本方法是：

划分阶段：按三角形的行，划分阶段，若有 n 行，则有 n-1 个阶段，找到问题求解的最优路径。

A．从根结点 13 出发，选取它的两个方向中的一条支路，当到倒数第二层时，每个结点其后继仅有两个结点，可以直接比较，选择最大值为前进方向，从而求得从根结点开始到底端的最大路径。

B．自底向上计算：（给出递推式和终止条件）

①从底层开始，本身数即为最大数；

②倒数第二层的计算，取决于底层的数据：12+6=18，13+14=27，**24+15**=39，24+8=32；

③倒数第三层的计算，取决于底二层计算的数据：27+12=39，39+7=46，39+**26**=65

④倒数第四层的计算，取决于底三层计算的数据：46+11=57，65+**8**=73

⑤最后的路径：13——8——26——15——24

C．数据结构及算法设计

①图形转化：直角三角形，更于搜索：向下、向右

②用三维数组表示数塔：a[x,y,1]表示行、列及结点本身数据,a[x,y,2]能够取得最大值,a[x,y,3]表示前进的方向——0 向下，1 向右；

③算法：

数组初始化，输入每个结点值及初始的最大路径、前进方向为 0；

从倒数第二层开始向上一层求最大路径，共循环 N-1 次；

从顶向下，输出路径：关键是 J 的值，由于行逐渐递增，表示向下，究竟向下还是向右取决于列的值。若 J 值比原先多 1 则向右，否则向下。

数塔问题的样例程序如下：

```
var a:array[1..50,1..50,1..3] of longint;x,y,n:integer;
begin
    write( 'please input the number of rows:');
    readln(n);
    for x:=1 to n do
        for y:=1 to x do
          begin
            read(a[x,y,1]);
```

```
            a[x,y,2]:=a[x,y,1];
            a[x,y,3]:=0
          end;
    for x:=n-1 downto 1 do
      for y:=1 to x do
          if a[x+1,y,2]>a[x+1,y+1,2] then
              begin  a[x,y,2]:=a[x,y,2]+a[x+1,y,2]; a[x,y,3]:=0 end
          else  begin a[x,y,2]:=a[x,y,2]+a[x+1,y+1,2];a[x,y,3]:=1 end;
    writeln('max=',a[1,1,2]);
    y:=1;
    for x:=1 to n-1 do
      begin
        write(a[x,y,1],'-> ');
        y:=y+a[x,y,3]
      end;
    writeln(a[n,y,1])
end.
```

输入：

```
5    {数塔层数}
13
11   8
12   7    26
6    14   15    8
12   7    13   24    11
```

输出结果　max=86

13—8—26—15—24

例 3:求最长不下降序列

㈠问题描述：设有由 n 个不相同的整数组成的数列，记为:b(1)、b(2)、……、b(n)且 b(i)<>b(j)　(i<>j)，若存在 i1<i2<i3< … < ie 且有 b(i1)<b(i2)< … <b(ie)则称为长度为 e 的不下降序列。程序要求，当原数列给出之后，求出最长的不下降序列。

例如 13，7，9，16，38，24，37，18，44，19，21，22，63，15。例中 13，16，18，19，21，22，63 就是一个长度为 7 的不下降序列，同时也有 7 ，9，16，18，19，21，22，63 长度为 8 的不下降序列。

㈡算法分析：根据动态规划的原理，由后往前进行搜索。

1・对 b(n)来说，由于它是最后一个数，所以当从 b(n)开始查找时，只存在长度为 1 的不下降序列；

2・若从 b(n-1)开始查找，则存在下面的两种可能性：

①若 b(n-1)<b(n)则存在长度为 2 的不下降序列 b(n-1)，b(n)。

②若 b(n-1)>b(n)则存在长度为 1 的不下降序列 b(n-1)或 b(n)。

3・一般若从 b(i)开始，此时最长不下降序列应该按下列方法求出:

在 b(i+1),b(i+2),…,b(n)中，找出一个比 b(i)大的且最长的不下降序列，作为它的后继。

㈢数据结构：为算法上的需要，定义一个数组整数类型二维数组 b(N,3)

1·b(I,1)表示第 I 个数的数值本身；

2·b(I,2)表示从 I 位置到达 N 的最长不下降序列长度

3·b(I,3)表示从 I 位置开始最长不下降序列的下一个位置，若 b[I,3]=0 则表示后面没有连接项。

㈣求解过程：

①从倒数第二项开始计算，后面仅有 1 项，比较一次，因 63>15，不符合要求，长度仍为 1。

②从倒数第三项开始其后有 2 项，需做两次比较，得到目前最长的不下降序列为 2，如下表：

	11	12	13	14	……		11	12	13	14
		22	63	15	……		21	22	63	15
		2	1	1	……		3	2	1	1
		13	0	0	……		12	13	0	0

㈤一般处理过程是：

①在 i+1,i+2,…,n 项中，找出比 b[I,1]大的最长长度 L 以及位置 K;

②若 L>0，则 b[I,2]:=L+1;b[I,3]:=k;

最后本题经过计算，其数据存储表如下：

1	2	3	4	5	6	7	8	9	10	11	12	13	14
13	7	9	16	38	24	37	18	44	19	21	22	63	15
7	8	7	6	3	4	3	5	2	4	3	2	1	1
4	3	4	8	9	7	9	10	13	11	12	13	0	0

初始化：

```
for i:=1 to n do
    begin
      read(b[i,1]);
      b[i,2]:=1;b[i,3]:=0;
    end;
```

下面给出求最长不下降序列的算法：

```
for i:=n-1 downto 1 do
      begin
         L:=0;k:=0;
         for j:=i+1 to n do
         if(b[j,1]>b[i,1])and(b[j,2]>L) then begin
                                              L:=b[j,2];k:=j;
                                            end;
         if L>0 then begin
                        b[i,2]:=L+1;b[i,3]:=k;
                     end;
      end;
```

下面找出最长不下降序列：

```
L:=1;
for j:=2 to n do
  if b[j,2]>b[L,2] then L:=j;
```

最长不下降序列长度为 B(L, 2)序列

```
while L<>0  do
  begin
    write(b[L,1]:4);
    L:=b[L,3];
  end;
```

【参考程序】

```
var
    n,i,L,k,j:integer;
    b:array[1..100,1..3]of integer;
begin
  writeln('input n:');
  readln(n);
  for i:=1 to n do
    begin
      read(b[i,1]);
      b[i,2]:=1;b[i,3]:=0;
    end;
  for i:=n-1 downto 1 do
      begin
         L:=0;k:=0;
         for j:=i+1 to n do
         if(b[j,1]>b[i,1])and(b[j,2]>L) then begin
                                               L:=b[j,2];
                                               k:=j;
                                             end;
         if L>0 then begin
                       b[i,2]:=L+1;b[i,3]:=k;
                     end;
      end;
      L:=1;
      for j:=2 to n do
         if b[j,2]>b[L,2] then L:=j;
      writeln('max=',b[L,2]);
        while L<>0  do
```

```
          begin
            write(b[L,1]:4);
            L:=b[L,3];
          end;
        writeln;
        readln;
end.
```

程序运行结果：

输入：13 7 9 16 38 24 37 18 44 19 21 22 63 15

输出：max=8

7 9 16 18 19 21 22 63

例 4：拦截导弹。某国为了防御敌国的导弹袭击，发展出一种导弹拦截系统。但是这种拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都不能高于前一发的高度。某天，雷达捕捉到敌国的导弹来袭，由于该系统还在试用阶段。所以只有一套系统，因此有可能不能拦截所有的导弹。

输入导弹依次飞来的高度(雷达给出的高度不大于 30000 的正整数)。计算这套系统最多能拦截多少导弹。

输入：N 颗依次飞来的导弹高度，(导弹个数<=1000)。

输出：一套系统最多拦截的导弹数，并依次打印输出被拦截导弹的高度。

在本题中不仅要求输出最优解，而且还要求输出最优解的形成过程。为此，我们设置了一张记忆表 C[i]，在按从后往前方式求解的过程中，将每一个子问题的最佳决策保存起来，避免在输出方案时重复计算。

阶段 i：由右而左计算导弹 n‥导弹 1 中可拦截的最多导弹数（1≤i≤n)；

状态 B[i]：由于每个阶段中仅一个状态，因此可通过一重循环

for i := n-1 downto 1 do 枚举每个阶段的状态 B[i]；

决策 k：在拦截导弹 i 之后应拦截哪一枚导弹可使得 B[i]最大（i+1≤k≤n)，

1	2	3	4	5	6	7	8	9	10	11	12	13	14	I
13	7	9	16	38	24	37	18	44	19	21	22	63	15	A[I] {高度}
2	1	1	2	4	3	3	2	3	2	2	2	2	1	B[I] {可拦截数}
2	0	0	14	6	8	8	14	10	14	14	14	14	0	C[I] {再拦截}

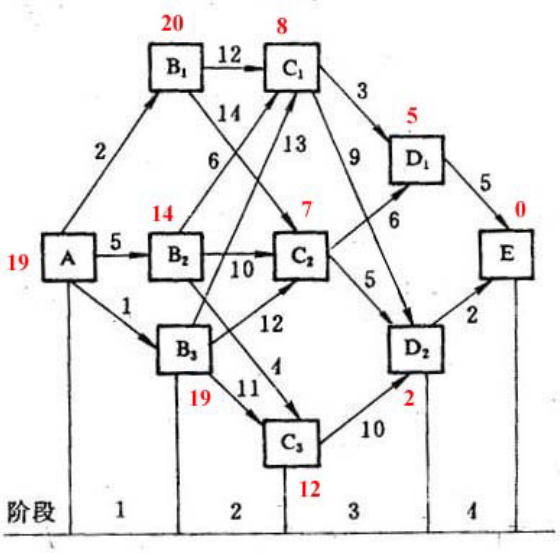
【参考程序】

```
var
   a,b,c       : array[1..1000] of word;
   n,i,j,k,max : word;
begin
  n := 0;                          {初始化，读入数据}
  while not eoln do begin          {eoln : end of line}
    inc(n); read(a[n]); b[n] := 1; c[n] := 0;
  end;
  readln;
```

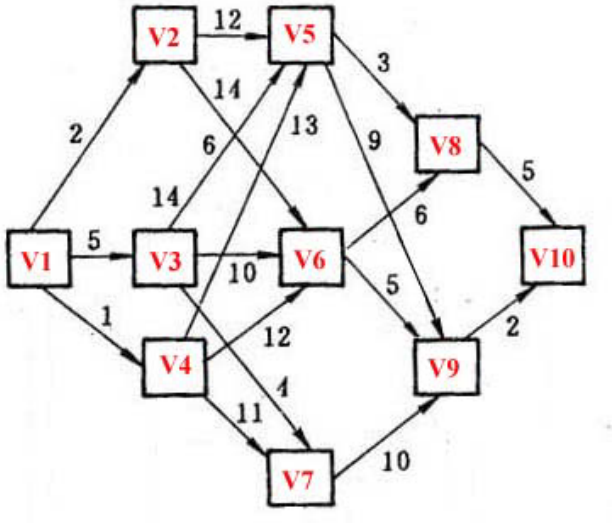
```
  for i := n-1 downto 1 do begin      {枚举每一个阶段的状态，设导弹 i 被拦截}
    max := 0; j := 0;
    for k := i+1 to n do              {枚举决策，计算最佳方案中拦截的下一枚导弹}
      if (a[k] <= a[i]) and (b[k] > max) then begin
        max := b[k]; j := k;
      end;
    b[i] := max+1; c[i] := j;         {若导弹 i 之后拦截导弹 j 为最佳方案，则记下}
  end;
  max := 0;
  for i := 1 to n do                  {枚举求出一套系统能拦截的最多导数}
    if b[i] > max then begin max := b[i]; j := i; end;
  writeln('OUTPUT');                  {打印输出结果}
  writeln('Max = ',b[j]);
  while j > 0 do begin
    write(a[j]:5); j := c[j];
  end;
  readln;
end.
```

例 5:下图表示城市之间的交通路网，线段上的数字表示费用，单向通行由 A->E。试用动态规划的最优化原理求出 A->E 的最省费用。

交通图 1



交通图 2



如图：求 v1 到 v10 的最短路径长度及最短路径。

【样例输入】

short.in

```
10
0  2  5  1  0  0  0  0  0  0
0  0  0  0 12 14  0  0  0  0
0  0  0  0  6 10  4  0  0  0
0  0  0  0 13 12 11  0  0  0
0  0  0  0  0  0  0  3  9  0
0  0  0  0  0  0  0  6  5  0
0  0  0  0  0  0  0  0 10  0
0  0  0  0  0  0  0  0  0  5
0  0  0  0  0  0  0  0  0  2
0  0  0  0  0  0  0  0  0  0
```

【样例输出】

```
short.out
minlong=19
1  3  5  8  10
```

采用逆推法

设 f（x）表示点 x 到 v10 的最短路径长度

则 f(10)=0

f(x)=min{ f(i)+a[x,i] 当 a[x,i]>0 ,x<i<=n}

【参考程序】（逆推法）

```
program short;
var
 a:array[1..100,1..100] of integer;
 b,c:array[1..100] of integer;
 i,j,n,x:integer;
begin
 assign(input,'short.in');
 assign(output,'short.out');
 reset(input);rewrite(output);
 readln(n);
 for i:=1 to n do
   for j:=1 to n do
     read(a[i,j]);
 for i:=1 to n do
     b[i]:=maxint;
 b[n]:=0;
 for i:= n-1 downto 1 do
  for j:=n downto i+1 do
   if (a[i,j]>0) and (b[j]<>maxint) and (b[j]+a[i,j]<b[i])
```

```
    then begin b[i]:=b[j]+a[i,j];c[i]:=j end;
 writeln('minlong=',b[1]);
 x:=1;
  while x<>0 do
   begin
     write(x:5);
     x:=c[x];
   end;
  close(input);close(output);
end.
```

二、顺推法：

上面的例题都是逆推的，决策的选择也较多。下面我们来讲是顺推的，而且决策的选择较少。

例 6：数塔问题。设有一个三角形的数塔，如下图所示。顶点结点称为根结点，每个结点有一个整数数值，其值不超过 100。从顶点出发，可以向左走，也可以向右走。

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

从根 13 出发，向左走到达 11，再向右走到达 7，再向左走到达 14，再向左到达 7。由于 7 是最底层，无路可走。此时，我们找到一条从根结点开始到达底层的路径：13-11-7-14-7。路径上结点中数字的和，称为路径的值，如上面路径的值为 13+11+7+14+7 = 52。

当三角形数塔给出之后，找出一条路径，使路径上的值为最大，打印输出最大路径的值。数塔的层数 N 最多可为 100。

【算法分析】

此题贪心法往往得不到最优解 ,例如 13-11-12-14-13 其路径的值为 63 ,但这不是最优解。

穷举搜索往往是不可能的，当层数 N = 100 时，路径条数 P = 2^{99} 这是一个非常大的数，即使用世界上最快的电子计算机，也不能在短时间内计算出来。

对这道题唯一正确的方法是动态规划。如果得到一条由顶到底的某处的一条最佳路径，那么对于该路径上的每一个中间点来说，由顶至该中间点的路径所经过的数字和也为最大。因此本题是一个典型的多阶段决策最优化问题。在本题中仅要求输出最优解，为此我们设置了数组 A[i,j]保存三角形数塔，B[i,j]保存状态值，按从上往下方式进行求解。

阶段 i：以层数来划分阶段，由从上往下方式计算层数 1‥层数 N（1≤i≤n）；因此可通过第一重循环

```
for i := 1 to n do begin 枚举每一阶段;
```

状态 B[i,j]：由于每个阶段中有多个状态，因此可通过第二重循环

```
for j := 1 to i do begin 求出每个阶段的每个状态的最优解 B[i,j];
```

决策：每个状态最多由上一层的两个结点连接过来，因此不需要做循环。

```
Var   a,b   : array[1..100,0..100] of word;
      i,j,n : byte;    max   : word;
begin
  repeat
    write('N = '); readln(n);
  until n in [1..100];
  fillchar(a,sizeof(a),0);
  b := a;
  for i := 1 to n do begin
    for j := 1 to i do read(a[i,j]);
    readln;
  end;
  b[1,1] := a[1,1];
  for i := 2 to n do
    for j := 1 to i do
      if b[i-1,j-1] > b[i-1,j]
        then b[i,j] := b[i-1,j-1]+a[i,j]
        else b[i,j] := b[i-1,j]+a[i,j];
  max := 0;
  for i := 1 to n do
    if b[n,i] > max then max := b[n,i];
  writeln('Max = ',max);
  readln;
end.
```

例 7： 拦截导弹

某国为了防御敌国的导弹袭击，发展出一种导弹拦截系统。但是这种导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都不能高于前一发的高度。某天，雷达捕捉到敌国的导弹来袭。由于该系统还在试用阶段，所以只有一套系统，因此有可能不能拦截所有的导弹。

输入导弹依次飞来的高度（雷达给出的高度数据是不大于 30000 的正整数），计算这套系统最多能拦截多少导弹，如果要拦截所有导弹最少要配备多少套这种导弹拦截系统。

样例：

INPUT	OUTPUT
389 207 155 300 299 170 158 65	6（最多能拦截的导弹数）
	2（要拦截所有导弹最少要配备的系统数）

【算法分析】

第一部分是要求输入数据串中的一个最长不上升序列的长度，可使用递推的方法，具体做法是从序列的第一个元素开始，依次求出以第 i 个元素为最后一个元素时的最长不上升序列的长度 s(i)，递推公式为 s(1)=1，s(i)=max(s(j)+1)，其中 i>1，j=1,2,…i-1，且 j 同

时要满足条件：序列中第 j 个元素大于等于第 i 个元素。

第二部分比较有意思。由于它紧接着第一问，所以很容易受前面的影响，采取多次求最长不上升序列的办法，然后得出总次数，其实这是不对的。要举反例并不难，比如长为 7 的高度序列“7 5 4 1 6 3 2”， 最长不上升序列为“7 5 4 3 2”，用多次求最长不上升序列的结果为 3 套系统；但其实只要 2 套，分别击落“7 5 4 1”与“6 3 2”。那么，正确的做法又是什么呢？

我们的目标是用最少的系统击落所有导弹，至于系统之间怎么分配导弹数目则无关紧要；上面错误的想法正是承袭了“一套系统尽量多拦截导弹”的思维定势，忽视了最优解中各个系统拦截数较为平均的情况，本质上是一种贪心算法。如果从每套系统拦截的导弹方面来想行不通的话，我们就应该换一个思路，从拦截某个导弹所选的系统入手。

题目告诉我们，已有系统目前的瞄准高度必须不低于来犯导弹高度，所以，当已有的系统均无法拦截该导弹时，就不得不启用新系统。如果已有系统中有一个能拦截该导弹，我们是应该继续使用它，还是另起炉灶呢？事实是：无论用哪套系统，只要拦截了这枚导弹，那么系统的瞄准高度就等于导弹高度，这一点对旧的或新的系统都适用。而新系统能拦截的导弹高度最高，即新系统的性能优于任意一套已使用的系统。既然如此，我们当然应该选择已有的系统。如果已有系统中有多个可以拦截该导弹，究竟选哪一个呢？当前瞄准高度较高的系统的“潜力”较大，而瞄准高度较低的系统则不同，它能打下的导弹别的系统也能打下，它够不到的导弹却未必是别的系统所够不到的。所以，当有多个系统供选择时，要选瞄准高度最低的使用，当然瞄准高度同时也要大于等于来犯导弹高度。

解题时，用一个数组记下已有系统的当前瞄准高度，数据个数就是系统数目。

解法一：

```
program sheep506;
var
    i,n,j,best,num,x:integer;
    a,s,b:array[0..1000] of integer;
    f:text;
 begin
  assign(f,'cz2.txt');
  reset(f);
  readln(f,n);
  for i:=1 to n do
   begin
     read(f,a[i]);
      s[i]:=1;
   end;
  close(f);
  for i:=2 to n do
   begin
    for j:=1 to i-1 do
     if (a[j]>=a[i]) and (s[j]+1>s[i]) then
       s[i]:=s[j]+1;
     if s[i]>best then best:=s[i];
```

```
 end;
writeln('Max:',best);
num:=0;
b[0]:=9999;
for i:=1 to n do
 begin
  x:=0;
  for j:=1 to num do
   if (b[j]>a[i]) and (b[j]<b[x]) then x:=j;
  if x=0 then begin inc(num);b[num]:=a[i] end
  else b[x]:=a[i];
 end;
writeln(num);
readln;
end.
```

第一问经过计算，其数据存储表如下

	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
	389	207	155	300	299	170	158	65
	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]
I=1	1	1	1	1	1	1	1	1
I=2	1	2	1	1	1	1	1	1
I=3	1	2	3	1	1	1	1	1
I=4	1	2	3	2	1	1	1	1
I=5	1	2	3	2	3	1	1	1
I=6	1	2	3	2	3	4	1	1
I=7	1	2	3	2	3	4	5	1
I=8	1	2	3	2	3	4	5	6

第二问经过计算，其数据存储表如下：

	num	b[0]	b[1]	b[2]	b[3]	b[4]
I=1	1	9999	389			
I=2			207			
I=3			155			
I=4	2			300		
I=5				299		
I=6				170		
I=7				158		
I=8			65			

解法二：

第一问即经典的最长不下降子序列问题，可以用一般的 DP 算法，也可以用高效算法，但这个题的数据规模好像不需要。

高效算法是这样的：用 a[x]表示原序列中第 x 个元素，b[x]表示长度为 x 的不下降子序列的最后一个元素的最小值，b 数组初值为无穷大。容易看出，这个数组是递减的（当然

可能有相邻两个元素相等)。当处理第 a[x]时，用二分法查找它可以连接到长度最大为多少的不下降子序列后（即与部分 b[x]比较)。假设可以连到长度最大为 y 的不下降子序列后，则 b[y+1]:=min(b[y+1],a[x])。最后，b 数组被赋值的元素最大下标就是第一问的答案。由于利用了二分查找，这种算法的复杂度为 O(nlogn)，优于一般的 O(n^2)。

第二问用贪心法即可。每颗导弹来袭时，使用能拦截这颗导弹的防御系统中上一次拦截导弹高度最低的那一套来拦截。若不存在符合这一条件的系统，则使用一套新系统。

```
program tju1004;
const
  max=20;
var
  a,b,h:array[1..max]of word;
  i,j,m,n,x:byte;
begin
  repeat
    inc(i);
    read(a[i]);
    for j:=1 to i-1 do
      if a[j]>=a[i] then
        if b[j]>b[i] then b[i]:=b[j];
    inc(b[i]);
    if b[i]>m then m:=b[i];
    x:=0;
    for j:=1 to n do
      if h[j]>=a[i] then
        if x=0 then x:=j
            else if h[j]<h[x] then x:=j;
    if x=0 then begin inc(n);x:=n;end;
    h[x]:=a[i];
  until seekeof;
  writeln(m,' ',n);
end.
```

经过计算，其数据存储表如下

I	I=1	I=2	I=3	I=4	I=5	I=6	I=7	I=8
A[I]	389	207	155	300	299	170	158	65
B[I]	1	2	3	2	3	4	5	6
N 值	1			2				
H[1]	389	207	155					65
H[2]				300	299	170	158	

第三节　动态规划经典例题

【例 1】骑士游历问题

设有一个 n*m 的棋盘(2≤n≤50，2≤m≤50)，如图 1 1.2.1。在棋盘上任一点有一个中国象棋马，

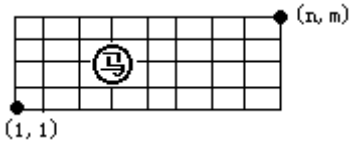


图 1 1.2.1

马走的规则为：

1.马走日字　2.马只能向右走。即图 1 1.2.2 所示：

图 1 1.2.2

当 N，M 给出之后，同时给出马起始的位置和终点的位置，试找出从起点到终点的所有路径的数目。例如：(N=10，M=10)，(1，5)(起点)，(3，5)(终点)。应输出 2(即由(1，5)到(3，5)共有 2 条路径，如图 1 1.2.3)：

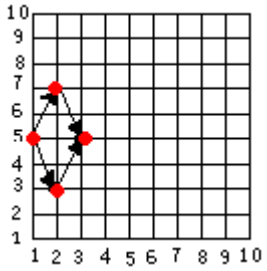


图 1 1.2.3

输入：

n，m，x1，y1，x2，y2(分别表示 n，m，起点坐标，终点坐标)

输出：

路径数目(若不存在从起点到终点的路径，输出 0)

【算法分析】

使用回溯法同样可以计算路径数目。只要将起点（1，1）和终点（n，m）调整为（x1，y1)、(x2，y2)，并在回溯程序(search(k，x，y))中，将马跳到目的地时由退出程序（halt）改为回溯（exit）即可。但问题是搜索效率太低，根本不可能在较短的时间内出解。本题并不要求每一条路径的具体走法。在这种情况下，是否非得通过枚举所有路径方案后才能得出路径数目，有没有一条简便和快效的“捷径”呢。

从(x_1，y_1)出发，按照由左而右的顺序定义阶段的方向。位于（x，y）左方且可达（x，y）的跳马位置集合都是（x，y）的子问题，起点至（x，y）的路径数实际上等于起点至这些位置集的路径数之和（图 11.2.4）。

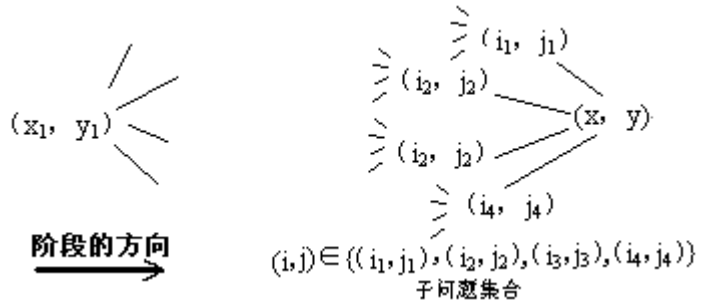


图 11.2.4

如此一来，状态转移关系便凸显出来。设状态转移方程 map，其中 map[i，j]为起点（x1，y1）至（i，j）的路径数目。由于棋盘规模的上限为 50*50，可能导致路径数目大得惊人，因此不妨设 map 数组的元素类型为 extended。初始时，除 map[x1，y1]=1 外其余为 0 。显然

$$map[x,y]=\sum_{(i,j)\in 可达(x,y)的坐标集}\{map[i,j]+map[x,y]\mid (x,y)在界内\}。$$

我们采用动态程序设计的方法计算起点(x1，y1)至终点（x2，y2）的路径数目 map[x2，y2]：

阶段 j：中国象棋马当前的列位置(y1≤j≤y2)；

状态 i：中国象棋马在 j 列的行位置（1≤i≤n)；

决策 k：中国象棋马在（i，j）的起跳方向(1≤k≤4)；

计算过程如下：

```
fillchar(map，sizeof(map)，0)；
map[x1，y1] ←1；                                    {从（x1，y1）出发}
for j←y1 to y2 do                                   {递推中国象棋马的列位置}
  for i←1 to n do                                   {递推中国象棋马在 j 列的行位置}
    for k←1 to 4 do                                 {递推中国象棋马在（i，j）的 4 个跳动方向}
      begin
        中国象棋马由（i，j）出发，沿着 k 方向跳至（x，y)；
        if (x∈｛1..n｝)∧(y∈{1..y2})                  {计算状态转移方程}
          then map[x，y] ←map[i，j]+map[x，y]
      end；{for}
writeln(map[x2，y2]：0：0)；                        {输出从（x1，y1）到（x2，y2）的路径数目}
```

【例 2】砝码称重

设有 1g，2g，3g，5g，10g，20g 的砝码各若干枚（其总重≤1000g)，要求：

输入：

a1　a2　a3　a4　a5　a6(表示 1g 砝码有 a1 个，2g 砝码有 a2 个，....20g 砝码有 a6 个)

输出：

Total=N (N 表示用这些砝码能称出的不同重量的个数，但不包括一个砝码也不用的情况)

输入样例：1 1 0 0 0 0

输出样例：Total=3，表示可以称出 1g，2g，3g 三种不同的重量

【算法分析】

设

```
const num: array[1..6] of shortint=(1, 2, 3, 5, 10, 20);        {砝码的重量序列}
var
  a: array[1..6] of integer;                                    { 6 种砝码的个数}
  visited: array[0..1000] of boolean;                           {重量的访问标志序列}
  n: array[0..1000] of integer;{n[0]—不同重量数; n[j]—第 j 种重量( 1 ≤j≤n[0])}
  total, i, j, k: integer;                                      {total—目前称出的重量}
```

我们按照第 1 种砝码，第 2 种砝码……第 6 种砝码的顺序分析。在分析第 i 种砝码的放置方案时，依次在现有的不同重量的基础上，放 1 块、2 块……a[i]块，产生新的不同重量。

n[n[0]+1]=total | total=n[j]+k*num[i], visited[total]=false, 1 ≤i≤6, 1 ≤j≤n[0], 1 ≤k≤a[i]

阶段 i： 分析第 i 种砝码(1 ≤i≤6);

状态 j：枚举现有的不同重量 (1 ≤j≤n[0]);

决策 k：在现有重量的基础上放 k 块第 i 种砝码，产生重量 n[j]+k*num[i](1 ≤k≤a[i]);

计算过程如下：

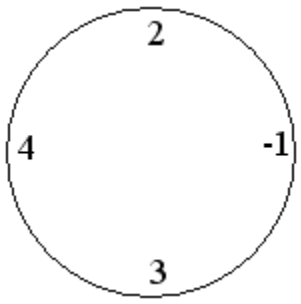
```
  fillchar(visited, sizeof(visited), false);
  write(' a1-a6:' ); for i←1 to 6 do read(a[i]);          {输入 6 种砝码的个数}
  n[0] ←1; n[1] ←0;                                       {产生第 1 种重量 0 }
  for i←1 to 6 do                                         {阶段：分析第 i 种砝码}
    for j←1 to n[0] do                                    {状态：枚举现有的不同重量}
      for k←1 to a[i] do                {决策：在现有重量的基础上放 k 块第 i 种砝码}
      begin
        total←n[j]+k*num[i];                                          {产生重量}
        if not visited[total] then              {若该重量未产生过，则设访问标志}
        begin
          visited[total] ←true;
          inc(n[0]);                                                  重量进入 n 序列}
          n[n[0]] ←total;
        end; {then}
      end; {for}
  writeln(n[0]-1);                                        {输出不同重量的个数}
```

【例 3】数字游戏

【问题描述】

丁丁最近沉迷于一个数字游戏之中。这个游戏看似简单，但丁丁在研究了许多天之后却发觉原来在简单的规则下想要赢得这个游戏并不那么容易。游戏是这样的，在你面前有一圈

整数（一共 n 个），你要按顺序将其分为 m 个部分，各部分内的数字相加，相加所得的 m 个结果对 10 取模后再相乘，最终得到一个数 k。游戏的要求是使你所得的 k 最大或者最小。例如，对于下面这圈数字（n=4，m=2）：



当要求最小值时，((2-1) mod 10)×((4+3) mod 10)=1×7=7，要求最大值时，为((2+4+3) mod 10)×(-1 mod 10)=9×9=81。特别值得注意的是，无论是负数还是正数，对 10 取模的结果均为非负值。

丁丁请你编写程序帮他赢得这个游戏。

【输入格式】

输入文件第一行有两个整数，n（1≤n≤50）和 m（1≤m≤9）。以下 n 行每行有个整数，其绝对值不大于 10^4，按顺序给出圈中的数字，首尾相接。

【输出格式】

输出文件有两行，各包含一个非负整数。第一行是你程序得到的最小值，第二行是最大值。

【输入样例】

```
4 2
4
3
-1
2
```

【输出样例】

```
7
81
```

【算法分析】

设圆周上的 n 个数字为 a_1、a_2、…a_n。按照模运算的规则（$a_1+a_2+\cdots+a_k$）mod 10=(a_1 mod 10+a_2 mod 10+ a_k mod 10)mod 10；g[i，j]为 a_i、a_{i+1}、…a_j的数和对 10 的模，即

$$g[i,j]=(a_i+a_{i+1}+\cdots+a_j) \bmod 10$$

显然

$$g[i,i]=(a_i \bmod 10+10) \bmod 10$$

$$g[i,j]=(g[i,j-1]+g[j,j]) \bmod 10 \quad (1\leqslant i\leqslant n,\ i+1\leqslant j\leqslant n)$$

当 m=1 时，程序得到的最小值和最大值为 g[1，n]；

fmax1[p，I，j]为 a_i、a_{i+1}、…a_j 分为 p 个部分，各部分内的数字相加，相加所得的 p

个结果对 10 取模后再相乘，最终得到最大数。显然，fmax1[1，i，j]= g[i，j]；

fmin1[p，I，j]为 a_i、a_{i+1}、…a_j 分为 p 个部分，各部分内的数字相加，相加所得的 p 个结果对 10 取模后再相乘，最终得到最小数。显然，fmin1[1，i，j]= g[i，j]；

(1≤p≤m)

问题是当 a_i、a_{i+1}、…a_j 划分成的部分数 p 大于 1 时，怎么办。我们采用动态程序设计的方法计算。设

阶段 p：a_i、a_{i+1}、…a_j 划分成的部分数，2≤p≤m-1；

状态（i，j）:将 a_i、a_{i+1}、…a_j 划分成 p 个部分，1≤i≤n，i≤j≤n；

决策 k: 将 a_i、a_{i+1}、…a_k 划分成 p-1 个部分，a_{k+1}、…a_j 为第 p 部分，i≤k≤j-1；

显然，状态转移方程为

$$fmin1[p,i,j]=\min_{i\le k\le j-1}\{fmin1[p-1,i,k]*g[k+1,j]\}$$

$$fmax1[p,i,j]=\max_{i\le k\le j-1}\{fmax1[p-1,i,k]*g[k+1,j]\}$$

2≤p≤m-1

按照上述公式递推出 a_i、a_{i+1}、…a_j 划分成 m-1 个部分的最大值 fmax1[m-1，i，j]和最小值 fmin1[m-1，i，j]。由于圆周上的 n 个数首尾相接，因此第 m 部分设为 a_1…a_{i-1}、a_{j+1}…a_n。显然 a_1、a_2、…a_n 划分成 m 个部分的最大值 max 和最小值 min 为

$$max=\max_{\substack{1\le i\le n\\ i\le j\le n}}\{(g[1,i-1]+g[j+1,n])\bmod 10*fmax1[m-1,i,j]\mid(i\ne 1)or(j\ne n)\}$$

$$min=\min_{\substack{1\le i\le n\\ i\le j\le n}}\{(g[1,i-1]+g[j+1,n])\bmod 10*fmin1[m-1,i,j]\mid(i\ne 1)or(j\ne n)\}$$

由于 p-1 阶段仅和 p 阶段发生联系，因此我们将 p-1 阶段的状态转移方程 fmin1[p-1，i，j]设为 fmin1[i,j]、fmax1[p-1,i,j] 设为 fmax1[i,j]，将 p 阶段的状态转移方程 fmin1[p，i，j]设为 fmin[i，j]、fmax1[p，i，j]设为 fmax[i，j]。由此得出算法：

```
read(n，m);{读数字个数和划分的部分数}
fillchar(fmax1，sizeof(fmax1)，0); fillchar(fmin1，sizeof(fmin1)，$FF);
                                                {状态转移方程初始化}
fillchar(g，sizeof(g)，0);
for i:=1 to n do                          {依次读入每个数，一个数组成一部分}
    begin
     read(g[i，i]);
     g[i，i]:=(g[i，i] mod 10+10) mod 10;
     fmin1[i，i]:=g[i，i];fmax1[i，i]:=g[i，i];
    end;{for}
for i:=1 to n do                  {计算一部分内的数和对 10 的模的所有可能情况}
  for j:=i+1 to n do
   begin
    g[i，j]:=(g[i，j-1]+g[j，j]) mod 10;
```

```
    fmax1[i, j]:=g[i, j];fmin1[i, j]:=g[i, j];
   end;{for}
for p:=2 to m-1 do                {阶段：递推计算划分 2 部分…m-1 部分的结果值}
 begin
  fillchar(fmax,sizeof(fmax),0);          { 划分 p 部分的状态转移方程初始化}
  fillchar(fmin, sizeof(fmin), $FF);
  for i:=1 to n do                       {状态：枚举被划分为 p 部分的数字区间}
   for j:=i to n do
    for k:=i to j-1 do                          {决策：ai…ak被划分为 p-1 部分}
     begin
      if fmax1[i, k]*g[k+1, j]>fmax[i, j] then
                          {计算将 ai、ai+1、…aj划分成 p 个部分的状态转移方程}
        fmax[i, j]:=fmax1[i, k]*g[k+1, j];
      if(fmin1[i,k]>=0)and((fmin1[i,k]*g[k+1,j]<fmin[i,j])or(fmin[i,j]<0))
         then fmin[i, j]:=fmin1[i, k]*g[k+1, j];
     end;{for}
   fmin1:=fmin;fmax1:=fmax;
end;{for}
max:=0;min:=maxlongint;  {将 a1、a2、…an划分成 m 个部分的最大值和最小值初始化}
if m=1
  then                                {计算 n 个数划分成一部分的最大值和最小值}
     begin max:=g[1, n];min:=g[1, n];end{then}
  else for i:=1 to n do{将 a1…ai-1、aj+1…an   设为第 m 部分，计算最大值和最小值}
       for j:=i to n do
       if (i<>1) or (j<>n) then
       begin
        if (g[1, i-1]+g[j+1, n]) mod 10*fmax1[i, j]>max
           then max:=(g[1, i-1]+g[j+1, n]) mod 10*fmax1[i, j];
        if(fmin1[i, j]>=0) and ((g[1, i-1]+g[j+1, n]) mod 10*fmin1[i, j]<min)
           then min:=(g[1, i-1]+g[j+1, n]) mod 10*fmin1[i, j];
       end;{then}
writeln(min); writeln(max);                          {输出最小值和最大值}
```

本题的计算过程分两个阶段

第一个阶段：将圆周上的 n 个数排成一个序列，计算 a_i、a_{i+1}、…a_j划分成 m-1 个部分的最大值 fmax1[i, j]和最小值 fmin1[i, j]；

第二个阶段：将序列首尾相接。枚举第 m 部分的所有可能情况，在 fmax1 和 fmin1 的基础上，计算圆周上的 n 个数划分成 m 个部分的最大值 max 和最小值 min。

能否将两个阶段合并，用一个状态转移方程来解决呢？可以，这个问题留给读者思考。动态程序设计方法是多样性的。我们在用动态程序设计方法解题的时候，可以多想一想是否有其它的解法。对于不同的解法，要注意比较，好在哪里，差在哪里，从各种不同解法的比较中

作出合适的选择。

【例 4】装箱问题

有一个箱子容量为 v（正整数，0≤v≤20000），同时有 n 个物品（0<n≤30），每个物品有一个体积（正整数）。

要求从 n 个物品中，任取若干个装入箱内，使箱子的剩余空间为最小。

输入：

箱子的容量 v

物品数 n

接下来 n 行，分别表示这 n 个物品的体积

输出：

箱子剩余空间

输入输出样例

输入：
```
24
6
8
3
12
7
9
7
```
输出：
```
0
```

【算法分析】

1. 使用回溯法计算箱子的最小剩余空间

容量为 v 的箱子究竟应该装入哪些物品可使得剩余空间最小。显然在 n 个物品的体积和小于等于 v 的情况下，剩余空间为 v-n 个物品的体积和。但在 n 个物品的体积和大于 v 的情况下，没有一种可以直接找到问题解的数学方法。无奈之下，只能采用搜索的办法。设 a 和 s 为箱子的体积序列。其中 a[i]为箱子 i 的体积，s[i]为前 i 个箱子的体积和 $\sum_{k=1}^{i} a[k]$（1≤i≤n）；

best 为目前所有的装箱方案中最小的剩余空间。初始时 best=v；

确定搜索的几个关键因素：

状态（k，v'）：其中 k 为待装入箱子的物品序号，v'为箱子目前的剩余空间。

目标 v'<best：若箱子的剩余空间为目前为止最小，则 best 调整为 v'（best←v）；

边界条件（v'-（s[n]-s[k-1]））≥best：即便剩下的物品全部装入箱子(未装物品的体积和为 s[n]-s[k-1])，其剩余空间仍不小于 best，则放弃当前方案，回溯；

搜索范围：在未装入全部物品的前提下（k≤n），搜索两种可能情况：

若剩余空间装得下物品 k（v'≥a[k]），则物品 k 装入箱子，递归计算子状态（k+1，

v’-a[k]);
物品 k 不装入箱子，递归计算子状态（k+1，v’）；

我们用递归过程 search(k，v)描述这一搜索过程：

```
procedure search(k，v:integer);          {从状态(k，v)出发，递归计算最小剩余空间}
  begin
    if v<best then best←v;                    {若剩余空间为目前最小，则调整 best}
    if v-(s[n]-s[k-1])>=best    {若箱子即便装下全部物品，其剩余空间仍不小于 best，则回溯}
      then exit;
    if k<=n then                          {在未装入全部物品的前提下搜索两种可能情况}
       begin
         if v>=a[k]  {若剩余空间装得下物品 k，则物品 k 装入箱子，递归计算子状态}
              then search(k+1，v-a[k]);
          search(k+1，v);                      {物品 k 不装入箱子，递归计算子状态}
       end; {then}
    end; {search}
```

主程序如下：

```
读箱子体积 v;
读物品个数 n;
s[0] ←0;                                                    {物品装入前初始化}
for i←1 to n do                                        {输入和计算箱子的体积序列}
begin
  读第 i 个箱子的体积 a[i];
  s[i]←s[i-1]+a[i];
end; {for}
best←v;                                              {初始时，最小剩余空间为箱子体积}
if s[n]<=v then best←v-s[n] {若所有物品能全部装入箱子，则剩余空间为问题解}
           else search(1，v);       {否则从物品 1 出发，递归计算最小剩余空间}
输出最小剩余空间 best;
```

2. 使用动态程序设计方法计算箱子的最小剩余空间

如果按照物品序号依次考虑装箱顺序的话，则问题具有明显的阶段特征。问题是当前阶段的剩余空间最小，并不意味下一阶段的剩余空间也一定最小，即该问题并不具备最优子结构的特征。但如果将装箱的体积作为状态的话，则阶段间的状态转移关系顺其自然，可使得最优化问题变为判定性问题。设状态转移方程

f[i，j]——在前 i 个物品中选择若干个物品（必须包括物品 i）装箱，其体积正好为 j 的标志。显然 f[i，j]=f[i-1，j-box[i]]，即物品 i 装入箱子后的体积正好为 j 的前提是 f[i-1，j-box[i]]=true。初始时，f[0，0]=true（1≤i≤n，box[i]≤j≤v)。

由 f[i，j]=f[i-1，j-box[i]]可以看出，当前阶段的状态转移方程仅与上一阶段的状态转移方程相关。因此设 f0 为 i-1 阶段的状态转移方程，f1 为 i 阶段的状态转移方程，这

样可以将二维数组简化成一维数组。我们按照下述方法计算状态转移方程 f1：

```
fillchar(f0, sizeof(f0),0);                    {装箱前，状态转移方程初始化}
f0[0]←true;
for i←1 to n do                    {阶段 i:按照物品数递增的顺序考虑装箱情况}
  begin
    f1←f0;                                      {i 阶段的状态转移方程初始化}
    for j←box[i] to v do                  {状态 j：枚举所有可能的装箱体积}
      if f0[j-box[i]] then f1[j]←true;
                        {若物品 i 装入箱子后的体积正好为 j,则物品 i 装入箱子}
    f0←f1;                                                {记下当前装箱情况}
  end; {for}
```

经过上述运算，最优化问题转化为判定性问题。再借用动态程序设计的思想，计算装箱的最大体积 K=$\max_{j=v\cdots 0}\{j \mid f[n,j]=true\}$。显然最小剩余空间为 v-k:

```
for i←v downto 0 do                          {按照递减顺序枚举所有可能的体积}
  if f1[i] then
      begin      {若箱子能装入体积为 i 的物品，则输出剩余空间 v-i，并退出程序}
         writeln(v-i);  halt
      end; {then}
end.{for}
writeln(v);                                  {在未装入一个物品的情况下输出箱子体积}
```

【例 5】合唱队形

【问题描述】

N 位同学站成一排，音乐老师要请其中的(N-K)位同学出列，使得剩下的 K 位同学排成合唱队形。

合唱队形是指这样的一种队形：设 K 位同学从左到右依次编号为 1，2，…，K，他们的身高分别为 T_1，T_2，…，T_K，则他们的身高满足 $T_1 < T_2 < \cdots < T_i$ ，$T_i > T_{i+1} > \cdots > T_K$ (1≤i≤K)。

你的任务是，已知所有 N 位同学的身高，计算最少需要几位同学出列，可以使得剩下的同学排成合唱队形。

【输入文件】

输入文件 chorus.in 的第一行是一个整数 N（2 ≤ N ≤ 100），表示同学的总数。第一行有 n 个整数，用空格分隔，第 i 个整数 T_i（130 ≤ T_i ≤ 230）是第 i 位同学的身高（厘米）。

【输出文件】

输出文件 chorus.out 包括一行，这一行只包含一个整数，就是最少需要几位同学出列。

【样例输入】

```
8
186 186 150 200 160 130 197 220
```

【样例输出】

4

【数据规模】对于 50%的数据，保证有 n ≤ 20；对于全部的数据，保证有 n≤100。

【算法分析】

如果不去洞悉其间蕴涵的数学规律，直接在穷举或搜索所有可能排列的基础是求解的话，时间复杂度为 O (n*2^n)，1 秒时限内仅能通过 n≤20 范围内的测试数据。显然，这个算法的效率明显不符要求（），必须另辟新径。

解法 1：动态程序设计方法

我们按照由左而右和由右而左的顺序，将 n 个同学的身高排成数列。如何分别在这两个数列中寻求递增的、未必连续的最长子序列，就成为问题的关键。设

a 为身高序列，其中 a[i]为同学 i 的身高；

b 为由左而右身高递增的人数序列，其中 b[i]为同学 1‥同学 i 间（包括同学 i）身高满足递增顺序的最多人数。显然 b[i]= $\max_{1\le j\le i-1}$ {b[j]|同学 j 的身高<同学 i 的身高}+1；

c 为由右而左身高递增的人数序列，其中 c[i]为同学 n‥同学 i 间（包括同学 i）身高满足递增顺序的最多人数。显然 c[i]= $\max_{i+1\le j\le n}$ {c[j]|同学 j 的身高<同学 i 的身高}+1；

由上述状态转移方程可知,计算合唱队形的问题具备了最优子结构性质(要使 b[i]和 c[i]最大，子问题的解 b[j]和 c[k]必须最大(1≤j≤i-1 ，i+1≤k≤n))和重迭子问题的性质(为求得 b[i]和 c[i]，必须一一查阅子问题的解 b[1]‥b[i-1]和 c[i+1]‥c[n])，因此可采用动态程序设计的方法求解。

显然，合唱队的人数为 $\max_{1\le i\le n}\{b[i]+c[i]\}-1$（公式中同学 i 被重复计算，因此减 1），n 减去合唱队人数即为解。具体算法如下：

```
readln(n); {读学生数}
for i:=1 to n do read(a[i]);                                  {读每个学生的身高}
fillchar(b,sizeof(b),0);fillchar(c,sizeof(c),0);{身高满足递增顺序的两个队列初始化}
for i:=1 to n do                                              {按照由左而右的顺序计算 b 序列}
 begin
     b[i]:=1;
     for j:=1 to i-1 do if (a[i]>a[j])and(b[j]+1>b[i]) then b[i]:=b[j]+1;
 end;{for}
for i:=n downto 1 do                                          {按照由右而左的顺序计算 c 序列}
 begin
  c[i]:=1;
     for j:=i+1 to n do if (a[j]<a[i])and(c[j]+1>c[i])then c[i]:=c[j]+1;
   end;{for}
max:=0;                                          {计算合唱队的人数 max(其中 1 人被重复计算)}
for i:=1 to n do if b[i]+c[i]>max then max:=b[i]+c[i];
```

```
writeln(n-max+1);                                    {输出出列人数}
```

这个算法的时间复杂度为 $O(n^2)$，在 1 秒时限内可解决 n≤100 范围内的问题。但是，这个算法并不是最优算法，还可以精益求精。

解法 2：二分法

设 x 为当前身高满足递增顺序的队列，其中 x[i]为第 i 高的队员身高；a 序列、b 序列和 c 序列的定义如解法 1。

设 x[i]的初始值为∞(1≤i≤n)，b[0]为 0。我们从同学 1 出发，按照由左而右的顺序计算身高递增的人数序列 b。在计算 b[i]时，通过二分法找出区间 x[1]‥x[i]中身高矮于同学 i 的元素个数 min（x 区间的左右指针为 min 和 max，中间指针为 mid)。寻找过程如下：

```
min:=0;max:=i;{设 x 区间的左右指针}
while min<max-1 do{若 x 区间存在，则通过二分法计算同学 i 前身高矮于同学 i 且满足递增顺序的最多人数 min}
  begin
    mid:=(min+max) div 2;{计算中间指针}
    if x[mid]<a[i]
    then min:=mid{搜索右区间}
    else max:=mid{搜索左区间}
  end;{while}
```

显然，b[i]=min+1，x[min+1]=a[i]。

在计算出 b 序列后，再采用类似方法由右而左计算身高递增的人数序列 c。最后得出出列人数为 $n-\max_{1\le i\le n}\{b[i]+c[i]\}-1$（公式中同学 i 被重复计算，因此减 1)。具体算法如下：

```
readln(n);{读学生数}
for i:=1 to n do read(a[i]);{读每个学生的身高}
for i:=1 to n do x[i]:=maxlongint;{身高满足递增顺序的队列初始化}
b[0]:=0;
for i:=1 to n do{由左而右计算 b 序列}
begin
  min:=0;max:=i;{设 x 区间的左右指针}
  while min<max-1 do{若 x 区间存在，则通过二分法计算同学 i 前身高满足递增顺序的最多人数 min}
  begin
    mid:=(min+max) div 2;{计算中间指针}
    if x[mid]<a[i]
    then min:=mid{搜索右区间}
    else max:=mid{搜索左区间}
  end;{while}
  b[i]:=min+1;x[min+1]:=a[i]{同学 1‥同学 i 间（包括同学 i）最多有 min+1 个同学可
```

```
排成身高递增的队列}
  end;{for}
  for i:=1 to n do x[i]:=maxlongint; {身高满足递增顺序的队列初始化}
  c[0]:=0;
  for i:=n downto 1 do{由右而左计算 c 序列}
  begin
    min:=0;max:=i;
    while min<max-1 do
    begin
      mid:=(min+max) div 2; 设 x 区间的左右指针}
      if x[mid]<a[i]
      then min:=mid{搜索右区间}
      else max:=mid{搜索左区间}
    end;{while}
    c[i]:=min+1;x[min+1]:=a[i] {同学 n‥同学 i 间（包括同学 i）最多有 min+1 个同学可排成身高递增的队列}
  end;{max}
  max:=0;{计算合唱队的人数 max（其中 1 人被重复计算）}
  for i:=1 to n do if b[i]+c[i]>max then max:=b[i]+c[i];
  writeln(n+1-max);{输出出列人数}
```

第二种解法的算法的时间复杂度为 O(n*log n)。事实证明，采用动态程序设计方法解题并不一定是最优的。由于第一种解法是通过顺序枚举的途径计算 b 序列和 c 序列的，而第二种解法采用了二分法计算，比第一种解法更精确地揭示了问题本质，冗余运算相对减少，因此其时效自然要好一些。

【例 6】橱窗布置（Flower）

〖题目描述〗

假设以最美观的方式布置花店的橱窗，有 F 束花，每束花的品种都不一样，同时，至少有同样数量的花瓶，被按顺序摆成一行，花瓶的位置是固定的，并从左到右，从 1 到 V 顺序编号，V 是花瓶的数目，编号为 1 的花瓶在最左边，编号为 V 的花瓶在最右边，花束可以移动，并且每束花用 1 到 F 的整数惟一标识，标识花束的整数决定了花束在花瓶中列的顺序即如果 I ＜ J，则花束 I 必须放在花束 J 左边的花瓶中。

例如，假设杜鹃花的标识数为 1，秋海棠的标识数为 2，康乃馨的标识数为 3，所有的花束在放人花瓶时必须保持其标识数的顺序，即：杜鹃花必须放在秋海棠左边的花瓶中，秋海棠必须放在康乃馨左边的花瓶中。如果花瓶的数目大于花束的数目，则多余的花瓶必须空，即每个花瓶中只能放一束花。

每一个花瓶的形状和颜色也不相同，因此，当各个花瓶中放人不同的花束时会产生不同的美学效果，并以美学值(一个整数)来表示，空置花瓶的美学值为 0。在上述例子中，花瓶与花束的不同搭配所具有的美学值，可以用如下表格表示。

根据表格，杜鹃花放在花瓶 2 中，会显得非常好看，但若放在花瓶 4 中则显得很难看。

为取得最佳美学效果，必须在保持花束顺序的前提下，使花的摆放取得最大的美学值，

如果具有最大美学值的摆放方式不止一种，则输出任何一种方案即可。题中数据满足下面条件：1≤F≤100，F≤V≤100，－50≤A_{IJ}≤50，其中 AII 是花束 I 摆放在花瓶 J 中的美学值。输入整数 F，V 和矩阵(A_{IJ})，输出最大美学值和每束花摆放在各个花瓶中的花瓶编号。

	花瓶 1	花瓶 2	花瓶 3	花瓶 4	花瓶 5
杜鹃花	7	23	-5	-24	16
秋海棠	5	21	-4	10	23
康乃馨	-21	5	-4	-20	20

1、假设条件

1≤F≤100，其中 F 为花束的数量，花束编号从 1 至 F。

F≤V≤100，其中 V 是花瓶的数量。

-50≤A_{ij}≤50，其中 A_{ij}是花束 i 在花瓶 j 中的美学值。

2、输入

输入文件是 flower.in。

第一行包含两个数：F，V。

随后的 F 行中，每行包含 V 个整数，A_{ij} 即为输入文件中第（i+1 ）行中的第 j 个数。

3、输出

输出文件必须是名为 flower.out 的正文文件，文件应包含两行：

第一行是程序所产生摆放方式的美学值。

第二行必须用 F 个数表示摆放方式，即该行的第 K 个数表示花束 K 所在的花瓶的编号。

4、例子

flower.in:

```
3 5
7 23 -5 -24 16
5 21 -4 10 23
-21 5 -4 -20 20
```

flower.out:

```
53
2 4 5
```

【算法分析】

问题实际就是给定 F 束花和 V 个花瓶，以及各束花放到不同花瓶中的美学值，要求你找

出一种摆放的方案,使得在满足编号小的花放进编号小的花瓶中的条件下,美学值达到最大。

(1)将问题进行转化，找出问题的原型。首先，看一下上述题目的样例数据表格。

将摆放方案的要求用表格表现出来,则摆放方案需要满足:每行选且只选一个数(花瓶);摆放方案的相邻两行中，下面一行的花瓶编号要大于上面一行的花瓶编号两个条件。这时可将问题转化为：给定一个数字表格，要求编程计算从顶行至底行的一条路径，使得这条路径所经过的数字总和最大(要求每行选且仅选一个数字)。同时，路径中相邻两行的数字，必须保证下一行数字的列数大于上一行数字的列数。

看到经过转化后的问题，发现问题与例题 6 的数学三角形问题十分相似，数字三角形问题的题意是：

给定一个数字三角形，要求编程计算从顶至底的一条路径，使得路径所经过的数字总和最大(要求每行选且仅选一个数字)。同时，路径中相邻两行的数字，必须保证下一行数字的列数与上一行数字的列数相等或者等于上一行数字的列数加 1。

上例中已经知道：数字三角形中的经过数字之和最大的最佳路径，路径的每个中间点到最底层的路径必然也是最优的，可以用动态规划方法求解，对于“花店橱窗布置”问题经过转化后，也可采取同样的方法得出本题同样符合最优性原理。因此，可以对此题采用动态规划的方法。

(2)对问题原型动态规划方法的修改。“数字三角形”问题的动态规划方法为：已知它是用行数来划分阶段。假设用 a[i，j]表示三角形第 i 行的第 j 个数字，用 p [i，j]表示从最底层到 a[i,j]这个数字的最佳路径(路径经过的数字总和最大)的数字和，易得问题的动态转移方程为：

p[n+1,j]=0　　(1≤i≤n+1)

p[i,j]=max{p[i+1,j],p[i+1,j+1]}+a[i,j]

(1≤i≤i≤n，其中 n 为总行数)

分析两题的不同之处，就在于对路径的要求上。如果用 path[i]表示路径中第 i 行的数字编号,那么两题对路径的要求就是:“数字三角形”要求 path[i]≤path[i+1]≤path[i]+1，而本题则要求 path[i+1]>path[i]。

在明确两题的不同之后，就可以对动态规划方程进行修改了。假设用 b[i，j]表示美学值表格中第 i 行的第 j 个数字，用 q[i，j]表示从表格最底层到 b[i，j]这个数字的最佳路径(路径经过的数字总和最大)的数字和，修改后的动态规划转移方程为：

q[i,V+1]=-∞(1≤i≤F+1)

q[F，j]=b[F，j](1≤j≤V)

q[i，j]=max{q[i+1，k](j<K≤V+1)}+A[I,J](1≤I≤F，1≤J≤V)

这样,得出的max{q[1,k]}　(1≤j≤V)就是最大的美学值,算法的时间复杂度为 $O(FV^2)$。

(3)对算法时间效率的改进。先来看一下这样两个状态的求解方法：

q[i,j]=max{q[i+1，k] (j ＜ k≤V+1)}+b[i，j]　(1≤i≤F，1≤j≤V)

q[i,j+1]=max{q[i+1，k] (j+1 ＜ k≤V+1)}+a[i，j+1]　(1≤i≤F，　1≤j+1≤V)

上面两个状态中求 max{q[i+1，k]}的过程进行了大量重复的比较。此时对状态的表示稍作修改，用数组 t[i，j]=max{q[i,k](j≤k≤V+1)}(1≤i≤F，　1≤j≤V)表示新的状态。经过修改后，因为 q[i，j]=t[i+1，j+1]+a[i，j]，而 t[i，j]=max{t[i，j+1]，q[i,j](1≤i≤F，1≤j≤V)，所以得出新的状态转移方程：

t[i,V+1]=-∞　(1≤i≤F 十 1)

t[F，j]=max{t[F，j+1]，b[F，j]}　(1≤j≤V)

t[i,j]=max{t[i，j+1]，t[i+1，j+1]+a[i，j]}　(1≤i≤F，1≤j≤V)

这样，得出的最大美学值为 t[1,1]，新算法的时间复杂度为 O(F*V)，而空间复杂度也为 O(F*V)，完全可以满足 1≤F≤V≤100 的要求。下面给出这一问题的源程序。

【参考程序】

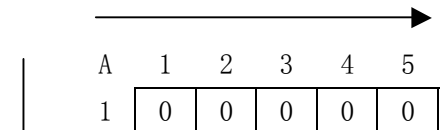
```
{$A+，B-，D+，E+，F-，G-，I+，L+，N-，O-，P-，Q-，R-，S+，T-，V+，X+}
{$M16384，0，655360}
program ex1;{花店橱窗布置问题}
    const st1='flower.in';      {输入文件名}
          st2='flower.out';     {输出文件名}
    var f,v:integer;  {f 为花束的数量；v 为花瓶的数量}
    b:array[1..100,1..100] of shortint;
                                    {b[i,j]为第 i 束花放进第 j 个花瓶的美学值}
    t:array[1..101,1..101] of integer;
         {t[i,¨为将第 i 到第 f 束花放进第 j 到第 v 个花瓶所可能得到的最大美学值}
    procedure readp;              {从文件中读入不同花束对应不同花瓶的美学值}
    var f1:text;
        i,j:integer;
    begin
      assign(f1,st1);reset(f1);
      readln (f1, f, v);
      for i:=1 to f do
        for j:=1 to v do
         read (f1, b [i, j]);
      close (f1);
    end;
  procedure main;             {用动态规划对问题求解}
   var i, j: integer;
   begin
      for i:=1 to f+1 do t[i, v+1]:=-9999;
      for j:=v downto 1 do
        if t[f,j+1]>b[f,j]
          then t[f,j]:=t[f, j+1]
          else t[f,j]:=b[f,j];{设定动态规划的初始条件,其中-9999 表示负无穷}
      for i:=f-1 downto 1 do
        for j:= v downto 1 do
          begin
            t[i, j]:=t[i, j+1];
            if t[i+1,j+1] + b[i,j] > t[i, j] then
               t[i, j]:=t[i+1, j+1] +b[i, j];
          end;
```

```
  end;
 procedure print;                          {将最佳美学效果和对应方案输出到文件}
  var f1: text;
      i,j,p:integer;
          {为当前需确定位置的花束编号，p 为第 i 束花应插入的花瓶编号的最小值}
  begin
    assign (f1, st2); rewrite (f1);
    writeln (f1, t[1,1]);
    p:=1;
    for i:=1 to f do
    begin                                  {用循环依次确定各束花应插入的花瓶}
      j:=p;
      while t[i, j] =t[i, p] do inc (j);
      write (f1, j-1,' '); p:=j;
    end;
    writeln (f1);
    close (f1);
  end;
begin
   readp;
   main;
   print;
end.
```

由此可看出，对于看似复杂的问题，通过转化就可变成简单的经典的动态规划问题。在问题原型的基础上，通过分析新问题与原问题的不同之处，修改状态转移方程，改变问题状态的描述和表示方式，就会降低问题规划和实现的难度，提高算法的效率。由此可见，动态规划问题中具体的规划方法将直接决定解决问题的难易程度和算法的时间与空间效率，而注意在具体的规划过程中的灵活性和技巧性将是动态规划方法提出的更高要求。

【例 7】方格取数

设有 n*n 的方格图（N≤8），我们将其中的某些方格中填入正整数，而其他的方格中则放入数字 0 。如图 1 1.2.5 所示（见样例）：



A	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	13	0	0	6	0	0
3	0	0	0	0	7	0	0	0
4	0	0	0	14	0	0	0	0
5	0	21	0	0	0	4	0	0
6	0	0	15	0	0	0	0	0

7	0	14	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

B

图 1 1.2.5

某人从图的左上角的A点出发，可以向下行走，也可以向右走，直到到达右下角的B点。在走过的路上，它可以取走方格中的数（取走后的方格中将变为数字 0 ）。此人从A点到B点共走两次，试找出 2 条这样的路径，使得取得的数之和最大。

【输入格式】

输入的第一行为一个整数N（表示N＊N的方格图），接下来的每行右三个整数，前两个表示位置，第三个数为该位置上所放的数。一行单独的 0 表示输入结束。

【输出格式】

只需输出一个整数，表示 2 条路径上取得的最大的和

【样例输入】

```
8
2 3 13
2 6 6
3 5 7
4 4 14
5 2 21
5 6 4
6 3 15
7 2 14
0 0 0
```

【样例输出】

67

【算法分析】

我们对这道题并不陌生。如果求一条数和最大的路径，读者自然会想到动态程序设计方法。现在的问题是，要找出这样的两条路径，是否也可以采用动态程序设计方法呢？回答是可以的。

1、状态的设计

对于本题来说，状态的选定和存储对整个问题的处理起了决定性的作用。

我们从（1，1）出发，每走一步作为一个阶段，则可以分成 2*n-1 个阶段：

第一个阶段，两条路径从（1，1）出发；

第二个阶段，两条路径可达（2，1）和（1，2）；

第三个阶段，两条路径可达的位置集合为（3，1）、（2，2）和（1，3）；

……

第 2*n-1 个阶段，两条路径汇聚（n，n）；

在第 k(1≤k≤2*n-1)个阶段，两条路径的终端坐标（x_1，y_1）（x_2，y_2）位于对应的右下对角线上。如图 1 1.2.6 所示：

如果我们将两条路径走第 i 步的所有可能位置定义为当前阶段的状态的话，面对的问

题就是如何存储状态了。方格取数问题的状态数目十分庞大，每一个位置是两维的，且又是求两条最佳路径，这就要求在存储上必须做一定的优化后才有可能实现算法的程序化。

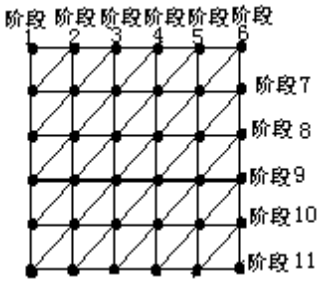


图 1 1.2.6

主要的优化就是：舍弃一切不必要的存储量。为此，我们取位置中的 X 坐标（x_1，x_2）作状态，其中

（$1 \leqslant x_1 \leqslant k$）∧（$x_1 \in \{1..n\}$）∧（$1 \leqslant x_2 \leqslant k$）∧（$x_2 \in \{1..n\}$）

直接由 X 坐标计算对应的 Y 坐标：

（$y_1=k+1-x_1$）∧（$y_1 \in \{1..n\}$）∧（$y_2=k+1-x_2$）∧（$y_2 \in \{1..n\}$）

2.状态转移方程

设两条路径在 k 阶段的状态为（x_1，x_2）。由（$y_1=k+1-x_1$）∧（$y_1 \in \{1..n\}$）得出第一条路径的坐标为（x_1，y_1）；由（$y_2=k+1-x_2$）∧（$y_2 \in \{1..n\}$）得出第二条路径的坐标为（x_2，y_2）。假设在 k-1 阶段，两条路径的状态为（x_1'，x_2'）且（x_1'，x_2'）位于（x_1，x_2）状态的左邻或下邻位置。因此我们设条路径的延伸方向为 d_1 和 d_2：$d_i=0$，表明第 i 条路径由（x_i'，y_i'）向右延伸至（x_i，y_i）；$d_i=1$，表明第 i 条路径由（x_i'，y_i'）向下延伸至（x_i，y_i）（$1 \leqslant i \leqslant 2$）。显然（$x_1' = x_2'$）∧（$d_1 = d_2$），表明两条路径重合，同时取走了（$x_1'$，$y_1'$）和（$x_1$，$y_1$）中的数，这种取法当然不可能得到最大数和的。

分析两种可能：

⑴若 $x_1=x_2$，则两条路径会合于 x_1 状态，可取走（x_1，y_1）中的数(图 11.2.6)；

x_1'（x_2'）→$x_1=x_2$

↑

x_2'（x_1'）

图 11.2.6

⑵若 $x_1 \neq x_2$，则在 k 阶段，第一条路径由 x_1' 状态延伸至 x_1 状态，第二条路径由 x_2' 状态延伸至 x_2 状态，两条路径可分别取走（x_1，y_1）和（x_2，y_2）中的数(图 11.2.7)；

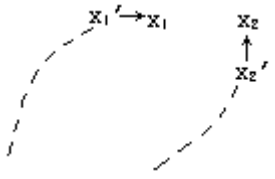


图 11.2.7

设

f[k，x_1，x_2]—在第 k 阶段，两条路径分别行至 x_1 状态和 x_2 状态的最大数和。显然

k=1 时，f[1，1，1]=0；

k≥2 时，f[k，x_1，x_2]=max{f[k-1， x_1'，x_2']+(x_1，y_1)的数字| x_1=x_2

f[k-1， x_1'，x_2']+(x_1，y_1)的数字+(x_2，y_2)的数字| x_1≠x_2}

1≤k≤2*n-1，x_1'∈可达 x_1的状态集合，x_2'∈可达 x_2的状态集合，（x_1'≠x_2'）∨(d_1≠d_2)；

上述状态转移方程符合最优子结构和重叠子问题的特性，因此适用于动态程序设计方法求解。由于第 k 个阶段的状态转移方程仅与第 k-1 个阶段的状态发生联系，因此不妨设

f_0—第 k-1 个阶段的状态转移方程；

f_1—第 k 个阶段的状态转移方程；

初始时，f_0[1，1]=0。经过 2*n-1 个阶段后得出的 f_0[n，n]即为问题的解。

3. 多进程决策的动态程序设计

由于方格取数问题是对两条路径进行最优化决策的，因此称这类动态程序设计为分阶段、多进程的最优化决策过程。设

阶段 i：准备走第 i 步（1≤i≤2*n-1)；

状态（x_1'，x_2'）： 第 i-1 步的状态号（1≤x_1'，x_2'≤i-1。x_1'，x_2'∈{1..n}）

决策（d_1，d_2)：第一条路径由 x_1'状态出发沿 d_1方向延伸、第二条路径由 x_2'状态 出发沿 d_2方向延伸，可使得两条路径的数和最大（0≤d_1，d_2≤1。方向 0 表示右移，方向 1 表示下移)；

具体计算过程如下：

```
fillchar(f0, sizeof(f0), 0);                    { 行走前的状态转移方程初始化}
f0[1, 1]←0;
for i←2 to n+n-1 do                                {阶段：准备走第 i 步}
  begin
  fillchar(f1, sizeof(f1), 0);                  { 走第 i 步的状态转移方程初始化}
  for x1' ←1 to i-1 do              {枚举两条路径在第 i-1 步时的状态 x1'和 x2'}
    for x2' ←1 to i-1 do
     begin
     计算 y1'和 y2';
     if (x1', y1')和(x2', y2')在界内 then
      for d1←0 to 1 do                    {决策：计算两条路径的最佳延伸方向}
       for d2←0 to 1 do
        begin
         第 1 条路径沿 d1方向延伸至(x1，y1);
         第 2 条路径沿 d2方向延伸至(x2，y2);
         if ((x1，y1)和(x2，y2)在界内)∧((d1≠d2)∨(x1≠x2)) {计算第 i 步的状态转移方程}
            then f1[x1，x2]←max{f0[x1' ，x2' ]+map[x1，y1] | x1=x2，f0[x1' ，
x2' ]+map[x1，y1]+map[x2，y2] | x1≠x2}
        end; {for}
     end; {for}
    f0←f1;                                          {记下第 i 步的状态转移方程}
  end; {for}
```

输出两条路径取得的最大数和 f0[n，n]；

【上机练习】

1、数塔问题(tower.pas)

有形如图 1 所示的数塔，从顶部出发，在每一结点可以选择向左走或是向右走，一起走到底层，要求找出一条路径，使路径上的值最大。

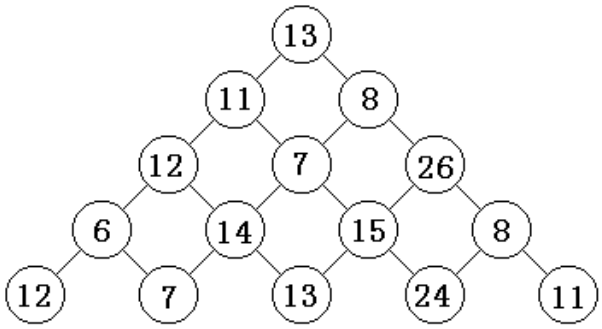


图 1

【输入输出样例】

输入：

5

13

11 8

12 7 26

6 14 15 8

12 7 13 24 11

输出：

max=86

2、拦截导弹(missile.pas)

某国为了防御敌国的导弹袭击，发展出一种导弹拦截系统。但是这种导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都不能高于前一发的高度。某天，雷达捕捉到敌国的导弹来袭。由于该系统还在试用阶段，所以只有一套系统，因此有可能不能拦截所有的导弹。

输入导弹依次飞来的高度（雷达给出的高度数据是不大于 30000 的正整数），计算这套系统最多能拦截多少导弹，如果要拦截所有导弹最少要配备多少套这种导弹拦截系统。

【输入输出样例】

输入：

389 207 155 300 299 170 158 65

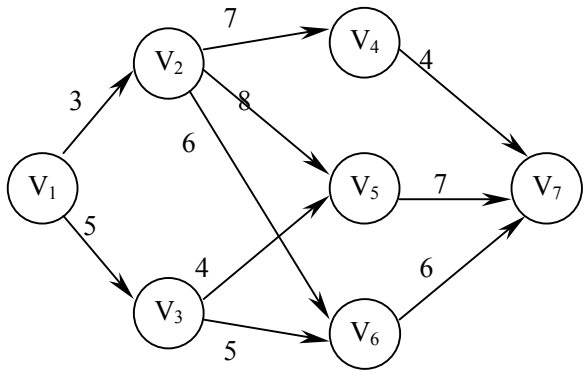
输出：

6（最多能拦截的导弹数）

2（要拦截所有导弹最少要配备的系统数）

3、最短路径（short.pas）

在下图中找出从起点到终点的最短路径。



[样例输入]

short.in

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

[样例输出]

short.out

```
minlong=14
1  2  4  7
```

4、骑士游历问题(knight.pas)

5、砝码称重(weight.pas)

6、数字游戏（Game.pas）

7、装箱问题(boxes.pas)

8、合唱队形（Chorus.pas）

9、橱窗布置（Flower.pas）

10、方格取数(pane.PAS)

第四节　背包问题

【例 1】0/1 背包

【问题描述】

一个旅行者有一个最多能用 m 公斤的背包，现在有 n 件物品，它们的重量分别是 W1，W2，...,Wn,它们的价值分别为 C1,C2,...,Cn.若每种物品只有一件求旅行者能获得最大总价值。

【输入格式】

第一行：两个整数，M(背包容量，M<=200)和 N(物品数量，N<=30)；

第 2..N+1 行：每行二个整数 Wi,Ci，表示每个物品的重量和价值。

【输出格式】

仅一行，一个数，表示最大总价值。

【样例输入】

package.in

10 4

2 1

3 3

4 5

7 9

【样例输出】

package.out

12

【解法一：动态规划】

【算法分析】

显然这个题可用深度优先方法对每件物品进行枚举(选或不选用 0,1 控制)。程序简单,但是当 n 的值很大的时候不能满足时间要求，时间复杂度为 $O(2^n)$。按递归的思想我们可以把问题分解为子问题,使用递归函数 。

设 f(i，x)表示前 i 件物品，总重量不超过 x 的最优价值

则 f（i，x）=max(f(i－1，x-W[i])+C[i]，f（i－1，x））

f（n，m）即为最优解，边界条件为 f（0，x）＝0 ，f(i,0)=0;

下面例出 F[I，X]的值，I 表示前 I 件物品，X 表示重量

	F[I,1]	F[I,2]	F[I,3]	F[I,4]	F[I,5]	F[I,6]	F[I,7]	F[I,8]	F[I,9]	F[I,10]
I=1	0	1	1	1	1	1	1	1	1	1
I=2	0	1	3	3	4	4	4	4	4	4

I=3	0	1	3	5	5	6	8	8	9	9
I=4	0	1	3	5	5	6	9	9	10	12

动态规划方法(顺推法):

【参考程序】

```
program package;
 const
      maxm=200;maxn=30;
 type
      ar=array[1..maxn] of integer;
 var
    m,n,j,i:integer;
    c,w:ar;
    f:array[0..maxn,0..maxm] of integer;

 function max(x,y:integer):integer;
 begin
  if x>y then max:=x else max:=y;
 end;

BEGIN
  assign(input,' package.in');
  assign(output,' package.out');
  reset(input); rewrite(output);
  readln(m,n);
  for i:= 1 to n do
     readln(w[i],c[i]);
  for i:=1 to m do f[0,i]:=0;
  for i:=1 to n do f[i,0]:=0;
  for i:=1 to n do
   for j:=1 to m do
    begin
      if j>=w[i]  then f[i,j]:=max(f[i-1,j-w[i]]+c[i],f[i-1,j])
                  else f[i,j]:=f[i-1,j];
    end;
  writeln(f[n,m]);
  close(input);
  close(output);
END.
```

使用二维数组存储各子问题时方便，但当 maxm 较大时如 maxn=2000 时不能定义二维数组 f,怎么办,其实可以用一维数组,但是上述中 j:=1 to m 要改为 j:=m downto 1,为什么?请大家自己解决。

【解法二：深搜算法】

【算法分析】

设 n 件物品的重量分别为 $w_1, w_2, \cdots, w_n$;，物品的价值分别为 $v_1, v_2, \cdots, v_n$。采用递归寻找物品的选择方案。设前面已有了多种选择的方案，并保留了其中总价值最大的方案于数组 result 中，该方案的总价值存于变量 maxv。当前正在考察某一新的方案，其物品选择情况保存于数组 option 中。假定当前方案已考虑了前 i-1 件物品，现在要考虑第 i 件物品；当前方案已包含的物品的重量之和为 tw；至此，若其余物品都选择是可能的话，本方案能达到的总价值的期望值设为 tv。算法引入 tv 是当一旦当前方案的总价值的期望值也小于前面方案的总价值 maxv 时，继续考察当前方案变成无意义的工作，应终止当前方案，立即去考察下一个方案。因为当方案的总价值不比 maxv 大时，该方案不会再被考察。这同时保证后面找到的方案一定会比前面的方案更好。

对于第 i 件物品的选择有两种可能：

（1）物品 i 被选择，这种可能性仅当包含它不会超过方案总重量的限制时才是可行的。选中后，继续递归去考虑其余物品的选择；

（2）物品 i 不被选择，这种可能性仅当不包含物品 i 也有可能会找到价值更大的方案的情况。

按以上思想写出递归算法如下：

【算法】背包问题，找最佳方案

```
try (物品 i,当前选择已达到的重量和 tw,本方案可能达到的总价值为 tv)
begin
    {考虑物品 i 包含在当前方案中的可能性}
    if 包含物品 i 是可接受的 then
    begin
        将物品 i 包含在当前方案中;
        if i＜n then try(i＋l,tw 十物品 i 的重量,tv);
           else {又一个完整方案，因它比前面的方案好，以它作为最佳方案}
                以当前方案作为临时最佳方案保存
        恢复物品 i 不包含状态;
    end;
    {考虑物品 i 不包含在当前方案中的可能性}
     if 不包含物品 i 仅是可考虑的 then
        if i＜n then try(i＋l, tw, tv-物品 i 的价值);
           else {又一个完整方案，因它比前面的方案好，以它作为最佳方案。}
                以当前方案作为临时最佳方案保存;
end;
```

按上述算法编写函数和程序如下：

【参考程序】

```
const maxn=20;
var i,n,limitw,maxv,totalv:longint;
    w,v:array[1..maxn] of longint;
    result,option:array[1..maxn] of boolean;
procedure try(i,tw,tv:longint);
var k:longint;
begin
     if tw+w[i]<=limitw then
     begin
          option[i]:=true;
          if i<n then try(i+1,tw+w[i],tv)
                 else begin for k:=1 to n do result[k]:=option[k];
                            maxv:=tv end;
          option[i]:=false
     end;
     if tv-v[i]>maxv then
        if i<n then try(i+1,tw,tv-v[i])
               else begin for k:=1 to n do result[k]:=option[k];
                          maxv:=tv-v[i] end
end;

BEGIN
     write('输入物品种数 n:'); readln(n);
     writeln('输入各物品的重量和价值:');
     totalv:=0;
     for i:=1 to n do
     begin
          write('Input w[',i,'],v[',i,']:');
          readln(w[i],v[i]);
          totalv:=totalv+v[i]
     end;
     write('输入限制重量 limitw:'); readln(limitw);
     maxv:=0;
     for i:=1 to n do option[i]:=false;
     try(1,0,totalv);
     write('选择方案为:');
     for i:=1 to n do if result[i] then write(i,' ');
     writeln;
     writeln('总价值为:',maxv)
```

END.

【例 2】完全背包

【问题描述】

设有 n 种物品，每种物品有一个重量及一个价值。但每种物品的数量是无限的，同时有一个背包，最大载重量为 M，今从 n 种物品中选取若干件(同一种物品可以多次选取)，使其重量的和小于等于 M，而价值的和为最大。

【输入格式】

第一行：两个整数，M(背包容量，M<=200)和 N(物品数量，N<=30)；

第 2..N+1 行：每行二个整数 Wi,Ui，表示每个物品的重量和价值。

【输出格式】

仅一行，一个数，表示最大总价值。

【样例一输入】

knapsack.in

```
10 4
2  1
3  3
4  5
7  9
```

【样例一输出】

knapsack.out

```
max=12
no.1   weight: 2   worth: 1   get: 0
no.2   weight: 3   worth: 3   get: 1
no.3   weight: 4   worth: 5   get: 0
no.4   weight: 7   worth: 9   get: 1
```

【样例二输入】

knapsack.in

```
12 4
2  1
3  3
4  5
7  9
```

【样例二输出】

knapsack.out

```
max=15
no.1   weight: 2   worth: 1   get: 0
no.2   weight: 3   worth: 3   get: 0
no.3   weight: 4   worth: 5   get: 3
```

no.4 weight: 7 worth: 9 get: 0

【解法一】

【算法分析】

• 记 W(1)，W(2)，…，W(N)每种物品的重量，U(1)，U(2)，…，U(N)为每种物品的价值，X(1)，X(2)，…，X(N)为每种物品选取的数量，问题成为满足：

条件 Σ X(I)W(I)≤M 且 Σ X(I)U(I) 为最大。

• 记 FK（Y）为取前 K 种物品，限制重量为 Y 时取得价值最大，则：

F0(Y)=0，即对一切 Y，一件物品都不取时，最大价值为 0；

FK(0)=0，即最大重量限制为 0 时，不能取得任何物品，所以最大价值也为 0；

F1(Y)=[y/w1]u1，即仅取第一种物品，则最大价值为尽可能多的装第一种物品，所以能装的数量为[y/w1]，而得到的价值为[y/w1]u1。

一般公式:FK(Y)=max{ FK-1(Y)，FK(Y-WK)+UK},并约定，当 Y<0 时，FK(Y)=负无穷大。

下面用一个具体的例子来说明求解的过程。

M=10, N=4

W1=2，W2=3，W3=4，W4=7

U1=1，U2=3，U3=5，U4=9

下面列出 FK(Y)：

X \ Y	1	2	3	4	5	6	7	8	9	10	
1	0	1	1	2	2	3	3	4	4	5	注 1
2	0	1	3	3	4	6	6	7	9	9	注 2
3	0	1	3	5	5	6	8	10	10	11	注 3
4	0	1	3	5	5	6	9	10	10	12	注 4

注 1 • 第 1 行表示 k=1，即取第一种物品，当 Y=1 时，无法取，所以 F1(1)=0，当 y=2 时，可取 1 个第一种物品，价值为 1，…，当 y=10(即 M)，此时，可取 5 个第一种物品，价值为 5。

注 2 • 当可以取 2 种物品时。

当 y=1 时，为 0(什么都不能取)；

当 y=2 时，仅能取第一种物品，所以 F2(2)=1；

当 y=3 时，有 2 种取法。取一个第一种物品，价值为 1；取一个第二种物品，价值为 3，取大者，所以 F2(3)=3。

当 y=4 也有 2 种取法，取第一种物品 2 件，价值为 2；取第二种物品 1 件价值为 3，所以 F2(4)=3。

当 y=5 时，有 2 种取法，取第一种物品 2 件，价值为 2；取第一种物品，第二种物品各 1 件，价值为 4，所以 F2(5)=4。以此类推。计算 F2(10)时，有 2 种考虑，第一种是全部取第一种，即 F1(10)=5；第二种是由 F2(7)+3=9，取大者，得到 F2(10)=9。

注 3·

注 4·F4(10)的计算方法为:

上面给出的是计算 F(I，J)的方法，但是还不能确定每种物品的选取个数。下面我们用倒推法来求出每种物品的个数。记 F(N，M)为目标，检查:F(N-1，M)与 F(N，M-WN)+UN，若前者大，则无输出，由 F(N，M)-> F(N-1，M)。若后者大，则输出 WN，计算 F(N，M-WN)。当 N-1，N-2，…，到达 1 时，则全部输出。

【参考程序】:（顺推法）

```
Program knapsack;
  Const  maxm=200;maxn=30;
  type
       tlist=array[1..maxn] of byte;
       tmake=array[0..maxn,0..maxm] of integer;
var i,n,m:integer;
    w,u:tlist;
    f:tmake;
procedure make;
  var i,j:byte;
begin
  for i:=1 to n do
  begin
    for j:=1 to w[i]-1 do
      f[i,j]:=f[i-1,j];
    for j:=w[i] to m do
    if f[i-1,j]>f[i,j-w[i]]+u[i] then   f[i,j]:=f[i-1,j]
                        else  f[i,j]:=f[i,j-w[i]]+u[i];
  end;
end;

procedure print;
var get:tlist;
   i,j:byte;
begin
  fillchar(get,sizeof(get),0);
  i:=n; j:=m;
    while i>0 do
       if f[i,j]=f[i-1,j] then dec(i)
         else begin
                dec(j,w[i]);
                inc(get[i]) ;
              end;
```

```
    writeln('max=', f[n,m]);
    for i:=1 to n do
      writeln('no.',i,'  weight:',w[i]:2,'  worth:',u[i]:2,'  get:',get[i]:2);
end;

BEGIN
  assign(input,'knapsack.in');
  assign(output,'knapsack.out');
  reset(input); rewrite(output);
  fillchar(w,sizeof(w),0);
  fillchar(u,sizeof(u),0);
  readln(m,n);
  for i:=1 to n do
       read(w[i],u[i]);
  make;
  print; {或者 writeln('max=', f[n,m]);}
  close(input); close(output);
END.
```

【解法二】

【算法分析】

本问题的数学模型如下：

设 f(x)表示重量不超过 x 公斤的最大价值， 则 f(x)=max{f(x-w[i])+u[i]}

当 x>=w[i] 1<=i<=n

【样例一】中间结果

下面例出 F[X]表示重量不超过 x 公斤的最大价值，X 表示重量

	F[1]	F[2]	F[3]	F[4]	F[5]	F[6]	F[7]	F[8]	F[9]	F[10]
X=1	0									
X=2	0	1								
X=3	0	1	3							
X=4	0	1	3	5						
X=5	0	1	3	5	5					
X=6	0	1	3	5	5	6				
X=7	0	1	3	5	5	6	9			
X=8	0	1	3	5	5	6	9	10		
X=9	0	1	3	5	5	6	9	10	10	
X=10	0	1	3	5	5	6	9	10	10	12

【样例二】中间结果

下面例出 F[X]表示重量不超过 x 公斤的最大价值，X 表示重量

	F[1]	F[2]	F[3]	F[4]	F[5]	F[6]	F[7]	F[8]	F[9]	F[10]	F[11]	F[12]
X=1	0											
X=2	0	1										
X=3	0	1	3									
X=4	0	1	3	5								
X=5	0	1	3	5	5							
X=6	0	1	3	5	5	6						
X=7	0	1	3	5	5	6	9					
X=8	0	1	3	5	5	6	9	10				
X=9	0	1	3	5	5	6	9	10	10			
X=10	0	1	3	5	5	6	9	10	10	12		
X=11	0	1	3	5	5	6	9	10	10	12	14	
X=12	0	1	3	5	5	6	9	10	10	12	14	15

【参考程序】：（顺推法）

```
program knapsack04;
 const  maxm=200;maxn=30;
 type   ar=array[0..maxn] of integer;
 var
    m,n,j,i,t:integer;
    u,w:ar;
    f:array[0..maxm] of integer;
BEGIN
  assign(input,'knapsack.in');
  assign(output,'knapsack.out');
  reset(input); rewrite(output);
  readln(m,n);
  for i:= 1 to n do
     readln(w[i],u[i]);
  f[0]:=0;
  for i:=1 to m do
   for j:=1 to n do
    begin
       if i>=w[j]  then t:=f[i-w[j]]+u[j];
       if t>f[i] then f[i]:=t
    end;
  writeln( 'max=' ,f[m]);
  close(input);
  close(output);
END.
```

【上机练习】

1、0/1 背包

【问题描述】

一个旅行者有一个最多能用 m 公斤的背包，现在有 n 件物品，它们的重量分别是 W1，W2，...,Wn,它们的价值分别为 C1,C2,...,Cn。若每种物品只有一件，求旅行者能获得最大总价值。

【输入格式】

第一行：两个整数，M(背包容量，M<=200)和 N(物品数量，N<=30)；

第 2..N+1 行：每行二个整数 Wi,Ci，表示每个物品的重量和价值。

【输出格式】

仅一行，一个数，表示最大总价值。

【样例输入】

package.in

```
10 4
2  1
3  3
4  5
7  9
```

【样例输出】

package.out

```
12
```

2、完全背包

【问题描述】

设有 n 种物品，每种物品有一个重量及一个价值。但每种物品的数量是无限的，同时有一个背包，最大载重量为 M，今从 n 种物品中选取若干件(同一种物品可以多次选取)，使其重量的和小于等于 M，而价值的和为最大。

【输入格式】

第一行：两个整数，M(背包容量，M<=200)和 N(物品数量，N<=30)；

第 2..N+1 行：每行二个整数 Wi,Ui，表示每个物品的重量和价值。

【输出格式】

仅一行，一个数，表示最大总价值。

【样例输入】

knapsack.in

```
10 4
2  1
3  3
4  5
7  9
```

【样例输出】

knapsack.out

max=12

3、货币系统(Money Systems)

【问题描述】

母牛们不但创建了他们自己的政府而且建立了自己的货币系统。他们对货币的数值感到好奇。传统地，一个货币系统是由 1，5，10，20 或 25，50，100 的单位面值组成的。母牛想知道用货币系统中的货币来构造一个确定的面值，有多少种不同的方法。

举例来说，使用一个货币系统 {1,2,5,10,...}产生 18 单位面值的一些可能的方法是:18x1, 9x2, 8x2+2x1, 3x5+2+1 等等其它。

写一个程序来计算有多少种方法，用给定的货币系统来构造一个确定的面值。

【输入格式】

货币系统中货币的种类数目是 V。 (1<= V<=25)

要构造的面值是 N。 (1<= N<=10,000)

第 1 行:	二个整数，V 和 N
第 2 ..V+1 行：	可用的货币 V 个整数 (每行一个)。

【输出格式】

单独的一行包含那个可能的构造的方案数。

【样例输入】

money.in

3 10

1

2

5

【样例输出】

money.out

10

4、竞赛总分(Score Inflation)

【问题描述】

学生在我们 USACO 的竞赛中的得分越多我们越高兴。我们试着设计我们的竞赛以便人们能尽可能的多得分。

现在要进行一次竞赛，总时间 T 固定，有若干类型可选择的题目，每种类型题目可选入的数量不限，每种类型题目有一个 si(解答此题所得的分数)和 ti(解答此题所需的时间)，现要选择若干题目，使解这些题的总时间在 T 以内的前提下，所得的总分最大。

输入包括竞赛的时间,M(1 <= M <= 10000)和题目类型数目 N(1 <= N <= 10000)。

后面的每一行将包括两个整数来描述一种"题型":

第一个整数说明解决这种题目能得的分数(1 <= points <= 10000),第二整数说明解决这种题目所需的时间(1 <= minutes <= 10000)。

【输入格式】

第 1 行:	两个整数：竞赛的时间 M 和题目类型数目 N。
第 2-N+1 行:	两个整数：每种类型题目的分数和耗时。

【输出格式】

单独的一行，在给定固定时间里得到的最大的分数。

【样例输入】

inflate.in

300 4
100 60
250 120
120 100
35 20

【样例输出】

inflate.out

605 {从第 2 种类型中选两题和第 4 种类型中选三题}

5、质数和分解（PRIME.PAS）

【问题描述】

任何大于 1 的自然数 N，都可以写成若干个大于等于2且小于等于 N 的质数之和表达式(包括只有一个数构成的和表达式的情况)，并且可能有不止一种质数和的形式。例如9 的质数和表达式就有四种本质不同的形式：9 = 2+5+2 = 2+3+2+2 = 3+3+3 = 2+7 。

这里所谓两个本质相同的表达式是指可以通过交换其中一个表达式中参加和运算的各个数的位置而直接得到另一个表达式。

试编程求解自然数 N 可以写成多少种本质不同的质数和表达式。

【输入文件】（PRIME.IN）：

文件中的每一行存放一个自然数 N ， 2≤N≤200。

【输出文件】（PRIME.OUT）：

依次输出每一个自然数 N 的本质不同的质数和表达式的数目。

样例一：

PRIME.IN
2
PRIME.OUT
1

样例二：

PRIME.IN
200
PRIME.OUT

9845164

第五节 动态规划应用举例

例 1、挖地雷（Mine.pas）

【问题描述】

在一个地图上有 N 个地窖（N<=200）,每个地窖中埋有一定数量的地雷。同时，给出地窖之间的连接路径，并规定路径都是单向的。某人可以从任一处开始挖地雷，然后沿着指出的连接往下挖（仅能选择一条路径），当无连接时挖地雷工作结束。设计一个挖地雷的方案，使他能挖到最多的地雷。

【输入格式】

N {地窖的个数}

W_1，W_2，……W_N {每个地窖中的地雷数}

X1，Y1 {表示从 X1 可到 Y1}

X2，Y2

……

0，0 {表示输入结束}

【输出格式】

K1——K2——……——Kv {挖地雷的顺序}

MAX {最多挖出的地雷数}

【输入样例】(Mine.in)

```
6
5 10 20 5 4 5
1 2
1 4
2 4
3 4
4 5
4 6
5 6
0 0
```

【输出样例】(Mine.out)

```
3-4-5-6
34
```

【算法分析】

本题是一个经典的动态规划问题。很明显，题目规定所有路径都是单向的，所以满足无后效性原则和最优化原理。设 W（i）为第 i 个地窖所藏有的地雷数，A（i,j）表示第 i 个

地窖与第 j 个地窖之间是否有通路，F（i）为从第 i 个地窖开始最多可以挖出的地雷数，则有如下递归式：

F（i）=MAX { F(j) + W(i) }　　(i<j<=n , A(i,j)=1)

边界：F（n）=W(n)

于是就可以通过递推的方法，从后(F（n）)往前逐个找出所有的 F（i），再从中找一个最大的即为问题 2 的解。对于具体所走的路径（问题 1），可以通过一个向后的链接来实现，具体请看下面的程序清单和注解。

【参考程序】

```
program Mine;
var f:array[1..200,1..2] of longint;
    a:array[1..200,1..200] of boolean;
    w:array[1..200] of longint;
    n,i,j,x,y,L,k,max:longint;
begin
  assign(input,'mine.in');
  assign(output,'mine.out');
  reset(input); rewrite(output);
  fillchar(a,sizeof(a),false);
  fillchar(f,sizeof(f),0);
  readln(n);
  for i:=1 to n do
  read(w[i]);readln;
  repeat
    readln(x,y);
    if (x<>0) and (y<>0) then a[x,y]:=true;
  until (x=0) and (y=0);
  f[n,1]:=w[n];
  for i:=n-1 downto 1 do
    begin
      L:=0;
      k:=0;
      for j:=i+1 to n do
      if a[i,j] and (f[j,1]>L) then
                begin  L:=f[j,1]; k:=j; end;
      f[i,1]:=L+w[i];
      f[i,2]:=k;
    end;
  L:=1;
  for j:=2 to n do
     if f[j,1]>f[L,1] then L:=j;
  max:=f[L,1];write(L);
```

```
  L:=f[L,2];
  while L<>0 do
    begin
      write('-',L);
      L:=f[L,2];
    end;
  writeln;
  writeln(max);
  close(input);close(output);
end.
```

例 2、防卫导弹（Catcher.pas）

【问题描述】

一种新型的防卫导弹可截击多个攻击导弹。它可以向前飞行，也可以用很快的速度向下飞行，可以毫无损伤地截击进攻导弹，但不可以向后或向上飞行。但有一个缺点，尽管它发射时可以达到任意高度，但它只能截击比它上次截击导弹时所处高度低或者高度相同的导弹。现对这种新型 防卫导弹进行测试，在每一次测试中，发射一系列的测试导弹（这些导弹发射的间隔时间固定，飞行速度相同），该防卫导弹所能获得的信息包括各进攻导弹的高度，以及它们发射次序。现要求编一程序，求在每次测试中，该防卫导弹最多能截击的进攻导弹数量，一个导弹能被截击应满足下列两个条件之一：

1、它是该次测试中第一个被防卫导弹截击的导弹；

2、它是在上一次被截击导弹的发射后发射，且高度不大于上一次被截击导弹的高度的导弹。

【输入格式】

从当前目录下的文本文件“CATCHER.IN”读入数据。该文件的第一行是一个整数 N（0〈=N〈=4000〉，表示本次测试中，发射的进攻导弹数，以下 N 行每行各有一个整数 hi（0〈=hi〈=32767〉，表示第 i 个进攻导弹的高度。文件中各行的行首、行末无多余空格，输入文件中给出的导弹是按发射顺序排列的。

【输出格式】

答案输出到当前目录下的文本文件“CATCHER.OUT”中，该文件第一行是一个整数 max，表示最多能截击的进攻导弹数，以下的 max 行每行各有一个整数，表示各个被截击的进攻导弹的编号（按被截击的先后顺序排列）。输出的答案可能不唯一，只要输出其中任一解即可。

【输入输出样例】

输入文件：CATCHER.IN
3 25 36 23

输出文件：CATCHER.OUT
2 1 3

题目讲得很麻烦,归根结底就是求一整串数中的最长不上升序列

这道题目一开始我使用回溯算法,大概可以拿到 1/3 的分吧,后来发现这其实是动态规划算法中最基础的题目,用一个二维数组 C[1..Max,1..2]来建立动态规划状态转移方程

(注:C[1..Max,1]表示当前状态最多可击落的导弹数,C[1..Max,2]表示当前状态的前继标志):Ci=Max{C[j]+1,(j=i+1..n)},然后程序也就不难实现了.

【参考程序】

```
program catcher_hh;
var  f:text;
     i,j,k,max,n,num:integer;
     a:array [1..4000] of integer;          {导弹高度数组}
     c:array [1..4000,1..2] of integer;     {动态规划数组}
procedure readfile;
begin
   assign(f,'catcher.in'); reset(f);
   readln(f,num);
   for i:=1 to num do
   readln(f,a[i]);
end;
procedure work;
begin
   fillchar(c,sizeof(c),0); c[num,1]:=1;      {清空数组,赋初值}
   {开始进行动态规划}
   for i:=num-1 downto 1 do
   begin
       n:=0; max:=1;
       for j:=i+1 to num do
       if (a[i]>a[j]) and (max<1+c[j,1])
       then begin n:=j; max:=1+c[j,1]; end;
       c[i,1]:=max; c[i,2]:=n;
   end;
   writeln; writeln('Max : ',max);    {打印最大值}
   max:=0; n:=0;
   for i:=1 to num do
   if c[i,1]>max then begin max:=c[i,1]; n:=i; end;
   repeat  {返回寻找路径}
       writeln(n,' '); n:=c[n,2];
   until n=0;
end;
begin
   readfile; work;
end.
```

例 3、轮船问题(ship.pas)

〖题目描述〗

某国家被一条河划分为南北两部分，在南岸和北岸总共有 N 对城市，每一城市在对岸都有唯一的友好城市，任何两个城市都没有相同的友好城市。每一对友好城市都希望有一条航线来往，于是他们向政府提出了申请。由于河终年有雾。政府决定允许开通的航线就互不交叉（如果两条航线交叉，将有很大机会撞船）。

你的任务是编写一个程序来帮政府官员决定他们应拨款兴建哪些航线以使在安全条件下有最多航线可以被开通。

【输入格式】

输入文件(ship.in):包括了若干组数据，每组数据格式如下：

第一行两个由空格分隔的整数 x，y，10〈=x〈=6000，10〈=y〈=100。x 表示河的长度而 y 表示宽。第二行是一个整数 N(1<=N<=5000)，表示分布在河两岸的城市对数。接下来的 N 行每行有两个由空格分隔的正数 C，D（C、D〈=x〉，描述每一对友好城市沿着河岸与西边境线的距离，C 表示北岸城市的距离而 D 表示南岸城市的距离。在河的同一边，任何两个城市的位置都是不同的。整个输入文件以由空格分隔的两个 0 结束。

【输出格式】

输出文件(ship.out)：要在连续的若干行里给出每一组数据在安全条件下能够开通的最大航线数目。

【输入输出样例】

Ship.in	Ship.out
30 4	4
7	
22 4	
2 6	
10 3	
15 12	
9 8	
17 17	
4 2	
0 0	

【算法分析】

对这道题的最一般想法是进行回溯，但是回溯对于数据规模达到 5000 的情况是不可行的。

于是我们改变一下思路，将每对友好城市看成一条线段，则这道题的描述化为：有 N 条线段，问最少去掉多少条线，可以使剩下的线段互不交叉？

顺理成章，删掉的线应该是和其它线交叉最多的，其"交叉数"最大。所谓"交叉数"是指某线与其它线的交叉情况，初始值为 0，若和其它线交叉则加 1 所得的值。按"交叉数"从大到小删除线，直到所有线都不交叉为止。此时，我们要解决的问题有：1、如何计算交叉数？

2、怎么删线？

对第一个问题，以北岸为线的起点而南岸为线的终点；先将所有的线按照起点坐标值从小到大排序，按照每条线的终点坐标计算交叉数：求线 I 的交叉数 J[I]，则检查所有 1..I-1 条线，若线 J（ 1<＝J＜I ）的终点值大于线 I 的终点值，则线 I 与线 J 相交。J[I]与 J[J]同时加 1。整个搜索量最大为 5000 X 5000。

对第二个问题，将 J 数组从大到小排序，每删除一条线，则将与之相交的线的 J 值减 1，重复这个过程，直到所有 J 值都为 0。此时剩下的线则全不交叉。

如上数据，则可得下面结果：

编号	南岸	北岸	交叉数
1	1	3	1
2	2	4	2
3	3	1	2
4	4	5	1
5	4	2	2

此时，2、3、5 航线的交叉数都一样，如果删去的是 5、3 线，则剩下的 1、2、5 线互不相交，最多航线数为 3；但如果删去的是 2、3，则还要删去第 5 线才符合要求，此时的最多航线数为 2，不是最优解。

于是，我们从上面的分析中再深入，将航线按起点坐标排好序后，如上所述，在本题中，只要线 J 的起点小于线 I 的起点，同时它的终点也小于线 I 的终点，则线 J 和线 I 不相交。因此，求所有线中最多能有多少条线不相交，实际上是从终点坐标值数列中求一个最长不下降序列。这就把题目转化为一个非常典型的动态规划题目了。

求最长不下降序列的规划方程如下：

L(Si)=max{L(Sj)}+1；1< = j < i 且 Sj < Si。 Si 为航线的终点坐标值。

如上数据可以得下解：

编号	南岸	北岸	L 值和前趋
1	1	3	1，0
2	2	4	2，1
3	3	1	1，0
4	4	5	3，2
5	4	2	2，3

非常明显，可以得出解为 3，航线为 4，2，1。

从以上分析过程可以得出：当我们拿到一道题时，不要急于求解，而应先将题目的表面现象一层层象剥竹笋一样去掉，只留下最实质的内容。这时再来设计算法，往往能事半功倍。

例 4、车队过桥(bridge.pas)

〖题目描述〗

GDOI 工作组遇到了一个运输货物的问题。现在有 N 辆车要按顺序通过一个单向的小桥，由于小桥太窄，不能有两辆车并排通过，所以在桥上不能超车。另外，由于小桥建造的时间已经很久，所以小桥只能承受有限的重量，记为 Max（吨）。所以，车辆在过桥的时候必须要有管理员控制，将这 N 辆车按初始顺序分组，每次让一个组过桥，并且只有在一个组中所有的车辆全部过桥以后才让下一组车辆上桥。现在，每辆车的重量和最大速度是已知的，而每组车的过桥时间由该组中速度最慢的那辆车决定。现在请你编一个程序，将这 N 辆车分组，使得全部车辆通过小桥的时间最短。

【输入格式】

数据存放在当前目录下的文本文件“bridge.in”中。
文件的第一行有三个数，分别为 Max（吨），Len（桥的长度，单位：Km），N（三个数之间用一个或多个空格分开）。
接下来有 N 行，每行两个数，第 i 行的两个数分别表示第 i 辆车的重量(吨)和最大速度(m/s)。
注意：所有的输入都为整数，并且任何一辆车的重量都不会超过 Max。

【输出格式】

答案输出到当前目录下的文本文件“bridge.out”中。
文件只有一行，输出全部车辆通过小桥的最短时间（s），精确到小数点后一位。

【输入输出样例】

bridge.in	bridge.out
100 5 9	1050.0
40 25	
50 20	
70 10	
12 50	
9 70	
49 30	
38 25	
27 50	
19 70	

【参考程序】

```
program bridge;
  var
      a:array [0..1000] of real;
      b:array [1..1000,1..2] of longint;
      max,len,n:longint;
      c:longint;
     f1,f2:Text;
```

```
procedure try;
  var
    c,d,e,f,g:longint; flag:boolean;
    time,t,t1:real;
  begin
    a[0]:=0; a[1]:=(len/b[1,2]);
    for c:=2 to n do begin
      flag:=true; d:=c+1; e:=0; f:=maxint; time:=maxint;
      while flag do begin
        d:=d-1;
        if (e+b[d,1])<=max then begin
                                  e:=e+b[d,1]; if b[d,2]<f then f:=b[d,2]; end
                          else flag:=false;
        if flag then begin
          t1:=len/f; t:=t1+a[d-1];
          if t<time then time:=t;
          if d=1 then flag:=false;
        end;
      end;
      a[c]:=time;
   end;
   rewrite (f2); writeln (f2,a[n]*60:1:1);
   close (f2);
 end;

BEGIN
  assign (f1,'bridge.in');
  assign (f2,'bridge.out');
  reset (f1);
  readln (f1,max,len,n);
  for c:=1 to n do
    readln (f1,b[c,1],b[c,2]);
  close (f1);
  try;
END.
```

例 5、复制书稿（book.pas）

【问题描述】

有 M 本书（编号为 1，2，…，M），每本书都有一个页数（分别是 P_1，P_2，…，P_M）。想将每本都复制一份。将这 M 本书分给 K 个抄写员（1<=K<=M<=500），每本书只能分配给一个抄写员进行复制。每个抄写员至少被分配到一本书，而且被分配到的书必须是连续顺序的。

复制工作是同时开始进行的，并且每个抄写员复制一页书的速度都是一样的。所以，复制完所有书稿所需时间取决于分配得到最多工作的那个抄写员的复制时间。试找一个最优分配方案，使分配给每一个抄写员的页数的最大值尽可能小。

【输入格式】

第一行两个整数 M、K；（K<=M<=100）

第二行 M 个整数，第 i 个整数表示第 i 本书的页数。

【输出格式】

共 K 行，每行两个正整数，第 i 行表示第 i 个人抄写的书的起始编号和终止编号。K 行的起始编号应该从小到大排列，如果有多解，则尽可能让前面的人少抄写。

【输入样例】BOOK.IN

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

【输出样例】BOOK.OUT

```
1 5
6 7
8 9
```

【算法分析】

该题中 M 本书是顺序排列的，K 个抄写员选择数也是顺序且连续的。不管以书的编号，还是以抄写员标号作为参变量划分阶段，都符合策略的最优化原理和无后效性。考虑到 K<=M，以抄写员编号来划分阶段会方便些。

设 F（I，J）为前 I 个抄写员复制前 J 本书的最小“页数最大数”。于是便有 F（I，J）=MIN{F（I-1，V），T（V+1，J）}（1<=I<=K，I<=J<=M-K+I，I-1<=V<=J-1）。其中 T（V+1，J）表示从第 V+1 本书到第 J 本书的页数和。边界条件 F（1,i）=T(1,j)。

【参考程序】

```
program book;
const maxm=100;
var p,t:array[1..maxm] of longint;
    f:array[1..maxm,1..maxm] of longint;
    m,k,i,j,v:longint;

function max(a,b:longint):longint;
begin
     if a>b then max:=a else max:=b;
end;

procedure out(i,j:longint);     {递归输出}
```

```
var v:longint;
begin
     if i=1 then
        begin
             writeln(1,' ',j);
             exit;
        end;
     for v:=i-1 to j-1 do
         if max(f[i-1,v],t[j]-t[v])<=f[k,m] then
            begin
                 out(i-1,v);
                 writeln(v+1,' ',j);
                 exit;
            end;
end;

BEGIN  {main}
     assign(input,'book.in');
     assign(output,'book.out');
     reset(input);rewrite(output);
     read(m,k);
     for i:=1 to m do read(p[i]);         {p[i]存放每本书的页数}
     t[1]:=p[1];
     for i:=2 to m do t[i]:=t[i-1]+p[i];  {t[i]存放前 i 本书的总页数}
     for j:=1 to m do f[1,j]:=t[j];       {边界条件、初始状态}
     for i:=2 to k do
         for j:=i to m do                 {记忆化递归}
             begin
                  f[i,j]:=maxlongint;
                  for v:=i-1 to j-1 do
                      if f[i-1,v]>t[j]-t[v]
                         then begin
                                   if f[i-1,v]<f[i,j] then
                                      f[i,j]:=f[i-1,v];
                              end
                         else begin
                                   if t[j]-t[v]<f[i,j] then
                                      f[i,j]:=t[j]-t[v];
                              end;
             end;
     out(k,m);
```

```
    close(input); close(output);
END.
```

例 6、拔河比赛（tug.pas）

【问题描述】

一个学校举行拔河比赛，所有的人被分成了两组，每个人必须（且只能够）在其中的一组，要求两个组的人数相差不能超过 1，且两个组内的所有人体重加起来尽可能地接近。

【输入格式】

输入数据的第 1 行是一个 n，表示参加拔河比赛的总人数，n<=100，接下来的 n 行表示第 1 到第 n 个人的体重，每个人的体重都是整数（1<=weight<=450）。

【输出格式】

输出数据应该包含两个整数：分别是两个组的所有人的体重和，用一个空格隔开。注意如果这两个数不相等，则请把小的放在前面输出。

【输入样例】tug.in

```
3
100
90
200
```

【输出样例】tug.out

```
190 200
```

【算法分析】

这道题目不满足动态规划最优子结构的特性。因为最优子结构要求一个问题的最优解只取决于其子问题的最优解。就这道题目而言，当前 n-1 个人的分组方案达到最优时，并不意味着前 n 个人的分组方案也最优。但题目中标注出每个人的最大体重为 450，这就提醒我们可以从这里做文章，否则的话，命题者大可把最大体重标注到长整型。假设 w[i]表示第 i 个人的体重。c[i,j,k]表示在前 i 个人中选 j 个人在一组，他们的重量之和等于 k 是否可能。显然，c[i,j,k]是 boolean 型，其值为 true 代表有可能，false 代表没有可能。那 c[i,j,k]与什么有关呢？从前 i 个人中选出 j 个人的方案，不外乎两种情况：⑴第 i 个人没有被选中，此时就和从前面 i-1 个人中选出 j 个人的方案没区别，所以 c[i,j,k]与 c[i-1,j,k]有关。⑵第 i 个人被选中，则 c[i,j,k]与 c[i-1,j-1,k-w[i]]有关。综上所述，可以得出：

c[i,j,k]=c[i-1,j,k] or c[i-1,j-1,k-w[i]]。

这道题占用的空间看似达到三维，但因为 i 只与 i-1 有关，所以在具体实现的时候，可以把第一维省略掉。另外在操作的时候，要注意控制 j 与 k 的范围(0<=j<=i/2,0<=k<=j*450)，否则有可能超时。

这种方法的实质是把解本身当作状态的一个参量，把最优解问题转化为判定性问题，用递推的方法求解。这种问题有一个比较明显的特征，就是问题的解被限定在一个较小的范围内，如这题中人的重量不超过 450。

【参考程序】

```
program Tug;
const  maxn=100;
       maxn2=maxn div 2;
       maxrange=450;
var  c:array [0..maxn2,0..maxn2*maxrange] of boolean;
     w:array [1..maxn] of longint;
     n,n2,sum,i,j,k,min,ans:longint;
BEGIN
  assign(input,'tug.in');
  assign(output,'tug.out');
  reset(input);
  rewrite(output);

  read(n); n2:=(n+1) div 2;
  sum:=0;
  for i:=1 to n do
    begin
      read(w[i]);
      inc(sum,w[i]);
    end;

  fillchar(c,sizeof(c),0);
  c[0,0]:=true;
  for i:=1 to n do
      for j:=n2-1 downto 0 do
        for k:=maxrange*i downto 0 do
          if c[j,k] then c[j+1,k+w[i]]:=true;

  min:=maxlongint;
  ans:=0;
  for k:=0 to maxrange*n2 do
      if c[n2,k] and (abs(sum-k-k)<min) then
      begin
        min:=abs(sum-k-k);
        if k<=sum div 2 then ans:=k else ans:=sum-k;
      end;
  write(ans,' ',sum-ans);
  close(output);
  close(input);
END.
```

例 7、投资问题（invest.pas）

【问题描述】

有 n 万元的资金，可投资于 m 个项目，其中 m 和 n 为小于 100 的自然数。对第 i（1≤i≤m）个项目投资 j 万元(1≤j≤n，且 j 为整数)可获得的回报为 Q(i,j)，请你编一个程序，求解并输出最佳的投资方案（即获得回报总值最高的投资方案）。

输入数据放在文件 invest.in 中，格式如下：

m n

Q(1,0) Q(1,1)……Q(1,n)

Q(2,0) Q(2,1)……Q(2,n)

……

Q(m,0) Q(m,1)……Q(m,n)

输出数据放在文件 invest.out 中，格式为：

r(1) r(2) …… r(m) P

其中 r(i)（1≤i≤m）表示对第 i 个项目的投资万元数，P 为总的投资回报值，保留两位有效数字，任意两个数之间空一格。当存在多个并列的最佳投资方案时，只要求输出其中之一即可。

【输入样例】

invest.in：

2 3

0 1.1 1.3 1.9

0 2.1 2.5 2.6

【输出样例】

invest.out：

1 2 3.6

【算法分析】

本题可供考虑的递推角度只有资金和项目两个角度，从项目的角度出发，逐个项目地增加可以看出只要算出了对前 k 个项目投资 j 万元最大投资回报值(1≤j≤n)，就能推出增加第 k+1 个项目后对前 k+1 个项目投资 j 万元最大投资回报值(1≤j≤n)，设 P[k,j]为前 k 个项目投资 j 万元最大投资回报值，则 P[k+1,j]= Max(P[k,i]+Q[k+1,j-i]),0<=i<=j，k=1 时，对第一个项目投资 j 万元最大投资回报值(0≤j≤n)是已知的（边界条件）。

【算法设计】

动态规划的时间复杂度相对于搜索算法是大大降低了，却使空间消耗增加了很多。所以在设计动态规划的时候，需要尽可能节省空间的占用。本题中如果用二维数组存储原始数据和最大投资回报值的话，将造成空间不够，事实上，从前面的问题分析可知：在计算对前 k+1 个项目投资 j 万元最大投资回报值时，只要用到矩阵 Q 的第 k+1 行数据和对前 k 个项目投资 j 万元最大投资回报值，而与前面的数据无关，后者也只需有一个一维数组存储即可，程序中用一维数组 Q 存储当前项目的投资回报值，用一维数组 maxreturn 存储对当前项目之前的所有项目投资 j 万元最大投资回报值(0≤j≤n)，用一维数组 temp 存储对到当前项目为

止的所有项目投资 j 万元最大投资回报值(0≤j≤n)。为了输出投资方案，程序中使用了一个二维数组 invest，invest[k,j]记录了对前 k 个项目投资 j 万元获得最大投资回报时投资在第 k 个项目上的资金数。

【参考程序】

```
program invest(input,output);
const maxm=100; maxn=100;
type arraytype=array [0..maxn] of real;
var i,j,k,m,n,rest:integer;
    q,maxreturn,temp:arraytype;
    invest:array[1..maxm,0..maxn] of integer;
    result:array[1..maxm] of integer;
begin
     assign(input,' invest.in' ); reset(input);
     readln(m,n);
     for j:=0 to n do read(q[j]);
     readln;
     for i:=1 to m do
         for j:=0 to n do invest[i,j]:=0;
     maxreturn:=q;
     for j:=0 to n do invest[1,j]:=j;
     for k:=2 to m do
     begin
          temp:=maxreturn;
          for j:=0 to n do invest[k,j]:=0;
          for j:=0 to n do read(q[j]);
          readln;
          for j:=0 to n do
              for i:=0 to j do
                  if maxreturn[j-i]+q[i]>temp[j] then
                     begin
                          temp[j]:=maxreturn[j-i]+q[i];
                          invest[k,j]:=i
                     end;
          maxreturn:=temp
     end;
     close(input);
     assign(output,' invest.out' );rewrite(output);
     rest:=n;
     for i:=m downto 1 do
     begin
          result[i]:=invest[i,rest];
```

```
            rest:=rest-result[i]
        end;
        for i:=1 to m do write(result[i],' ');
        writeln(maxreturn[n]:0:2);
        close(output)
end.
```

例 8、花店橱窗布置问题(FLOWER.PAS)

【问题描述】

假设你想以最美观的方式布置花店的橱窗。现在你有 F 束不同品种的花束，同时你也有至少同样数量的花瓶被按顺序摆成一行。这些花瓶的位置固定于架子上，并从 1 至 V 顺序编号，V 是花瓶的数目，从左至右排列，则最左边的是花瓶 1，最右边的是花瓶 V。花束可以移动，并且每束花用 1 至 F 间的整数唯一标识。标识花束的整数决定了花束在花瓶中的顺序，如果 I＜J，则令花束 I 必须放在花束 J 左边的花瓶中。

例如，假设一束杜鹃花的标识数为 1，一束秋海棠的标识数为 2，一束康乃馨的标识数为 3，所有的花束在放入花瓶时必须保持其标识数的顺序，即：杜鹃花必须放在秋海棠左边的花瓶中，秋海棠必须放在康乃馨左边的花瓶中。如果花瓶的数目大于花束的数目。则多余的花瓶必须空置，且每个花瓶中只能放一束花。

每一个花瓶都具有各自的特点。因此，当各个花瓶中放入不同的花束时，会产生不同的美学效果，并以美学值（一个整数）来表示，空置花瓶的美学值为零。

在上述例子中，花瓶与花束的不同搭配所具有的美学值，如下表所示。

		花　瓶				
		1	2	3	4	5
花束	1（杜鹃花）	7	23	-5	-24	16
	2（秋海棠）	5	21	-4	10	23
	3（康乃馨）	-21	5	-4	-20	20

例如，根据上表，杜鹃花放在花瓶 2 中，会显得非常好看；但若放在花瓶 4 中则显得十分难看。

为取得最佳美学效果，你必须在保持花束顺序的前提下，使花束的摆放取得最大的美学值。如果有不止一种的摆放方式具有最大的美学值，则其中任何一直摆放方式都可以接受，但你只要输出任意一种摆放方式。

1、假设条件

1≤F≤100，其中 F 为花束的数量，花束编号从 1 至 F。

F≤V≤100，其中 V 是花瓶的数量。

$-50 \leq A_{ij} \leq 50$，其中 A_{ij} 是花束 i 在花瓶 j 中的美学值。

2、输入

输入文件是 flower.in。

第一行包含两个数：F，V。

随后的 F 行中，每行包含 V 个整数，A_{ij} 即为输入文件中第（i+1 ）行中的第 j 个数。

3、输出

输出文件必须是名为 flower.out 的正文文件，文件应包含两行：

第一行是程序所产生摆放方式的美学值。

第二行必须用 F 个数表示摆放方式，即该行的第 K 个数表示花束 K 所在的花瓶的编号。

4、例子

flower.in:

```
3 5
7 23 -5 -24 16
5 21 -4 10 23
-21 5 -4 -20 20
```

flower.out:

```
53
2 4 5
```

【算法分析】

flower 一题是 IOI99 第一天第一题，该题如用组合的方法处理，将会造成超时。正确的方法是用动态规划，考虑角度为一束一束地增加花束，假设用 b(i,j)表示 1～i 束花放在 1 到 j 之间的花瓶中的最大美学值，其中 i<=j ，则 b(i,j)=max(b[i-1,k-1]+A[i,k])，其中 i<=k<=j，A(i,k)的含义参见题目。输出结果时，显然使得 b[F,k]取得总的最大美观值的第一个 k 值就是第 F 束花应该摆放的花瓶位置，将总的最大美观值减去 A[i,k]的值即得到前 k-1 束花放在前 k-1 个瓶中的最大美观值，依次使用同样的方法就可求出每一束花应该摆放的花瓶号。由于这一过程是倒推出来的，所以程序中用递归程序来实现。

【参考程序】

```
program ex8_4;
const max=100;
var f,v,i,j,k,cmax,current,max_val:integer;
    table,val:array[1..max,1..max] of integer;

procedure print(current,max_val:integer);
var i:integer;
begin
     if current>0 then
     begin
```

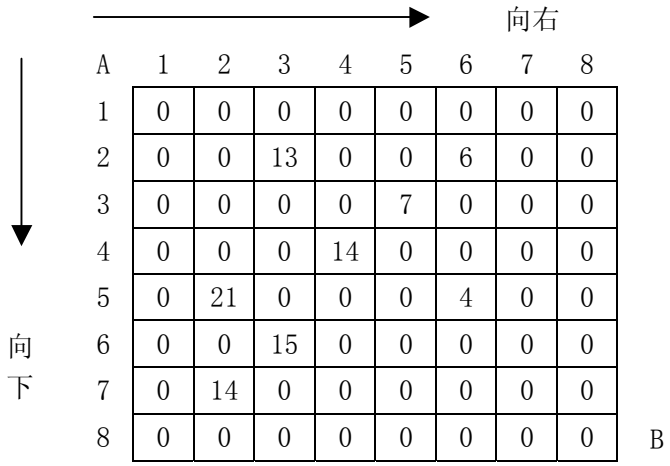
```
          i:=current;
          while val[current,i]<>max_val do i:=i+1;
          print(current-1,max_val-table[current,i]);
          write(i,' ')
     end
end;
begin
     assign(input,'flower.in');
     assign(output,'flower.out');
     reset(input);rewrite(output);
     readln(f,v);
     for i:=1 to f do
     begin
          for j:=1 to v do read(table[i,j]);
          readln;
     end;
     close(input);
     max_val:=-maxint;
     for i:=1 to v do
         if max_val<table[1,i]
            then begin val[1,i]:=table[1,i];max_val:=table[1,i] end
            else val[1,i]:=table[1,i];
     for i:=2 to f do
         for j:=i to v-f+i do
         begin
              max_val:=-maxint;
              for k:=i-1 to j-1 do
              begin
                   cmax:=-maxint;
                   for current:=k+1 to j do
                       if table[i,current]>cmax then cmax:=table[i,current];
                   if cmax+val[i-1,k]>max_val then max_val:=cmax+val[i-1,k]
              end;
              val[i,j]:=max_val
         end;
     max_val:=-maxint;
     for i:=f to v do
         if val[f,i]>max_val then max_val:=val[f,i];
     writeln(max_val);
     print(f,max_val);
     writeln
```

end.

例 9、方格取数（pane.pas）

【问题描述】

设有 N*N 的方格图（N≤8），我们将其中的某些方格中填入正整数，而其他的方格中则放入数字 0。如下图所示（见样例）：



某人从图的左上角的 A 点出发，可以向下行走，也可以向右走，直到到达右下角的 B 点。在走过的路上，他可以取走方格中的数（取走后的方格中将变为数字 0）。

此人从 A 点到 B 点共走两次，试找出 2 条这样的路径，使得取得的数之和为最大。

【输入格式】

输入文件 pane.in，第一行为一个整数 N（表示 N*N 的方格图），接下来的每行有三个整数，前两个表示位置，第三个数为该位置上所放的数。一行单独的 0 表示输入结束。

【输出格式】

输出文件 pane.out，只需输出一个整数，表示 2 条路径上取得的最大的和。

【输入样例】

```
8
2  3  13
2  6  6
3  5  7
4  4  14
5  2  21
5  6  4
6  3  15
7  2  14
0  0  0
```

【输出样例】

67

【算法分析】

本题是从 1997 年国际信息学奥林匹克的障碍物探测器一题简化而来，如果人只能从 A 点到 B 点走一次，则可以用动态规划算法求出从 A 点到 B 点的最优路径。具体的算法描述如下：从 A 点开始，向右和向下递推，依次求出从 A 点出发到达当前位置（i，j）所能取得的最大的数之和，存放在 sum 数组中，原始数据经过转换后用二维数组 data 存储，为方便处理，对数组 sum 和 data 加进零行与零列，并置它们的零行与零列元素为 0。易知

$$sum[i,j]=\begin{cases} data[i,j] & \text{当 } i=0 \text{ 或 } j=0 \\ \max(sum[i-1,j],sum[i,j-1])+data[i,j] & \text{当 } i>0,\text{且 } j>0 \end{cases}$$

求出 sum 数组以后，通过倒推即可求得最优路径，具体算法如下：

```
置 sum 数组零行与零列元素为 0
for i:=1 to n do
    for j:=1 to n do
        if sum[i-1,j]>sum[i,j-1]
          then sum[i,j]:=sum[i-1,j]+data[i,j]
          else sum[i,j]:=sum[i,j-1]+data[i,j];
 i:=n; j:=n;
 while (i>1) or (j>1) do
       if (i>1) and (sum[i,j]=sum[i-1,j]+data[i,j])
          then begin data[i,j]:=-1; i:=i-1 end
          else begin data[i,j]:=-1; j:=j-1 end;
 data[1,1]:=-1;
```

凡是被标上-1 的格子即构成了从 A 点到 B 点的一条最优路径。

那么是否可以按最优路径连续走两次而取得最大数和呢?这是一种很自然的想法,并且对样例而言也是正确的,具体算法如下：

求出数组 sum，

s1:=sum[n,n]，

求出最优路径，

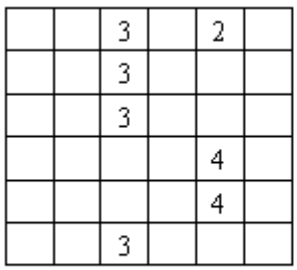
将最优路径上的格子中的值置为 0，

将数组 sum 的所有元素置为 0，

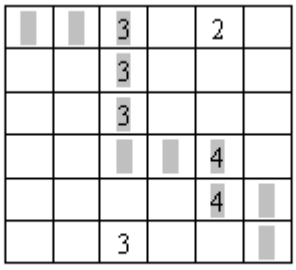
第二次求出数组 sum，

输出 s1+sum[n,n]。

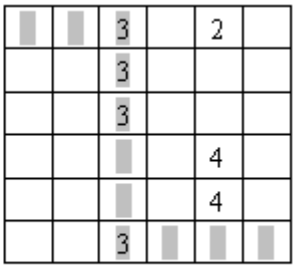
虽然这一算法保证了连续的两次走法都是最优的，但却不能保证总体最优，相应的反例也不难给出，请看下图：



图一



图二



图三

图二按最优路径走一次后，余下的两个数 2 和 3 就不可能同时取倒了，而按图三中的非最优路径走一次后却能取得全部的数，这是因为两次走法之间的协作是十分重要的，而图 2 中的走法并不能考虑全局，因此这一算法只能算是“贪心算法”。虽然在一些情况下，贪心算法也能够产生最优解，但总的来说“贪心算法”是一种有明显缺陷的算法。

既然简单的动态规划行不通，那么看看穷举行不行呢？因为问题的规模比较小，启发我们从穷举的角度出发去思考，首先让我们来看看 N=8 时，从左上角 A 到达右下角 B 的走法共有多少种呢？显然从 A 点到 B 点共需走 14 步，其中向右走 7 步，向下走 7 步，共有 C_{14}^{7}=3432 种不同的路径，走两次的路径组合总数为 C_{3432}^{2}=3432＊3431/2=5887596，从时间上看是完全可以承受的，但是如果简单穷举而不加优化的话，对极限数据还是会超时的，优化的最基本的方法是以空间换时间，具体到本题就是预先将每一条路径以及路径上的数字之和（称之为路径的权 weight）求出并记录下来，然后用双重循环穷举任意两条不同路径之组合即可。

考虑到记录所有的路径需要大量的存储空间，我们可以将所有的格子逐行进行编号，这样原来用二维坐标表示的格子就变成用一个 1 到 n^2 之间的自然数来表示，格子（i,j）对应的编号为（i-1）*n+j。一条路径及其权使用以下的数据结构表示：

```
const maxn=8;
type arraytype=array[1..2*maxn-2] of byte;
    recordtype=record path:arraytype;
                      weight:longint
               end;
```

数组 path 依次记录一条路径上除左上和右下的全部格子的编号。

将所有的路径以及路径的权求出并记录下来的算法可用一个递归的过程实现，其中 i,j 表示当前位置的行与列，step 表示步数，sum 记录当前路径上到当前位置为止的数之和，当前路径记录在数组 position 中（不记录起始格），主程序通过调用 try(1,1,0,data[1,1]) 求得所有路径。

```
procedure try(i,j,step,sum:longint);
begin
     if (i=n) and (j=n)
        then begin
                  total:=total+1;
                  a[total].path:=position;
                  a[total].weight:=sum;
             end
        else begin
                  if i+1<=n then
                  begin
                       position[step]:=i*n+j;
```

```
                    try(i+1,j,step+1,sum+data[i+1,j])
                end;
                if j+1<=n then
                begin
                    position[step]:=(i-1)*n+j+1;
                    try(i,j+1,step+1,sum+data[i,j+1])
                end
            end
end;
```

在穷举了二条不同的路径后，只要将二条路径的权相加再减去二条路径中重叠格子中的数即为从这二条路径连走两次后取得的数之和，具体算法如下：

```
for i:=1 to n do  {将二维数组转化为一维数组}
    for j:=1 to n do d1[(i-1)*n+j]:=data[i,j];
max:=0;
for i:=1 to total-1 do
    for j:=i+1 to total do
    begin
        current:=a[i].weight+a[j].weight;
        for k:=1 to 2*n-3 do {判断重叠格子，但不包括起点的终点}
        begin
            if a[i].path[k]=a[j].path[k]{第 k 步到达同一方格}
               then current:=current-d1[a[i].path[k]]
        end;
        if current>max then max:=current
    end;
writeln(max-data[1,1]-data[n,n]);
```

应该看到穷举的效率是十分低下的，如果问题的规模再大一些，穷举法就会超时，考虑到走一次可以使用动态规划，则只需穷举走第一次的路径，而走第二次可以用动态规划，这样可大大提高程序的效率，其算法复杂度为$o(n^2C_{2n-2}^{n-1})$，实现时只需将前面二种算法结合起来即可，但这样做仍然不能使人满意，因为只要穷举了从 A 点到 B 点所有路径，算法的效率就不可能很高，要想对付尽可能大的 n，还是要依靠动态规划算法。实际上本问题完全可以用动态规划解决，只是递推起来更为复杂些而已，前面在考虑只走一次的情况，只需考虑一个人到达某个格子（i，j）的情况，得出 sum[i，j]=max（sum[i-1，j]，sum[i，j-1]）+data[i，j]，现在考虑两个人同时从 A 出发，则需考虑两个人到达任意两个格子(i1，j1)与(i2，j2)的情况，显然要到达这两个格子，其前一状态必为(i1-1，j1)，(i2-1，j2)；(i1-1，j1)，(i2，j2-1)；(i1，j1-1)，(i2-1，j2)；(i1，j1-1)，(i2，j2-1) 四种情况之一，类似地，可以推导出：

设 p=max(sum[i1-1,j1,i2-1,j2],sum[i1-1,j1,i2,j2-1],sum[i1,j1-1,i2-1,j2],sum[i1,j1-1,i2,j2-1])，则

$$sum[i1,j1,i2,j2]=\begin{cases}0 & \text{当 i1=0 或 j1=0 或 i2=0 或 j2=0}\\ p+data[i1,j1] & \text{当 i1,j1,i2,j2 均不为零，且 i1=i2，j1=j2}\\ p+data[i1,j1]+data[i2,j2] & \text{当 i1,j1,i2,j2 均不为零,且 i1}\neq i2\text{ 或 }j1\neq j2\end{cases}$$

具体算法如下：

```
置 sum 数组所有元素全为 0;
for i1:=1 to n do
    for j1:=1 to n do
        for i2:=1 to n do
            for j2:=1 to n do
            begin
                if sum[i1-1,j1,i2-1,j2]>sum[i1,j1,i2,j2]
                   then sum[i1,j1,i2,j2]:=sum[i1-1,j1,i2-1,j2];
                if sum[i1-1,j1,i2,j2-1]>sum[i1,j1,i2,j2]
                   then sum[i1,j1,i2,j2]:=sum[i1-1,j1,i2,j2-1];
                if sum[i1,j1-1,i2-1,j2]>sum[i1,j1,i2,j2]
                   then sum[i1,j1,i2,j2]:=sum[i1,j1-1,i2-1,j2];
                if sum[i1,j1-1,i2,j2-1]>sum[i1,j1,i2,j2]
                   then sum[i1,j1,i2,j2]:=sum[i1,j1-1,i2,j2-1];
                sum[i1,j1,i2,j2]:=sum[i1,j1,i2,j2]+data[i1,j1];
                if (i1<>i2) or (j1<>j2)
                   then sum[i1,j1,i2,j2]:=sum[i1,j1,i2,j2]+data[i2,j2]
            end;
writeln(sum[n,n,n,n])
```

【参考程序】

```
program pane;
const maxn=10;
type arraytype=array [0..maxn,0..maxn] of longint;
var i,j,k,n,i1,i2,j1,j2:longint;
    data:arraytype;
   sum:array [0..maxn,0..maxn,0..maxn,0..maxn] of longint;

function max(x,y:longint):longint;
begin
     if x>y then max:=x else max:=y;
end;

BEGIN  {main}
     Assign(input,' pane.in' );
     Assign(output,' pane.out' );
     Reset(input);Rewrite(output);
     for i:=1 to maxn do
         for j:=1 to maxn do data[i,j]:=0;
     readln(n);
     repeat
           readln(i,j,k);
```

```
        data[i,j]:=k
    until (i=0) and (j=0) and (k=0);
    fillchar(sum,sizeof(sum),0);
    for i1:=1 to n do
        for j1:=1 to n do
            for i2:=1 to n do
                for j2:=1 to n do
                begin
                    if sum[i1-1,j1,i2-1,j2]>sum[i1,j1,i2,j2]
                       then sum[i1,j1,i2,j2]:=sum[i1-1,j1,i2-1,j2];
                    if sum[i1-1,j1,i2,j2-1]>sum[i1,j1,i2,j2]
                       then sum[i1,j1,i2,j2]:=sum[i1-1,j1,i2,j2-1];
                    if sum[i1,j1-1,i2-1,j2]>sum[i1,j1,i2,j2]
                       then sum[i1,j1,i2,j2]:=sum[i1,j1-1,i2-1,j2];
                    if sum[i1,j1-1,i2,j2-1]>sum[i1,j1,i2,j2]
                       then sum[i1,j1,i2,j2]:=sum[i1,j1-1,i2,j2-1];
                    sum[i1,j1,i2,j2]:=sum[i1,j1,i2,j2]+data[i1,j1];
                    if (i1<>i2) or (j1<>j2)
                       then sum[i1,j1,i2,j2]:=sum[i1,j1,i2,j2]+data[i2,j2]
                end;
    writeln(sum[n,n,n,n]);
    close(input);close(output)
END.
```

【运行示例】

```
pane.in:
3
1 1 10
1 3 5
2 2 6
2 3 4
3 1 8
3 2 2
0 0 0

pane.out:
30
```

【上机练习】

例 1、挖地雷(Mine.pas)
例 2、防卫导弹(Catcher.pas)
例 3、轮船问题(ship.pas)
例 4、车队过桥(bridge.pas)
例 5、复制书稿(book.pas)
例 6、拔河比赛(tug.pas)
例 7、投资问题(invest.pas)
例 8、合并石子(Unite.pas)

【问题描述】

在一个操场上一排地摆放着N堆石子。现要将石子有次序地合并成一堆。规定每次只能选相邻的 2 堆石子合并成新的一堆，并将新的一堆石子数记为该次合并的得分。

【问题求解】

试设计一个程序，计算出将N堆石子合并成一堆的最小得分。

【输入格式】

第一行为一个正整数N (2≤N≤100)；

以下N行,每行一个正整数，小于 10000，分别表示第 i 堆石子的个数(1≤i≤N)。

【输出格式】

为一个正整数，即最小得分。

【输入样例】

```
UNITE.IN
7
13
7
8
16
21
4
18
```

【输出样例】

```
UNITE.OUT
239
```