

# 动态规划经典教程

引言：本人在做过一些题目后对 DP 有些感想，就写了这个总结：

## 第一节 动态规划基本概念

一，动态规划三要素：阶段，状态，决策。

他们的概念到处都是，我就不多说了，我只说说我对他们的理解：

如果把动态规划的求解过程看成一个工厂的生产线，阶段就是生产某个商品的不同的环节，状态就是工件当前的形态，决策就是对工件的操作。显然不同阶段是对产品的一个前面各个状态的小结，有一个一个小结构成了最终的整个生产线。每个状态间又有关联（下一个状态是由上一个状态做了某个决策后产生的）。

下面举个例子：

要生产一批雪糕，在这个过程中要分好多环节：购买牛奶，对牛奶提纯处理，放入工厂加工，加工后的商品要包装，包装后就去销售……，这样每个环节就可以看做是一个阶段；产品在不同的时候有不同的状态，刚开始时只是白白的牛奶，进入生产后做成了各种造型，从冷冻库拿出来后就变成雪糕（由液态变成固态）。每个形态就是一个状态，那从液态变成固态经过了冰冻这一操作，这个操作就是一个决策。

一个状态经过一个决策变成了另外一个状态，这个过程就是状态转移，用来描述状态转移的方程就是**状态转移方程**。

经过这个例子相信大家动态规划有所了解了吧。

下面再说说我对动态规划的另外一个理解：

用图论知识理解动态规划：把动态规划中的状态抽象成一个点，在有直接关联的状态间连一条有向边，状态转移的代价就是边上的权。这样就形成了一个**有向无环图 AOE 网**（为什么无环呢？往下看）。对这个图进行拓扑排序，删除一个边后同时出现入度为 0 的状态在同一阶段。这样对图求最优路径就是动态规划问题的求解。

二，动态规划的适用范围

动态规划用于解决多阶段决策最优化问题，但是不是所有的最优化问题都可以用动态规划解答呢？

一般在题目中出现**求最优解的问题**就要考虑动态规划了，但是否可以用还要满足两个条件：

**最优子结构**（最优化原理）

**无后效性**

最优化原理在下面的最短路径问题中有详细的解答；

什么是无后效性呢？

就是说在状态 i 求解时用到状态 j 而状态 j 求解又用到状态 k……状态 N。

而求状态 N 时又用到了状态 i 这样求解状态的过程形成了环就没法用动态规划解答了，这也是上面用图论理解动态规划中形成的图无环的原因。

也就是说当前状态是前面状态的完美总结，现在与过去无关。。。

当然，要是换一个划分状态或阶段的方法就满足无后效性了，这样的问题仍然可以用动态规划解。

三，动态规划解决问题的一般思路。

拿到**多阶段决策最优化问题**后，第一步要判断这个问题是否可以用动态规划解决，如果不能就要考虑搜索或贪心了。当确定问题可以用动态规划后，就要用下面介绍的方法解决问题了：

（1）模型匹配法：

最先考虑的就是这个方法了。挖掘问题的本质，如果发现问题是自己熟悉的某个基本的模型，就直接套用，但要小心其中的一些小的变动，现在考题办都是基本模型的变形套用时要小心条件，三思而后行。这些基本模型在后面的分类中将一一介绍。

（2）三要素法

仔细分析问题尝试着确定动态规划的三要素，不同问题的确定方向不同：

先确定**阶段**的问题：数塔问题，和走路问题（详见解题报告）

先确定**状态**的问题：大多数都是先确定状态的。

先确定**决策**的问题：背包问题。（详见解题报告）

一般都是先从比较明显的地方入手，至于怎么知道哪个明显就是经验问题了，多做题就会发现。

（3）寻找规律法：

这个方法很简单，耐心推几组数据后，看他们的规律，总结规律间的共性，有点贪心的意思。

（4）边界条件法

找到问题的边界条件，然后考虑边界条件与它的领接状态之间的关系。这个方法也很起效。

（5）放宽约束和增加约束

这个思想是在陈启锋的论文里看到的，具体内容就是给问题增加一些条件或删除一些条件使问题变的清晰。

## 第二节 动态规划分类讨论

这里用状态维数对动态规划进行了分类：

### 1. 状态是一维的

#### 1. 1 下降/非降子序列问题：

问题描述： {挖掘题目的本质，一但抽象成这样的描述就可以用这个方法解}

在一个无序的序列  $a_1, a_2, a_3, a_4 \dots a_n$  里，找到一个最长的序列满足： $a_i \leq a_j \leq a_k \dots \leq a_m$ ，且  $i < j < k \dots < m$ 。（最长非降子序列）或  $a_i > a_j > a_k \dots > a_m$ ，且  $i > j > k \dots > m$ 。（最长下降子序列）。

问题分析：

如果前  $i-1$  个数中用到  $a_k$  ( $a_k > a_i$  或  $a_k \leq a_i$ ) 构成了一个最长的序列加上第  $i$  个数  $a_i$  就是前  $i$  个数中用到  $i$  的最长的序列了。那么求用到  $a_k$  构成的最长的序列有要求前  $k-1$  个数中……

从上面的分析可以看出这样划分问题满足最优子结构，那满足无后效性么？显然对于第  $i$  个数时只考虑前  $i-1$  个数，显然满足无后效性，可以用动态规划解。

分析到这里动态规划的三要素就不难得出了：如果按照序列编号划分阶段，设计一个状态  $opt[i]$  表示前  $i$  个数中用到第  $i$  个数所构成的最优解。那么决策就是在前  $i-1$  个状态中找到最大的  $opt[j]$  使得  $a_j > a_i$  (或  $a_j \leq a_i$ )， $opt[j]+1$  就是  $opt[i]$  的值；用方程表示为：{我习惯了这种写法，但不是状态转移方程的标准写法}

$opt[i] = \max(opt[j]) + 1 \quad (0 \leq j < i \text{ 且 } a_j < a_i) \quad \{\text{最长非降子序列}\}$

$opt[i] = \max(opt[j]) + 1 \quad (0 \leq j < i \text{ 且 } a_j > a_i) \quad \{\text{最长下降子序列}\}$

实现求解的部分代码：

```
opt[0] := maxsize; {maxsize 为 maxlongint 或 -maxlongint}
```

```
for i:=1 to n do
```

```
for j:=0 to i-1 do
```

```
if (a[j]>a[i]) and (opt[j]+1>opt[i]) then
```

```
opt[i]:=opt[j]+1;
```

```
ans:=-maxlongint;
```

```
for i:=1 to n do
```

```
if opt[i]>ans then ans:=opt[i]; {ans 为最终解}
```

复杂度：从上面的实现不难看出时间复杂度为  $O(N^2)$ ，空间复杂度  $O(N)$ ；

#### 例题 1 拦截导弹 (missile.pas/c/cpp) 来源：NOIP1999(提高组) 第一题

【问题描述】

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。

【输入文件】missile.in

单独一行列出导弹依次飞来的高度。

【输出文件】missile.out

两行，分别是最多能拦截的导弹数，要拦截所有导弹最少要配备的系统数

【输入样例】

389 207 155 300 299 170 158 65

【输出样例】

6

2

【提交链接】

[http://www.rqnoj.cn/Problem\\_217.html](http://www.rqnoj.cn/Problem_217.html)

【问题分析】

有经验的选手不难看出这是一个求最长非升子序列问题，显然标准算法是动态规划。

以导弹依次飞来的顺序为阶段，设计状态  $opt[i]$  表示前  $i$  个导弹中拦截了导弹  $i$  可以拦截最多能拦截到的导弹的个数。

状态转移方程：

$opt[i] = \max(opt[j]) + 1 \quad (h[i] \geq h[j], 0 \leq j < i) \quad \{h[i] \text{ 存, 第 } i \text{ 个导弹的高度}\}$

最大的  $opt[i]$  就是最终的解。

这只解决了第一问，对于第二问最直观的方法就是求完一次  $opt[i]$  后把刚才要打的导弹去掉，再求一次  $opt[i]$  直到打完所有的导弹，但这样做就错了。

不难举出反例：6 1 7 3 2

错解：6 3 2/1/7 正解：6 1/7 3 2

其实认真分析一下题就会发现：每一个导弹最终的结果都是要被打的，如果它后面有一个比它高的导弹，那打它的这个装置无论如何也不能打那个导弹了，经过这么一分析，这个问题便抽象成在已知序列里找最长上升子序列的问题。

求最长上升序列和上面说的求最长非升序列是一样的，这里就不多说了。

复杂度：时间复杂度为  $O(N^2)$ ，空间复杂度为  $O(N)$ 。

【源代码】

<pre>program missile; const   fin='missile.in';   fout='missile.out';   maxn=10000; var   a,opt:array[0..maxn] of     longint;   n,anslen,ansime:longint; procedure init; var   x:longint; begin   assign(input,fin);   reset(input);   assign(output,fout);   rewrite(output);   n:=0;   repeat     inc(n);   until seekeof; end; procedure main; var   i,j:longint; begin   fillchar(opt,sizeof(opt),0);   a[0]:=maxlongint;   for i:=1 to n do     for j:=i-1 downto 0 do       if (a[j]&gt;=a[i]) and         (opt[j]+1&gt;opt[i]) then         opt[i]:=opt[j]+1;   anslen:=0;   for i:=1 to n do     if opt[i]&gt;anslen then       anslen:=opt[i];   fillchar(opt,sizeof(opt),0);   a[0]:=-maxlongint;   for i:=1 to n do     for j:=i-1 downto 0 do</pre>	<pre>if (a[j]&lt;a[i]) and   (opt[j]+1&gt;opt[i]) then   opt[i]:=opt[j]+1;   ansime:=0;   for i:=1 to n do     if opt[i]&gt;ansime then       ansime:=opt[i];   end;   procedure print;   begin     writeln(anslen);     writeln(ansime);     close(input);     close(output);   end;   begin     init;     main;     print;   end.</pre>
--	---

**例题 2 合唱队形** (chorus.pas/c/cpp) 来源：NOIP2004(提高组) 第一题

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 T1, T2, ..., TK，则他们的身高满足  $T_1 < \dots < T_i < T_{i+1} < \dots < T_K$  ( $1 \leq i \leq K$ )。

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 ( $2 \leq N \leq 100$ )，表示同学的总数。第二行有 n 个整数，用空格分隔，第 i 个整数  $T_i$  ( $130 \leq T_i \leq 230$ ) 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  $n \leq 20$ ；

对于全部的数据，保证有  $n \leq 100$ 。

【提交链接】

[http://www.rqnoj.cn/Problem\\_26.html](http://www.rqnoj.cn/Problem_26.html)

【问题分析】

出列人数最少，也就是说留的人最多，也就是序列最长。

这样分析就是典型的最长下降子序列问题。只要枚举每一个人站中间时可以得到的最优解。显然它就等于，包括他在内向左求最长上升子序列，向右求最长下降子序列。

我们看一下复杂度：

计算最长下降子序列的复杂度是  $O(N^2)$ ，一共求 N 次，总复杂度是  $O(N^3)$ 。这样的复杂度对于这个题的数据范围来说是可以 AC 的。但有没有更好的方法呢？

其实最长子序列只要一次就可以了。因为最长下降（上升）子序列不受中间人的影响。

只要用 OPT1 求一次最长上升子序列，OPT2 求一次最长下降子序列。这样答案就是  $N - \max(opt1[i] + opt2[i] - 1)$ 。

复杂度由  $O(N^3)$  降到了  $O(N^2)$ 。

【源代码】

```
program chorus;
const
  fin='chorus.in';
  fout='chorus.out';
  maxn=200;
var
  opt1,opt2,a:array[0..maxn] of
longint;
  n,ans:longint;
procedure init;
var
  i:longint;
begin
  assign(input,fin);
  reset(input);
  assign(output,fout);
  rewrite(output);
  readln(n);
```

```
  for i:=1 to n do
    read(a[i]);
  end;
  procedure main;
  var
    i,j:longint;
  begin
    a[0]:=-maxlongint;
    for i:=1 to n do
      for j:=i-1 downto 0 do
        if (a[j]<a[i]) and
          (opt1[j]+1>opt1[i]) then
          opt1[i]:=opt1[j]+1;
    a[n+1]:=-maxlongint;
    for i:=n downto 1 do
      for j:=i+1 to n+1 do
        if (a[j]<a[i]) and
          (opt2[j]+1>opt2[i]) then
          opt2[i]:=opt2[j]+1;
```

```
  ans:=0;
  for i:=1 to n do
    if opt1[i]+opt2[i]>ans then
      ans:=opt1[i]+opt2[i];
  end;
  procedure print;
  begin
    writeln(n-ans+1);
    close(input);
    close(output);
  end;
  begin
    init;
    main;
    print;
  end.
```

**例题 3 Buy Low Buy Lower** (buylow.pas/c/cpp) 来源: USACO 4-3-1

【问题描述】

“逢低吸纳”是炒股的一条成功秘诀。如果你想成为一个成功的投资者，就要遵守这条秘诀：

“逢低吸纳，越低越买”

这句话的意思是：每次你购买股票时的股价一定要比你上次购买时的股价低。按照这个规则购买股票的次数越多越好，看看你最多能按这个规则买几次。

给定连续的  $N$  天中每天的股价。你可以在任何一天购买一次股票，但是购买时的股价一定要比你上次购买时的股价低。写一个程序，求出最多能买几次股票。

以下面这个表为例，某几天的股价是：

天数 1 2 3 4 5 6 7 8 9 10 11 12

股价 68 69 54 64 68 64 70 67 78 62 98 87

这个例子中，聪明的投资者(按上面的定义)，如果每次买股票时的股价都比上一次买时低，那么他最多能买 4 次股票。一种买法如下(可能有其他的买法)：

天数 2 5 6 10

股价 69 68 64 62

【输入文件】buylow.in

第 1 行： $N$  ( $1 \leq N \leq 5000$ )，表示能买股票的天数。

第 2 行以下： $N$  个正整数(可能分多行)，第  $i$  个正整数表示第  $i$  天的股价。这些正整数大小不会超过  $\text{longint}(\text{pascal})/\text{long}(\text{c++})$ 。

【输出文件】buylow.out

只有一行，输出两个整数：

能够买进股票的天数，长度达到这个值的股票购买方案数量

在计算解的数量时，如果两个解所组成的字符串相同，那么这样的两个解被认为是相同的(只能算做一个解)。因此，两个不同的购买方案可能产生同一个字符串，这样只能计算一次。

【输入样例】

12

68 69 54 64 68 64 70 67

78 62 98 87

【输出样例】

4 2

【提交链接】

[http://www.rqnoj.cn/Problem\\_456.html](http://www.rqnoj.cn/Problem_456.html)

【问题分析】

从题目描述就可以看出这是最长下降子序列问题，于是求解第一问不是难事，以天数为阶段，设计状态  $\text{opt}[i]$  表示前  $i$  天中要买第  $i$  天的股票所能得到的最大购买次数。

状态转移方程：

$\text{opt}[i] = \max(\text{opt}[j]) + 1$  ( $a[i] > a[j]$ ,  $0 \leq j < i$ ) { $a[i]$  存第  $i$  天股价}

最大的  $\text{opt}[i]$  就是最终的解。

第二问呢，稍麻烦一点。

从问题的边界出发考虑问题，当第一问求得一个最优解  $opt[i]=X$  时，  
在 1 到 N 中如果还有另外一个  $opt[j]=x$  那么选取 J 的这个方案肯定是成立的。  
是不是统计所有的  $opt[j]=x$  的个数就是解了呢？

显然没那么简单，因为选取 J 这天的股票构成的方案不见得  $= 1$ ，看下面一个例子：

5 6 4 3 1 2

方案一：5 4 3 1

方案二：5 4 3 2

方案三：6 4 3 1

方案四：6 4 3 2

$x=4$

也就是所虽然  $opt[5]=X$  和  $opt[6]=X$  但个数只有两个，而实际上应该有四个方案，但在仔细观察发现，  
构成  $opt[5]=x$  的这个方案中  $opt[j]=x-1$  的方案数有两个， $opt[j]=x-2$  的有一个， $opt[j]=x-3$  的有一个……

显然这是满足低归定义的设计函数  $F(i)$  表示前 I 张中用到第 i 张股票的方案数。

递推式：

1 ( $i=0$ )

$F(i)=$

$\text{Sum}(F(j))$  ( $0 \leq j \leq n, a[j] > a[i], opt[j] = opt[i]-1$ ) {sum 代表求和}

答案  $= \text{sum}(F(j))$  ( $0 \leq j \leq n, opt[j] = x$ )

复杂度：

求解第一问时间复杂度是  $O(N^2)$ ，求解第二问如果用递推或递归+记忆化时间复杂度仍为  $O(N^2)$  但要是用赤裸裸的递归那就复杂多了……，因为那样造成了好多不必要的计算。

你认为这样做就解决了这道题目了么？还没有，还要注意一些东西：

(1) 如果有两个方案中出现序列一样，视为一个方案，要需要加一个数组 next 用  $next[i]$  记录和第 i 个数情况一样（即： $opt[i]=opt[j]$  且  $a[i]=a[j]$ ）可看做一个方案的最近的位置。

递推时 j 只要走到  $next[i]$  即可。

(2) 为了方便操作可以将  $a[n+1]$  赋值为  $-\text{maxlongint}$  这样可以认为第  $n+1$  个一定可以买，答案就是  $\text{sum}(F(n+1))$ 。

(3) 看数据规模显然要用高精度。

注：USACO 上最后一个点错了。我在程序中做了处理。

【源代码】

```
program buylow;
const
  fin='buylow.in';
  fout='buylow.out';
  maxn=5010;
  maxsize=10;
  jz=100000000;
type
  arrtype=array[0..maxsize] of
    longint;
  var
    a,opt,next:array[0..maxn] of
    longint;
  F:array[0..maxn] of arrtype;
  n:longint;
  procedure init;
  var
    i:longint;
  begin
    assign(input,fin);
    reset(input);
    assign(output,fout);
    rewrite(output);
    readln(n);
    if n=5 then {最后
      一个点错了，我只好这么写了}
    begin
      writeln('2 5');
      close(input);
      close(output);
```

```
halt;
end;
for i:=1 to n do
  read(a[i]);
end;
procedure Hinc(var
  x:arrtype;y:arrtype);
var
  i,z:longint;
begin
  z:=0;
  for i:=1 to maxsize do
  begin
    z:=z div jz+x[i]+y[i];
    x[i]:=z mod jz;
  end;
end;
procedure main;
var
  i,j:longint;
begin
  a[0]:=maxlongint;
  a[n+1]:=-maxlongint;
  for i:=1 to n+1 do
  for j:=0 to i-1 do
  if (a[j]>a[i]) and
    (opt[j]+1>opt[i]) then
    opt[i]:=opt[j]+1;
  for i:=1 to n do
  begin
    j:=i-1;
```

```
while (j>0) and
  ((opt[i]<>opt[j]) or (a[i]<>a[j]))
do
  dec(j);
  next[i]:=j;
end;
F[0,1]:=1;
for i:=1 to n+1 do
  for j:=i-1 downto next[i] do
  if (opt[j]=opt[i]-1) and
    (a[j]>a[i]) then
    Hinc(F[i],F[j]);
  end;
  procedure print;
  var
    i,top,m:longint;
  begin
    write(opt[n+1]-1,' ');
    top:=maxsize;
    while (top>1) and
      (F[n+1][top]=0) do
      dec(top);
      write(F[n+1,top]);
      for i:=top-1 downto 1 do
      begin
        m:=F[n+1,i];
        while m<maxsize div 10 do
        begin
          write('0');
          m:=m*10;
        end;
```



```
write(F[n+1,i]);
end;
writeln;
close(input);
```

```
close(output);
end;
begin
init;
```

```
main;
print;
end.
```

#### 例题 4 船 (ships.pas/c/cpp) 来源:《奥赛经典》(提高篇)

##### 【问题描述】

PALMIA 国家被一条河流分成南北两岸,南北两岸上各有  $N$  个村庄。北岸的每一个村庄有一个唯一的朋友在南岸,且他们的朋友村庄彼此不同。

每一对朋友村庄想要一条船来连接他们,他们向政府提出申请以获得批准。由于河面上常常有雾,政府决定禁止船只航线相交(如果相交,则很可能导致碰船)。

你的任务是编写一个程序,帮助政府官员决定批准哪些船只航线,使得不相交的航线数目最大。

##### 【输入文件】ships.in

输入文件由几组数据组成。每组数据的第一行有 2 个整数  $X, Y$ , 中间有一个空格隔开,  $X$  代表 PALMIA 河的长度 ( $10 \leq X \leq 6000$ ),  $Y$  代表河的宽度 ( $10 \leq Y \leq 100$ )。第二行包含整数  $N$ , 表示分别坐落在南北两岸上的村庄的数目 ( $1 \leq N \leq 5000$ )。在接下来的  $N$  行中, 每一行有两个非负整数  $C, D$ , 由一个空格隔开, 分别表示这一对朋友村庄沿河岸与 PALMIA 河最西边界的距离 ( $C$  代表北岸的村庄,  $D$  代表南岸的村庄), 不存在同岸又同位置的村庄。最后一组数据的下面仅有一行, 是两个 0, 也被一空格隔开。

##### 【输出文件】ships.out

对输入文件的每一组数据, 输出文件应在连续的行中表示出最大可能满足上述条件的航线的数目。

##### 【输入样例】

```
30 4
7
22 4
2 6
10 3
15 12
9 8
17 17
4 2
0 0
```

##### 【输出样例】

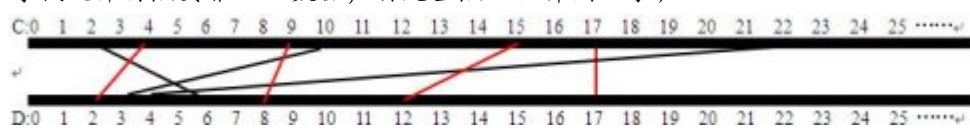
```
4
```

##### 【问题分析】

这道题目相对来说思考难度较高, 从字面意义上看不出问题的本质来, 有点无法下手的感觉。这里我给大家推荐两种思考这类问题的方法。

**方法一:** 寻找规律法 (我前面总结提到的第 3 个方法)

寻找规律自然要推几组数据, 首先当然又从样例入手;



仔细观察上图: 红色航线是合法的, 那么他们满足什么规律呢?

C1 C2 C3 C4

北岸红线的端点: 4 9 15 17

南岸红线的端点: 2 8 12 17

D1 D2 D3 D4

不难看出无论线的斜率如何, 都有这样的规律:

$C1 < C2 < C3 < C4$  且  $D1 < D2 < D3 < D4$

如果我们把输入数据排升序, 问题变抽象为:

在一个序列 (D) 里找到最长的序列满足  $D_i < D_j < D_k \dots$  且  $i < j < k$

这样的话便是典型的最长非降子序列问题了。。。

**方法二:** 边界条件法 (我前面总结提到的第 4 个方法)

边界法其实就是把数据往小了缩, 显然  $N=1$  时答案是 1。  $N=2$  时呢?

考虑这样一组数据:

$N=2$

C1 D1

C2 D2

当  $C1 < C2$  时, 如果  $D1 > D2$  那么一定会相交, 反之则不会相交。

当  $C1 > C2$  时, 如果  $D1 < D2$  那么一定会相交, 反之则不会相交。

$N=3$

C1 D1

C2 D2

C3 D3

.....

其实不用再推导  $N=3$  了, 有兴趣的可以去推导。看  $N=2$  时就能得出:

对于任意两条航线如果满足  $C_i < C_j$  且  $D_i < D_j$  则两条航线不相交。这样的话要想在一个序列里让所有的航线都不相交那当然满足,  $C1 < C2 < C3 \cdots Cans$  且  $D1 < D2 < D3 \cdots Dans$ , 也就是将  $C$  排序后求出最长的满足这个条件的序列的长度就是解。

这样分析后显然是一个**最长非降子序列**问题。

复杂度: 排序可以用  $O(n \log n)$  的算法, 求最长非降子序列时间复杂度是  $O(n^2)$ 。

总复杂度为  $O(n^2)$ 。

#### 【源代码】

```
program ships;
const
  fin='ships.in';
  fout='ships.out';
  maxn=5010;
type
  retype=record
    C,D:longint;
  end;
var
  x,y,n,ans:longint;
  a:array[0..maxn] of retype;
  opt:array[0..maxn] of longint;
procedure init;
var
  i:longint;
begin
  readln(n);
  for i:=1 to n do
    read(a[i].c,a[i].d);
  end;
procedure quick(L,r:longint);
var
  i,j,x:longint;
  y:retype;
begin
```

```
  i:=L;
  j:=r;
  x:=a[(i+j) div 2].c;
  repeat
    while a[i].c<x do
      inc(i);
    while a[j].c>x do
      dec(j);
    if i<=j then
      begin
        y:=a[i];
        a[i]:=a[j];
        a[j]:=y;
        inc(i);
        dec(j);
      end;
  until i>j;
  if j>L then quick(L,j);
  if i<r then quick(i,r);
end;
procedure main;
var
  i,j:longint;
begin
  fillchar(opt,sizeof(opt),0);
  quick(1,n);
  for i:=1 to n do
```

```
    for j:=0 to i-1 do
      if (a[j].d<a[i].d) and
        (opt[j]+1>opt[i]) then
        opt[i]:=opt[j]+1;
    ans:=maxlongint;
    for i:=1 to n do
      if ans<opt[i] then
        ans:=opt[i];
    writeln(ans);
  end;
begin
  assign(input,fin);
  reset(input);
  assign(output,fout);
  rewrite(output);
  read(x,y);
  while (x+y<>0) do
    begin
      init;
      main;
      read(x,y);
    end;
  close(input);
  close(output);
end.
```

## 1.2 背包问题

首先说说**背包问题的基本模型**:

现有  $N$  个物品, 每个物品的价值为  $V$ , 重量为  $W$ 。求用一个载重量为  $S$  的背包装这些物品, 使得所装物品的总价值最高。

背包问题用贪心和搜索解, 当然效果不佳, 不过在我的贪心和搜索总结中会说到。显然标准的解法是动态规划, 我在解决这个问题时习惯设计一维状态, 还可以设计二维的, 二维状态在下面会讲, 现在只讨论用一维状态实现背包问题。

背包问题的分类:

(1) 小数背包: 物品的重量是实数, 如油, 水等可以取 1.67 升.....

(2) 整数背包:  $<1>0/1$  背包: 每个物品只能选一次, 或不选。不能只选一部分

$<2>$  部分背包: 所选物品可以是一部分。

(3) 多重背包: 背包不只一个。

小数背包: 在贪心总结中会细讲。

整数背包:

部分背包: 同小数背包。

$0/1$  背包: 这个问题是最经常出现的问题, 应该熟练掌握。

我们先看一下 **0/1 背包的简化版**：

现有  $N$  个物品，每个物品重量为  $W$ ，这些物品能否使在载重量为  $S$  的背包装满（即重量和正好为  $S$ ）？如果不能那能使物品重量和最重达到多少？

针对这一问题我们以物品的个数分阶段，设计一个状态  $opt[i]$  表示载重量为  $i$  的背包可否装满，显然  $opt[i]$  的基类型是 `boolean`。

决策是什么呢？

当要装第  $i$  个物品时，如果前  $i-1$  个物品可以使载重为  $k$  的背包装满，那么载重为  $k+w[i]$  的背包就可以装满。于是对于一个  $opt[j]$  来说，只要  $opt[j-w[i]]$  是 `true`（表示可装满）那  $opt[j]$  就可以装满，但要注意：针对每一个物品枚举背包的载重量时如果这样正向的推导会使同一个物品用好几次，因为  $k+w[i]$  可装满那  $k+w[i]+w[i]$  就可装满，但实际上是装不满的因为物品只有一个。解决这个问题很简单，只要逆向推导就 OK 了。

这样划分阶段，设计状态，满足无后效性么？

显然对于物品只有一个每一个阶段都是独立的，物品的顺序并不影响求解，因为装物品的次序不限。而对于  $opt[j]$  只考虑  $opt[j-w[i]]$  而不考虑后面的，所以满足无后效性。

有了上面的分析不难写出状态转移方程：

$opt[j] := opt[j-w[i]] \quad \{opt[j-w[i]] = true\}$

时间复杂度：

阶段数  $O(S)$  \* 状态数  $(O(N))$  \* 转移代价  $(O(1)) = O(SN)$

下面看几个例题：

### 例题 5 装箱问题 (pack.pas/c/cpp) 来源：NOIP2001(普及组) 第四题

#### 【问题描述】

有一个箱子容量为  $V$ （正整数， $0 \leq V \leq 20000$ ），同时有  $n$  个物品（ $0 \leq n \leq 30$ ），每个物品有一个体积（正整数）。

要求  $n$  个物品中，任取若干个装入箱内，使箱子的剩余空间为最小。

#### 【输入文件】

第一行，一个正整数  $V$  表示箱子的容量，第二行，一个正整数  $N$  表示物品个数，接下来  $N$  行列出这  $N$  个物品各自的体积。

#### 【输出文件】

单独一行，表示箱子最小的剩余空间。

#### 【输入样例】

```
24
6
8
3
12
7
9
7
```

#### 【输出样例】

```
0
```

#### 【提交链接】

[http://www.rqnoj.cn/Problem\\_147.html](http://www.rqnoj.cn/Problem_147.html)

#### 【问题分析】

本题是经典的 0/1 背包问题，并且是 0/1 背包的简化版，把箱子看做背包，容量看做载重量，体积看做重量，剩余空间最小也就是尽量装满背包。于是这个问题便成了：

有一个载重量为  $V$  的背包，有  $N$  个物品，尽量多装物品，使背包尽量重。

设计一个状态  $opt[i]$  表示重量  $i$  可否构成。

状态转移方程： $opt[j] := opt[j-w[i]] \quad \{opt[j-w[i]] = true\}$

最终的解就是  $v-x$  ( $x \leq n$  且  $opt[x] = true$  且  $opt[x+1..n] = false$ )。

#### 【源代码 1】

<pre>program pack; const   fin='pack.in';   fout='pack.out';   maxv=20010;   maxn=100; var   opt:array[0..maxv] of boolean;</pre>	<pre>w:array[0..maxn] of longint; v,n,x:longint; procedure init; var   i:longint; begin   assign(input,fin);   reset(input);   assign(output,fout);   rewrite(output);</pre>	<pre>read(v); read(n); for i:=1 to n do   read(w[i]); end; procedure main; var   i,j:longint; begin   fillchar(opt,sizeof(opt),false);</pre>
---	--	--



<pre> opt[0]:=true; for i:=1 to n do for j:=v downto w[i] do if      opt[j-w[i]]      then opt[j] :=true; x:=v; while not opt[x] do  dec(x); </pre>	<pre> end; procedure print; begin writeln(v-x); close(input); close(output); end; </pre>	<pre> begin init; main; print; end. </pre>
---	--	--

#### 例题 6 砝码称重 来源: NOIP1996 (提高组) 第四题

##### 【问题描述】

设有 1g、2g、3g、5g、10g、20g 的砝码各若干枚（其总重 $\leq 1000$ ），用他们能称出的重量的种类数。

##### 【输入文件】

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，中间有空格）。

##### 【输出文件】

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

##### 【输出样例】

TOTAL=3

##### 【问题分析】

把问题稍做一个改动，已知  $a_1+a_2+a_3+a_4+a_5+a_6$  个砝码的重量  $w[i]$ ,  $w[i] \in \{1,2,3,5,10,20\}$  其中砝码重量可以相等，求用这些砝码可称出的不同重量的个数。

这样一改就是经典的 0/1 背包问题的简化版了，求解方法完全和上面说的一样，这里就不多说了，只是要注意这个题目不是求最大载重量，是统计所有的可称出的重量的个数。

##### 【源代码 1】

<pre> program P4; const maxn=1010; w:array[1..6]      of longint=(1,2,3,5,10,20); var opt:array[0..maxn] of boolean; a:array[1..6] of longint; procedure init; var i:longint; begin for i:=1 to 6 do </pre>	<pre> read(a[i]); end; procedure main; var i,j,k:longint; begin fillchar(opt,sizeof(opt),false); opt[0]:=true; for i:=1 to 6 do for j:=1 to a[i] do for k:=maxn downto w[i] do if      opt[k-w[i]] then  opt[k]:=true; end; procedure print; </pre>	<pre> var ans,i:longint; begin ans:=0; for i:=1 to maxn do if opt[i] then inc(ans); writeln(ans); end; begin init; main; print; end. </pre>
---	---	---

#### 例题 7 积木城堡 来源: vijos P1059

##### 【问题描述】

XC 的儿子小 XC 最喜欢玩的游戏用积木垒漂亮的城堡。城堡是用一些立方体的积木垒成的，城堡的每一层是一块积木。小 XC 是一个比他爸爸 XC 还聪明的孩子，他发现垒城堡的时候，如果下面的积木比上面的积木大，那么城堡便不容易倒。所以他在垒城堡的时候总是遵循这样的规则。

小 XC 想把自己垒的城堡送给幼儿园里漂亮的女孩子们，这样可以增加他的好感度。为了公平起见，他决定送给每个女孩子一样高的城堡，这样可以避免女孩子们为了获得更漂亮的城堡而引起争执。可是他发现自己在垒城堡的时候并没有预先考虑到这一点。所以他现在要改造城堡。由于他没有多余的积木了，他灵机一动，想出了一个巧妙的改造方案。他决定从每一个城堡中挪去一些积木，使得最终每座城堡都一样高。为了使他的城堡更雄伟，他觉得应该使最后的城堡都尽可能的高。

##### 任务：

请你帮助小 XC 编一个程序，根据他垒的所有城堡的信息，决定应该移去哪些积木才能获得最佳的效果。

##### 【输入文件】

第一行是一个整数  $N(N \leq 100)$ ，表示一共有几座城堡。以下  $N$  行，每行是一系列非负整数，用一个空格分隔，按从下往上的顺序依次给出一座城堡中所有积木的棱长。用 -1 结束。一座城堡中的积木不超过 100 块，每块积木的棱长不超过 100。

### 【输出文件】

一个整数，表示最后城堡的最大可能的高度。如果找不到合适的方案，则输出 0。

### 【输入样例】

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

### 【输出样例】

```
3
```

### 【提交链接】

[http://www.vijos.cn/Problem\\_Show.asp?id=1059](http://www.vijos.cn/Problem_Show.asp?id=1059)

### 【问题分析】

首先要说明一点，可以挪走任意一个积木，不见得是最上面的。

初看题目有点茫然，但抽象一下就。。。。。。。。。

其实塔好积木再拿走，就相当于当初搭的时候没选拿走的积木。这样一转化思维，问题就清楚了。把积木可搭建的最大高度看做背包的载重，每块积木的高度就是物品的重量。也就是用给定的物品装指定的包，使每个包装的物品一样多，且在符合条件的前提下尽量多。

这样就变成经典的背包问题了。

对于每一个城堡求一次，最终找到每一个城堡都可达到的最大高度即可。

### 【源代码 1】

```
const
maxhig=7000;
maxn=100;
var
n,top:longint;
opt:array[0..maxn,0..maxhig]
of boolean;
a:array[0..maxn] of longint;
procedure init;
var
i:longint;
begin
readln(n);
fillchar(opt,sizeof(opt),false);
for i:=1 to n do
opt[i,0]:=true;
end;
function
can(m:longint):boolean;
var
```

```
i:longint;
begin
can:=true;
for i:=1 to n do
if not opt[i,m] then
exit(false);
end;
procedure main;
var
ii,m,tothig,i,j,ans:longint;
begin
for ii:=1 to n do
begin
top:=0;
read(m);
tothig:=0;
while m>0 do
begin
inc(top);
a[top]:=m;
inc(tothig,m);
```

```
read(m);
end;
for i:=1 to top do
for j:=tothig downto 1 do
if (j-a[i]>=0) and
(opt[ii,j-a[i]]) then
opt[ii,j]:=true;
end;
ans:=maxhig;
while not opt[1,ans] do
dec(ans);
while not can(ans) do
dec(ans);
writeln(ans);
end;
begin
init;
main;
end.
```

回顾上面的内容充分说明**动态规划的本质就是递推**。其实按照我的理解（动规涉及最优决策，递推是单纯的总结）背包问题的简化版更准确点算是递推而非动态规划，至于动归和递推之间的界线本来就很模糊（至少我这么认为）把它看做什么都可以，没必要咬文嚼字。

回到 0/1 背包的原问题上（如果你忘了就去上面看看）。

如果在不知道这个模型的情况下我们怎么做这个题呢？

这就要用到第一节提到的方法二：三要素法。

题目中明确说明对于一个物品要不就拿走要不就放下，其实题目赤裸裸的告诉我们**决策**就是不拿（用 0 表示）或拿（用 1 表示）。这样想都不用想就知道了决策，这也是本题的突破口。知道了决策写个搜索的程序应该是很轻松的了。

那么阶段是什么呢？

显然，给你一堆东西让你往包里塞，你当然是一个一个的拿来，塞进去。那么**阶段**很明显就是物品的个数。

状态又是什么呢？

有的人在装东西时有个习惯（比如说我）就是先把东西分类，然后把同类的东西打个小包，最后再把小包放进去，我们可以按这个思想给物品打一些小包，也就是按照单位为 1 的递增的顺序准备好多个小包，比如载重是 6 的包，可以为它准备载重是 1, 2, 3, 4, 5 的小包。这样**状态**就可以想出来了：

设计状态  $opt[i, j]$  表示装第  $i$  个物品时载重为  $j$  的包可以装到的最大价值。

$opt[i-1, j] \quad (j-w[i]<0, i>0)$

状态转移方程： $opt[i, j]=\max\{opt[i-1, j], opt[i-1, j-w[i]]+v[i]\} \quad (j-w[i]\geq 0, i>0)$

$w[i]$ : 第  $i$  个物品的重量， $v[i]$  第  $i$  个物品的价值

解释：要载重为  $j$  的背包空出  $w[i]$  ( $j-w[i]$ ) 的空间且装上第  $i$  个物品，比不装获得的价值大就装上它。

边界条件： $\text{opt}[0,i]=0$ ; ( $i \in \{1..S\}$ )

注：

这种二维的状态表示应该在下节讲，但是为了方便理解先在这里说了。

上面的方法动态规划三要素都很明显，实现也很简单。但是在我初学背包时却用另外一种一维的状态表示法。

用第一节说的思考方法五（放宽约束和增加约束）再重新思考一下这个问题：

怎么放宽约束呢？

把题目中的价值去掉，不考虑价值即最优就变成背包问题的简化版了。那简化版的求解对我们有何启示呢？

再一次增加约束：背包只能装满。

显然对于  $N$  个装满背包的方案中只要找到一个价值最大的就是问题的解。那么装不满怎么办呢？其实装不满背包，它总要达到一定的重量 ( $X$ )。我们可以把这种情况看作是装满一个载重为  $X$  的小包。

总结一下上面的思维过程：

放宽约束让我们找到问题的突破口——和背包问题简化版一样，我们可以确定载重为  $S$  的背包是否可以装满。

增加约束让我们找到问题的求解方法——在装满背包的方案中选择最优的一个方案。

这样问题就解决了。

设计一个状态  $\text{opt}[j]$  表示装满载重为  $j$  的背包可获得的最大价值。对于第  $i$  个物品，只要  $\text{opt}[j-w[i]]$  可以装满且  $\text{opt}[j-w[i]]+v[i]$  比  $\text{opt}[j]$  大就装上这个物品（更新  $\text{opt}[j]$ ）。

怎么使  $\text{opt}[j]$  既有是否构成又有最优的概念呢？

$\text{opt}[j]$  只表示最优，只不过使初始条件+1，判断  $\text{opt}[j]$  是否为 0，如果  $\text{opt}[j]=0$  说明  $j$  装不满。

边界条件： $\text{opt}[0]=1$ ;

状态转移方程： $\text{opt}[j]=\max\{\text{opt}[j-w[i]]\}$  ( $0 < i < n, w[i] \leq j \leq S$ )

问题解： $\text{ans}=\max\{\text{opt}[i]\}-1$  ( $0 < i \leq s$ )

时间复杂度：阶段数  $O(S)$  \* 状态数 ( $O(N)$ ) \* 转移代价 ( $O(1)$ ) =  $O(SN)$

下面看几个例题：

### 例题 8 采药 (medic.pas/c/cpp) 来源：NOIP2005（普及组） 第三题

#### 【问题描述】

辰辰是个天资聪颖的孩子，他的梦想是成为世界上最伟大的医师。为此，他想拜附近最有威望的医师为师。医师为了判断他的资质，给他出了一个难题。医师把他带到一个到处都是草药的山洞里对他说：“孩子，这个山洞里有一些不同的草药，采每一株都需要一些时间，每一株也有它自身的价值。我会给你一段时间，在这段时间里，你可以采到一些草药。如果你是一个聪明的孩子，你应该可以让采到的草药的总价值最大。”

如果你是辰辰，你能完成这个任务吗？

#### 【输入文件】

输入文件 medic.in 的第一行有两个整数  $T$  ( $1 \leq T \leq 1000$ ) 和  $M$  ( $1 \leq M \leq 100$ )，用一个空格隔开， $T$  代表总共能够用来采药的时间， $M$  代表山洞里的草药的数目。接下来的  $M$  行每行包括两个在 1 到 100 之间（包括 1 和 100）的整数，分别表示采摘某株草药的时间和这株草药的价值。

#### 【输出文件】

输出文件 medic.out 包括一行，这一行只包含一个整数，表示在规定的时间内，可以采到的草药的最大总价值。

#### 【输入样例】

```
70 3
71 100
69 1
1 2
```

#### 【输出样例】

```
3
```

#### 【数据规模】

对于 30% 的数据， $M \leq 10$ ;

对于全部的数据， $M \leq 100$ 。

#### 【提交链接】

[http://www.rqnoj.cn/Problem\\_15.html](http://www.rqnoj.cn/Problem_15.html)

#### 【问题分析】

这是一道典型的 0/1 背包问题，把时间看做标准模型中的重量，把规定的时间看做载重为  $T$  的背包，这样问题和基本模型就一样了，具体实现这里不多说了。

【源代码 1】 { 二维状态 }

program medic;

const

fin='medic.in';

fout='medic.out';

maxt=1010;

<pre> maxn=110; var   opt:array[0..maxn,0..maxt] of longint;   w,v:array[0..maxn]      of longint;   t,n:longint; procedure init; var   i:longint; begin   assign(input,fin);   reset(input);   assign(output,fout);   rewrite(output);   read(t,n);   for i:=1 to n do     read(w[i],v[i]);   close(input); end; function max(x,y:longint):longint; begin   if x&gt;y then max:=x   else max:=y; end; procedure main; var   i,j:longint; begin   fillchar(opt,sizeof(opt),0);   for i:=1 to n do     for j:=1 to t do </pre>	<pre>       if j-w[i]&lt;0 then         opt[i,j]:=opt[i-1,j]       else         opt[i,j]:=max(opt[i-1,j],opt[i-1,j-w [i]]+v[i]);       end;     procedure print;     begin       writeln(opt[n,t]);       close(output);     end;     begin       init;       main;       print;     end.     【源代码 2】 {一维状态}     program medic;     const       fin='medic.in';       fout='medic.out';       maxt=1010;       maxn=110;     var       opt:array[0..maxt] of longint;       w,v:array[0..maxn]      of longint;       ans,t,n:longint;     procedure init;     var       i:longint;     begin       assign(input,fin); </pre>	<pre>       reset(input);       assign(output,fout);       rewrite(output);       readln(t,n);       for i:=1 to n do         read(w[i],v[i]);       close(input);     end;     procedure main;     var       i,j:longint;     begin       fillchar(opt,sizeof(opt),0);       opt[0]:=1;       for i:=1 to n do         for j:=t downto w[i] do           if (opt[j-w[i]]&gt;0) and (opt[j-w[i]]+v[i]&gt;opt[j]) then             opt[j]:=opt[j-w[i]]+v[i];           ans:=maxlongint;           for i:=1 to t do             if opt[i]&gt;ans then ans:=opt[i];           end;           procedure print;           begin             writeln(ans-1);             close(output);           end;           begin             init;             main;             print;           end. </pre>
---	---	---

### 例题 9 开心的金明 来源：NOIP2006（普及组）第二题

#### 【问题描述】

金明今天很开心，家里购置的新房就要领钥匙了，新房里有一间他自己专用的很宽敞的房间。更让他高兴的是，妈妈昨天对他说：“你的房间需要购买哪些物品，怎么布置，你说了算，只要不超过 N 元钱就行”。今天一早金明就开始做预算，但是他想买的东西太多了，肯定会超过妈妈限定的 N 元。于是，他把每件物品规定了一个重要度，分为 5 等：用整数 1~5 表示，第 5 等最重要。他还从因特网上查到了每件物品的价格（都是整数元）。他希望在不超过 N 元（可以等于 N 元）的前提下，使每件物品的价格与重要度的乘积的总和最大。设第 j 件物品的价格为 v[j]，重要度为 w[j]，共选中了 k 件物品，编号依次为 j1...jk，则所求的总和为：v[j1]\*w[j1]+...+v[jk]\*w[jk] 请你帮助金明设计一个满足要求的购物单。

#### 【输入文件】

输入的第 1 行，为两个正整数，用一个空格隔开：N m

（其中 N（<30000）表示总钱数，m（<25）为希望购买物品的个数。）

从第 2 行到第 m+1 行，第 j 行给出了编号为 j-1 的物品的的基本数据，每行有 2 个非负整数 v p

（其中 v 表示该物品的价格（v≤10000），p 表示该物品的重要度（1~5））

#### 【输出文件】

输出只有一个正整数，为不超过总钱数的物品的价格与重要度乘积的总和的最大值（<100000000）

#### 【输入样例】

```

1000 5
800 2
400 5
300 5
400 3
200 2

```

#### 【输出样例】

```

3900

```

#### 【提交链接】

[http://www.rqnoj.cn/Problem\\_2.html](http://www.rqnoj.cn/Problem_2.html)

#### 【问题分析】

这仍然是一道典型的 0/1 背包，只不过注意这个问题中的价值对应背包模型中的重量，这个问题中的价值和重要度的乘积是背包模型中的价值。（很饶口啊）。

具体实现同背包模型一样，这里不多说了。

#### 【源代码】

```
program p2;
const
  maxn=30010;
  maxm=30;
var
  opt:array[0..maxn] of longint;
  v,p:array[0..maxm] of longint;
  n,m:longint;
procedure init;
var
  i:longint;
begin
  readln(n,m);
  for i:=1 to m do
    readln(v[i],p[i]);
end;
procedure main;
var
  i,j:longint;
begin
  fillchar(opt,sizeof(opt),0);
  opt[0]:=1;
  for i:=1 to m do
    begin
      for j:=n downto 1 do
        begin
          if (j-v[i]>=0) and
             (opt[j-v[i]]>0) and
             (opt[j-v[i]]+v[i]*p[i]>opt[j])then
            opt[j]:=opt[j-v[i]]+v[i]*p[i];
        end;
      end;
    end;
end;
procedure print;
var
  i,ans:longint;
begin
  ans:=0;
  for i:=1 to n do
    if opt[i]>ans then
      ans:=opt[i];
  writeln(ans-1);
end;
begin
  init;
  main;
  print;
end.
```

#### 例题 10 金明的预算方案 (budget.pas/c/cpp) 来源: NOIP2006 第二题

#### 【问题描述】

金明今天很开心，家里购置的新房就要领钥匙了，新房里有一间金明自己专用的很宽敞的房间。更让他高兴的是，妈妈昨天对他说：“你的房间需要购买哪些物品，怎么布置，你说了算，只要不超过 N 元钱就行”。今天一早，金明就开始做预算了，他把想买的物品分为两类：主件与附件，附件是从属于某个主件的，下表就是一些主件与附件的例子：

主件	附件
电脑	打印机，扫描仪
书柜	图书
书桌	台灯，文具
工作椅	无

如果要买归类为附件的物品，必须先买该附件所属的主件。每个主件可以有 0 个、1 个或 2 个附件。附件不再有从属于自己的附件。金明想买的东西很多，肯定会超过妈妈限定的 N 元。于是，他把每件物品规定了一个重要度，分为 5 等：用整数 1~5 表示，第 5 等最重要。他还从因特网上查到了每件物品的价格（都是 10 元的整数倍）。他希望在不超过 N 元（可以等于 N 元）的前提下，使每件物品的价格与重要度的乘积的总和最大。

设第 j 件物品的价格为 v[j]，重要度为 w[j]，共选中了 k 件物品，编号依次为 j1, j2, …, jk，则所求的总和为：

$v[j_1]*w[j_1]+v[j_2]*w[j_2]+ \dots +v[j_k]*w[j_k]$ 。（其中\*为乘号）

请你帮助金明设计一个满足要求的购物单。

#### 【输入文件】

输入文件 budget.in 的第 1 行，为两个正整数，用一个空格隔开：N m

（其中 N（<32000）表示总钱数，m（<60）为希望购买物品的个数。）

从第 2 行到第 m+1 行，第 j 行给出了编号为 j-1 的物品的的基本数据，每行有 3 个非负整数：v p q

（其中 v 表示该物品的价格（v<10000），p 表示该物品的重要度（1~5），q 表示该物品是主件还是附件。如果 q=0，表示该物品为主件，如果 q>0，表示该物品为附件，q 是所属主件的编号）

#### 【输出文件】

输出文件 budget.out 只有一个正整数，为不超过总钱数的物品的价格与重要度乘积的总和的最大值（<200000）。

#### 【输入样例】

```
1000 5
800 2 0
400 5 1
300 5 1
400 3 0
500 2 0
```



【输出样例】

2200

【提交链接】

[http://www.rqnoj.cn/Problem\\_6.html](http://www.rqnoj.cn/Problem_6.html)

【问题分析】

这道题是一道典型的背包问题，比较麻烦的就是它还存在附件和主件之分。但这并不影响解题，由于附件最多两个，那么我们对主、附件做一个捆绑就行了。

(1) 标准算法

因为这道题是典型的背包问题，显然标准算法就是动态规划。由于我们要对主、附件做捆绑，由于题目没有直接给出每个主件对应的附件，所以还需要做一个预处理：另开两个数组  $q1$ ,  $q2$  来分别记录对应的第  $i$  个主件的附件。那么对于附件不需要处理。而主件的花费就有 4 种情况了。(下面用  $W$  表示花费)

$W1=v[i]$  (只买主件)

$W2=v[i]+v[q1[i]]$  (买主件和第一个附件)

$W3=v[i]+v[q2[i]]$  (买主件和第二个附件)

$W4=v[i]+v[q1[i]]+v[q2[i]]$  (买主件和那两个附件)

设计一个状态  $opt[i]$  表示花  $i$  元钱可买到的物品的价格个重要度最大值。边界条件是  $opt[0]=0$  但是为了区分花  $i$  元钱是否可买到物品我们把初始条件  $opt[0]=1$ ; 这样  $opt[i]>0$  说明花  $i$  元可以买到物品。这样就不难设计出这个状态的转移方程来：

$opt[i]=\max\{opt[i],opt[i-wj]\}$   $((i-wj>0) \text{ and } (opt[i-wj]>0)) (0<j\leq 4)$

显然题目的解就是  $opt[1]$  到  $opt[n]$  中的一个最大值。但在输出是要注意将解减 1。

注：价格是 10 的整数倍所以读入数据时可以使  $n=n \div 10, w_i=w_i \div 10$

【源代码】

<pre>program budget; const fin='budget.in'; fout='budget.out'; maxn=3200; maxm=100; var n,m,ans:longint; v,p,q1,q2,q:array[0..maxm] of longint; opt:array[0..maxn] of longint; procedure init; var i,x:longint; begin assign(input,fin); reset(input); assign(output,fout); rewrite(output); readln(n,m); n:=n div 10; fillchar(q1,sizeof(q1),0); fillchar(q2,sizeof(q2),0); for i:=1 to m do begin readln(v[i],p[i],q[i]); v[i]:=v[i] div 10;</pre>	<pre>q2[q[i]]:=q1[q[i]]; q1[q[i]]:=i; end; close(input); end; function max(x,y:longint):longint; begin if x&gt;y then exit(x); exit(y); end; procedure main; var i,j:longint; begin fillchar(opt,sizeof(opt),0); opt[0]:=1; for j:=1 to m do for i:=n downto v[j] do if q[j]=0 then begin if (i-v[j]&gt;=0) and (opt[i-v[j]]&gt;0) then opt[i]:=max(opt[i],opt[i-v[j]] +v[j]*p[j]); if (i-v[j]-v[q1[j]]&gt;=0) and (opt[i-v[j]-v[q1[j]]]&gt;0) then opt[i]:=max(opt[i],opt[i-v[j]-</pre>	<pre>v[q1[j]]+v[j]*p[j]+v[q1[j]]*p[q1[ j]]); if (i-v[j]-v[q2[j]]&gt;=0) and (opt[i-v[j]-v[q2[j]]]&gt;0) then opt[i]:=max(opt[i],opt[i-v[j]- v[q2[j]]+v[j]*p[j]+v[q2[j]]*p[q2[ j]]); if (i-v[j]-v[q1[j]]-v[q2[j]]&gt;=0) and (opt[i-v[j]-v[q1[j]]-v[q2[j]]]&gt;0) then opt[i]:=max(opt[i],opt[i-v[j]- v[q1[j]]-v[q2[j]]+v[j]*p[j]+v[q1[ j]]*p[q1[j]]+v[q2[j]]*p[q2[j]]); ans:=max(ans,opt[i]); end; end; procedure print; begin writeln((ans-1)*10); close(output); end; begin init; main; print; end.</pre>
--	--	---

上面提到的几个例题都是最基础的题目，而且把问题抽象后就与背包问题的基本模型一样了，但有些题目用到了基本模型，要求的解却不一定跟模型一样，下面看个例子：

### 例题 11 Money Systems (money.pas/c/cpp) 来源：USACO 2.3

【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。[In their own rebellious way], 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, 和 100 的单位面值组成的。

母牛想知道有多少种不同的方法用货币系统中的货币来构造一个确定的数值。

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。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定数量的面值。保证总数将会适合 long long (C/C++) 和 Int64 (Free Pascal)。

【输入文件】

货币系统中货币的种类数目是  $V$  ( $1 \leq V \leq 25$ )。要构造的数量钱是  $N$  ( $1 \leq N \leq 10,000$ )。

第 1 行：二整数， $V$  和  $N$

第 2 行：可用的货币  $V$  个整数。

【输出文件】

单独的一行包含那个可能的构造的方案数。

【输入样例】

3 10

1 2 5

【输出样例】

10

【问题分析】

把钱面值，要构造的钱看做载重为  $N$  的背包，这个问题便是 0/1 背包的简化版了，但这个问题和传统模型有所差异，不是判断  $N$  是否可构成，而是求构成  $N$  的方案，而且这里的面值是可以重复利用的（你可以看做是物品有无限多）。

对于第一个问题，只要把原来 BOOLEAN 型的状态改为 INT64，在递推过程中累加方案数即可。

对于第二个问题，基本模型中为了避免重复在内重循环枚举背包载重时采用倒循环，现在只要正向循环就 OK 了。

复杂度与原模型相同。

【源代码】

```
{
ID:hhzhaojia2
PROG:money
LANG:PASCAL
}
program money;
const
fin='money.in';
fout='money.out';
maxv=100;
maxn=10010;
var
a:array[0..maxv] of longint;
opt:array[0..maxn] of int64;
v,n:longint;
procedure init;
```

```
var
i:longint;
begin
assign(input,fin);
reset(input);
assign(output,fout);
rewrite(output);
read(v,n);
for i:= 1 to v do
read(a[i]);
close(input);
end;
procedure main;
var
i,j:longint;
begin
fillchar(opt,sizeof(opt),0);
```

```
opt[0]:=1;
for i:=1 to v do
for j:=a[i] to n do
inc(opt[j],opt[j-a[i]]);
end;
procedure print;
begin
writeln(opt[n]);
close(output);
end;
begin
init;
main;
print;
end.
```

背包问题方案的求法：

和大多数 DP 问题的方案的求法一样，增加一个数组 path 和状态维数相同用来记录当前状态的决策就 OK 了。

输出方案时候通过当前决策推出上一决策，这一连串的决策序列就是要求的方案。

下面看这样一个数据：

载重：6 物品个数：3

重量 价值

物品 1: 3 10

物品 2: 2 2

物品 3: 1 9

一维状态求解过程：

i=1: (枚举物品)

opt[0..6]=1 0 0 11 0 0 0

path[0..6]=0 0 0 1 0 0 0 {记录最后装入包中的物品的编号}

i=2

opt[0..6]=1 0 3 11 0 13 0

path[0..6]=0 0 2 1 0 2 0

i=3

opt[0..6]=1 10 3 12 20 13 22

path[0..6]=0 3 2 3 3 2 3

二维状态求解过程： (略)

可以看到一维状态的最优解是正确的，但细心分析发现一个惊人的问题：方案不对！

什么最优解正确而方案不正确呢？

因为在解  $i=3$  时  $opt[6]$  用到的方案数应该是  $9+2+10=21$ 。显然这个方案是正确的，所以最优解正确。但是求解完  $opt[6]$  后，接着求解  $opt[3]$  却把原来的  $opt[3]=10$  改成了  $opt[3]=2+9=11$  这样，在整个求解过程结束后最后的方案  $opt[6]=9+2+10$  就变成了  $opt[6]=9+2+2+9$  也就是说 1, 2 两个物品装了两次。

这也正是我要说的下面的问题；

背包问题一维状态于二维状态的优劣：

显然，**一维状态的维数少空间复杂度低**。甚至在一些问题上可以减轻思考负担。既然这样是不是我们就应该摒弃二维状态解法呢？

由于一维状态在**求解方案时存在错误**，所以二维状态还是很有用啊。当然有些问题虽然也是在求解方案但要求方案唯一这样就又可以用一维状态了。

看到这里觉得头晕就上趟厕所，返回来看下面的例题：

### 例题 12 新年趣事之打牌 来源： vijos P1071

#### 【问题描述】

过年的时候，大人们最喜欢的活动，就是打牌了。xiaomengxian 不会打牌，只好坐在一边看着。

这天，正当一群人打牌打得起劲的时候，突然有人喊道：“这副牌少了几张！”众人一数，果然是少了。于是这副牌的主人得意地说：“这是一幅特制的牌，我知道整副牌每一张的重量。只要我们称一下剩下的牌的总重量，就能知道少了哪些牌了。”大家都觉得这个办法不错，于是称出剩下的牌的总重量，开始计算少了哪些牌。由于数据量比较大，过了不久，大家都算得头晕了。

这时，xiaomengxian 大声说：“你们看我的吧！”于是他拿出笔记本电脑，编出了一个程序，很快就把缺少的牌找了出来。

如果是你遇到了这样的情况呢？你能办到同样的事情吗？

#### 【输入文件】

第一行一个整数 TotalW，表示剩下的牌的总重量。

第二行一个整数 N ( $1 < N \leq 100$ )，表示这副牌有多少张。

接下来 N 行，每行一个整数  $W_i$  ( $1 \leq W_i \leq 1000$ )，表示每一张牌的重量。

#### 【输出文件】

如果无解，则输出“0”；如果有多解，则输出“-1”；否则，按照升序输出丢失的牌的编号，相邻两个数之间用一个空格隔开。

#### 【输入样例】

270

4

100

110

170

200

#### 【输出样例】

2 4

#### 【提交链接】

[http://www.vijos.cn/Problem\\_Show.asp?id=1071](http://www.vijos.cn/Problem_Show.asp?id=1071)

#### 【问题分析】

如果你认真的做了前面的题，把这个题目抽象成背包问题对你来说应该易如反掌了，我就不多说了。

因为这个问题要求多方案时输出-1，也就是说要输出的方案是唯一的，这时你就不需要担心一维状态的正确性了，可以放心的用一维求解，但要注意只有当前状态没有方案时才记录当前的方案，否则会把正确方案替换了。

#### 【源代码 1】

```
program P1071;
const
  maxw=100010;
  maxn=110;
var
  path,opt:array[0..maxw] of
    int64;
  w:array[0..maxn] of longint;
  ans:array[0..maxn] of
    boolean;
  n,total:longint;
procedure init;
var
```

```
  i:longint;
begin
  read(total);
  read(n);
  for i:=1 to n do
    read(w[i]);
end;
procedure main;
var
  i,j:longint;
begin
  fillchar(opt,sizeof(opt),0);
  fillchar(ans,sizeof(ans),true);
  opt[0]:=1;
```

```
  for i:=1 to n do
    for j:=total downto w[i] do
      if opt[j-w[i]]>0 then
        begin
          if opt[j]=0 then
            path[j]:=i; {只有当前
            状态没求过才记录方案}
          inc(opt[j],opt[j-w[i]]);
        end;
      if opt[total]=0 then
        begin
          writeln('0');
          halt;
        end;
```

<pre> if opt[total]&gt;1 then begin writeln('-1'); halt; end; i:=total; while i&gt;0 do begin ans[path[i]]:=false; </pre>	<pre> i:=i-w[path[i]]; end; end; procedure print; var i:longint; begin for i:=1 to n do if ans[i] then write(i,' '); </pre>	<pre> end; begin init; main; print; end. </pre>
---	---	---

### 1.3 其它问题

一维动态规划最常见的就是前面总结的**最长下降/非降子序列**和**0/1 背包问题**了,当然还有别的一些题。由于不是很常见所以没有固定的解题模式,到时候具体问题具体分析。下面再看一个例子:

**例题 13 挖地雷问题** (P3.pas/c/cpp) 来源: NOIP1996 (提高组) 第三题 (有改动)

#### 【问题描述】

在一个地图上有  $N$  个地窖 ( $N \leq 20$ ), 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。

当地窖及其连接的数据给出之后, 某人可以从任一处开始挖地雷, 然后可以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, 使某人能挖到最多的地雷。

#### 【输入文件】

$N$ : (表示地窖的个数)  
 $W1, W2, W3, \dots, WN$  (表示每个地窖中埋藏的地雷数量)  
 $A12 \dots \dots \dots A1N$  ( $a_{ij}=1$  表示第  $i$  个地窖与第  $j$  个地窖有连接,  $=0$  表示无连接)  
 $A23 \dots \dots \dots A2N$   
 $\dots \dots$   
 $AN-1 \quad AN$

#### 【输出文件】

$K1-K2-\dots-KV$  (挖地雷的顺序)  
 $MAX$  (挖地雷的数量)

#### 【输入样例】

```

5
10 8 4 7 6
1 1 1 0
  0 0 0
    1 1
      1

```

#### 【输出样例】

```

1 -> 3 -> 4 -> 5
max=27

```

#### 【Hint】

题目中的路径是有向的且无环路 (这是我做的改动, 原题中没有要求)。

#### 【问题分析】

看到题目的第一印象是贪心——以一点出发找与他连接的地窖中地雷数最多的一个。

但很容易想到反例:

```

5
1 2 1 1 100
1 1 0 0
  0 1 0
    0 1
      0

```

按照贪心答案是 4, 但实际上答案是 102。

于是就不得不放弃贪心的想法。

但是贪心给了我们启示: 从一个顶点出发要选择向一个与他相连且以该点出发可以挖到较多雷的点走。(有点拗口)

另一种解释: 如果一个顶点连同  $N$  个分量, 显然选择一个较大的就是问题的解答, 这个定义是满足最优化原理的。

那它满足无后效性么?

因为图是有向的, 所以以与该顶点相连的点在往下走的路线中不包括该点。也就是说图是一个 AOV 网 (有向无环图)。

既然满足最优化原理，且无后效性，我们就可以用动态规划解了。

这个问题的阶段就是拓扑序列，但由于输入是倒三角形，所以我们没必要要求拓扑序列，只要从 N 倒着求解就可以了。

设计状态  $opt[i]$  表示以  $i$  点出发可以挖到最多的雷的个数。

状态转移方程： $opt[i]=\max\{opt[j]\}+w[i]$  ( $g[i,j]=1$ )

( $g$  存图， $w[i]$  存第  $i$  个地窖中的雷的个数)。

时间复杂度：

状态数  $O(n)$ \*转移代价  $O(n)=O(n^2)$

这个题目还要求出路径，可以用一个辅助数组  $path$  来记录， $path[i]$  表示从第  $i$  个出发走到的下一个点的编号。求解完只要按  $path$  记录的路径输出即可。

【源代码】

```
program P3;
const
  fin='P3.in';
  fout='P3.out';
  maxn=200;
var
  g:array[0..maxn,0..maxn] of
longint;
  n,ans:longint;
  w,opt,path:array[0..maxn] of
longint;
procedure init;
var
  i,j:longint;
begin
  assign(input,fin);
  reset(input);
  assign(output,fout);
  rewrite(output);
  read(n);
  fillchar(g,sizeof(g),0);
  for i:=1 to n do
    read(w[i]);
```

```
for i:=1 to n do
  for j:=i+1 to n do
    read(g[i,j]);
  close(input);
end;
procedure main;
var
  i,j:longint;
begin
  fillchar(opt,sizeof(opt),0);
  fillchar(path,sizeof(path),0);
  for i:=n downto 1 do
    begin
      for j:=i+1 to n do
        if (g[i,j]=1) and
          (opt[j]>opt[i]) then
          begin
            opt[i]:=opt[j];
            path[i]:=j;
          end;
      inc(opt[i],w[i]);
    end;
    ans:=1;
    for i:=2 to n do
```

```
if
  then ans:=i;
end;
procedure print;
var
  i:longint;
begin
  write(ans);
  i:=path[ans];
  while i>0 do
    begin
      write('-->',i);
      i:=path[i];
    end;
  writeln;
  writeln('max=',opt[ans]);
  close(output);
end;
begin
  init;
  main;
  print;
end.
```

## 2.状态是二维的

通过前面的学习，我想应该对动态规划不陌生了，我学习动态规划是没这么系统，二维，一维一起上。二维状态的动态规划是重中之重。

所谓二维状态就是说一般设计的状态是  $opt[i,j]$  形式的。那  $i,j$  可以代表什么呢？

有很多朋友问过我这个问题，我的回答是：

(1) $i,j$  组合起来代表一个点的坐标（显然是平面坐标系）（如：[街道问题](#)）。

(2) $i,j$  组合表示一个矩阵的单元的位置（第  $i$  行，第  $j$  列）（如：[数塔问题](#)）

(3)起点为  $i$  长度为  $j$  的区间。（如：[回文词](#)）

(4)起点为  $i$  终点为  $j$  的区间。（如：[石子合并问题](#)）

(5)两个没关联的事物，事物 1 的第  $i$  个位置，对应事物 2 的第  $j$  个位置（[花店橱窗设计](#)）

(6)两个序列，第一个序列的前  $i$  个位置或第  $i$  个位置对应第 2 个序列的第  $j$  个位置或前  $j$  个位置。（[最长公共子序列](#)）。

(7)其它

下面通过例题和基本模型进一步说明：

### 2.1 数塔问题

数塔问题来源于一道经典的 IOI 的题目，直接说题，通过题目总结共性。以后遇到类似的题目可以参照这个模型。

**例题 14 数塔问题** (numtri.pas/c/cpp) 来源：IOI94

【问题描述】

考虑在下面被显示的数字金字塔。

写一个程序来计算从最高点开始在底部任意处结束的路径经过数字的和的最大（小）。每一步可以走



到左下方的点也可以到达右下方的点。

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

在上面的样例中,从 7 到 3 到 8 到 7 到 5 的路径产生了最大和:30

【输入文件】

第一个行包含  $R(1 \leq R \leq 1000)$  , 表示行的数目。

后面每行为这个数字金字塔特定行包含的整数。

所有被供应的整数是非负的且不大于 100。

【输出文件】

单独的一行包含那个可能得到的最大的和。

【输入样例】

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

【输出样例】

```
30
```

【提交链接】

<http://poj.org/problem?id=1163>

【问题分析】

这个问题是学习动态规划最简单最经典的问题,说它经典是因为它的阶段、状态、决策都十分明显。

刚看到题目觉得没有入手点,连怎么储存,描述这个金字塔都是问题,看输入样例发现:数字金字塔可以变成像输入样例那样的下三角,这样可以用一个二维数组  $a$  储存它,并且可以用  $(i,j)$  描述一个数字在金字塔中的位置。

对于中间的一个点来说,想经过它则必须经过它的上方或左上(针对变化后的三角形)。也就是说经过这个点的数字和最大等于经过上方或左上方所得的“最大和”中一个更大的加上这个点中的数字。显然这个定义满足最优子结构。

这样阶段很明显就是金字塔的层,设计一个二维状态  $opt[i,j]$  表示走到第  $i$  行第  $j$  列时经过的数字的最大和。决策就是  $opt[i-1,j]$  或  $opt[i-1,j-1]$  中一个更大的加上  $(i,j)$  点的数字。

对于一个点只考虑上面或左上即前一阶段,满足无后效性。

状态转移方程(顺推):

$$opt[i,j] = \begin{cases} opt[i-1,j] + a[i,j] & (j=1) \\ opt[i-1,j-1] + a[i,j] & (j=i) \\ \max\{opt[i-1,j], opt[i-1,j-1]\} + a[i,j] & (1 < j < i) \end{cases}$$

实现时可以将  $opt[i,j]$  的左右边界定义的大点,初始  $opt[i,j]=0$

由于在  $j=1$  时  $opt[i-1,j-1]=0, opt[i-1,j] \geq 0$  所以方程也可以这样写:

$$opt[i,j] = \max\{opt[i-1,j], opt[i-1,j-1]\} + a[i,j]$$

同理  $j=i$  时方程也可以写成上面那样,所以方程综合为:

$$opt[i,j] = \max\{opt[i-1,j], opt[i-1,j-1]\} + a[i,j] \quad (0 < j \leq i)$$

显然答案是走到底后的一个最大值,即:

$$ans = \max\{opt[n,i]\} \quad (1 \leq i \leq n)$$

其实从上往下走和从下往上走结果是一样的,但是如果从下往上走结果就是  $opt[1,1]$  省下求最大值了,所以方程进一步改动(逆推):

$$opt[i,j] = \max\{opt[i+1,j], opt[i+1,j+1]\} + a[i,j] \quad (0 < j \leq i)$$

复杂度: 状态数  $O(N^2)$  \* 转移代价  $O(1) = O(N^2)$ 。

【源代码】

```
program numtri;
const
  fin='numtri.in';
  fout='numtri.out';
  maxn=1010;
var
  a,opt:array[0..maxn,0..maxn]
of longint;
  n:longint;
```

```
procedure init;
var
  i,j:longint;
begin
  assign(input,fin);
  reset(input);
  assign(output,fout);
  rewrite(output);
  read(n);
  for i:=1 to n do
```

```
  for j:=1 to i do
    read(a[i,j]);
  close(input);
end;
procedure main;
var
  i,j:longint;
begin
  fillchar(opt,sizeof(opt),0);
  for i:=n downto 1 do
```

<pre> for j:=1 to n do   if opt[i+1,j]&gt;opt[i+1,j+1] then   opt[i,j]:=opt[i+1,j]+a[i,j]   else   opt[i,j]:=opt[i+1,j+1]+a[i,j]; </pre>	<pre> end; procedure print; begin   writeln(opt[1,1]);   close(output); end; </pre>	<pre> begin   init;   main;   print; end. </pre>
--	---	--

**例题 15 Henry 捡钱** (money.pas/c/cpp) 来源: Dream Team 邀请赛

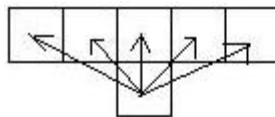
**【问题描述】**

最近, Henry 由于失恋(被某大牛甩掉!)心情很是郁闷。所以, 他去了大牛家, 寻求 Michael 大牛的帮助, 让他尽快从失恋的痛苦中解脱出来。Michael 大牛知道 Henry 是很爱钱的, 所以他是费尽脑水, 绞尽脑汁想出了一个有趣的游戏, 帮助 Henry....

Michael 感觉自己简直是个天才(我们从不那么认为), 就把这个游戏取名为: Henry 拣钱。为了帮助更多的人采用这种方法早日脱离失恋之苦, Michael 特地选在这次 DT 比赛中把游戏介绍给大家.... (大家鼓掌!!!)

其实, 这个游戏相当垃圾, 目的就是为了满足 Henry 这种具有强烈好钱心理的人。游戏是这样的: Michael 首先找到了一块方形的土地, 面积为  $m*n$ (米<sup>2</sup>)。然后他将土地划分为一平方米大小的方形小格。Michael 在每个格子下都埋有钱(用非负数  $s$  表示, 表示人民币的价值为  $s$ )和炸弹(用负数  $s$  表示, 表示 Henry 挖出该方格下的东西会花掉  $s$  的钱去看病, 医炸弹炸伤的伤口)…。游戏的要求就是让 Henry 从一侧的中间列出发, 按照下图的 5 种方式前进(前进最大宽度为 5), 不能越出方格。他每到一个格子, 必定要取走其下相应的东西。直到到达土地的另一侧, 游戏结束。不用说也知道, Henry 肯定想得到最多的人民币。所以他偷窥了 Michael 埋钱的全过程, 绘成了一张矩阵图。由于他自己手动找会很麻烦, 于是他就找到了学习编程的你。请给帮他找出, 最大人民币价值。

拣钱路线规则(只有 5 个方向, 如下图):



图例: ↖

16↖	4↖	3↖	12↖	6↖	0↖	3↖
4↖	-5↖	6↖	7↖	0↖	0↖	2↖
6↖	0↖	-1↖	-2↖	3↖	6↖	8↖
5↖	3↖	4↖	0↖	0↖	-2↖	7↖
-1↖	7↖	4↖	0↖	7↖	-5↖	6↖
0↖	-1↖	3↖	4↖	12↖	4↖	2↖
			H↖			

路径: 12—7—4—6—6—16↖

H 为 Henry 的出发点, 每组数据的出发点都是最后一行的中间位置!

(前方 5 个格子为当前可以到达的)

**【输入文件】**

第一行为  $m\ n$  ( $n$  为奇数), 入口点在最后一行的中间。

接下来为  $m*n$  的数字矩阵。

共有  $m$  行, 每行  $n$  个数字, 数字间用空格隔开, 代表该格子下是钱或炸弹。

为了方便 Henry 清算, 数字全是整数。

**【输出文件】**

一个数, 为你所找出的最大人民币价值。

**【输入样例】**

```

6 7
16 4 3 12 6 0 3
4 -5 6 7 0 0 2
6 0 -1 -2 3 6 8
5 3 4 0 0 -2 7
-1 7 4 0 7 -5 6
0 -1 3 4 12 4 2

```

0 -1 3 4 12 4 2

【输出样例】

51

【数据范围】

N and M<=200。

结果都在 longint 范围内。

【问题分析】

去掉题目华丽的伪装，我们可以这样描述这个题目：

给定一个数字矩阵，从最后一层的中间出发可以向图上所示的 5 个方向走，直到走到第一层，使经过的数字和最大。

如果我们不考虑负数对问题的影响，这个问题的描述就是经典的数塔问题了，只不过将塔变成了矩阵。这样就可以用刚刚讲过的数塔问题的模型解这个题目了，我就不多说了。

状态转移方程：

$opt[i,j]=\max\{opt[i-1,k]\}+a[i,j]$  ( $j-2\leq k\leq j+2$ )

【源代码】

```
program money;
const
  fin='money.in';
  fout='money.out';
  maxn=210;
var
  a,opt:array[0..maxn,0..maxn]
of longint;
  n,m,ans,max:longint;
procedure init;
var
  i,j:longint;
begin
  assign(input,fin);
  reset(input);
  assign(output,fout);
  rewrite(output);
  readln(n,m);
```

```
  for i:=1 to n do
    for j:=1 to m do
      read(a[i,j]);
    end;
  procedure main;
  var
    i,j,k:longint;
  begin
    for i:=1 to n do
      for j:=1 to m do
        begin
          max:=-maxlongint;
          for k:=j-2 to j+2 do
            if (k>0) and (k<=m) and
              (opt[i-1,k]>max) then
              max:=opt[i-1,k];
          opt[i,j]:=max+a[i,j];
        end;
      end;
    ans:=-maxlongint;
```

```
    i:=(1+m) div 2;
    for j:=i-2 to i+2 do
      if (j>0) and (j<=m) and
        (opt[n,j]>ans) then
        ans:=opt[n,j];
    end;
  procedure print;
  begin
    writeln(ans);
    close(input);
    close(output);
  end;
begin
  init;
  main;
  print;
end.
```

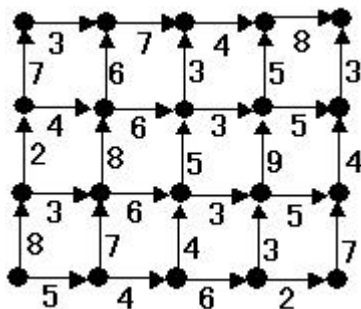
## 2.2 街道问题

和数塔问题一样街道问题也来源于一道典型的例题，下面我们看一下这道题。

**例题 16 街道问题** (way.pas/c/cpp) 来源：《奥赛经典》(提高篇)

【问题描述】

如图所示的矩形图中找到一条从左下角到右上角的最短路径，图中数字表示边的长度。只能向右或向上走。



【输入文件】第一行两个数，N M，矩形的点有 N 行 M 列。(0<N, M<1000) 接下来 N 行每行 M-1 个数描述横向边的长度。接下来 N-1 行每行 M 个数描述纵向边的长度。边的长度小于 10。

【输出文件】一个数——最短路径长度。

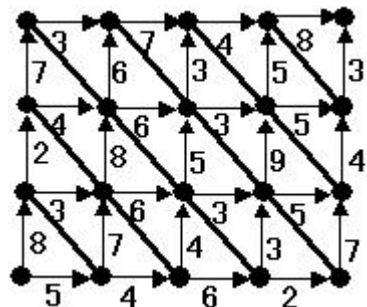
【输入样例】

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

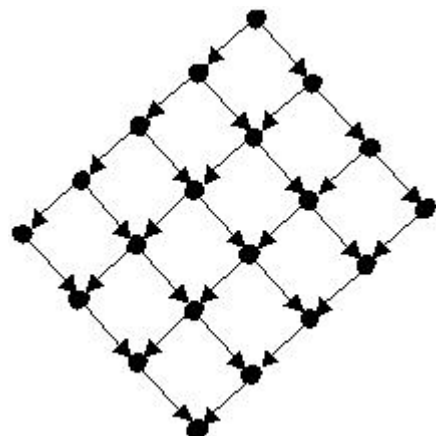
7 6 3 5 3  
2 8 5 9 4  
8 7 4 3 7  
【输出样例】

28  
【问题分析】

因为只能向右或向上走，所以阶段应该是这样的：



如果把图再做改动看看：



这样想就是上面说的数塔问题了，只不过数塔问题的数在点上而街道问题的数在边上。但是并不影响问题的求解我们可以用数塔问题的思路来解这个问题。

设计一个二维状态  $opt[i,j]$  表示走到  $(i,j)$  的最短路径，显然这个路径只可能是左边或上边走来的，所以决策就是这两个方向上加上经过的边的和中一个较短的路。于是有下面的状态转移方程：

$$opt[i,j] = \begin{cases} opt[i+1,j] + z[i,j] & (j=1) \\ opt[i,j-1] + h[i,j] & (i=n) \\ \min\{opt[i+1,j] + z[i,j], opt[i,j-1] + h[i,j]\} & (0 < i \leq n, 0 < j \leq m) \end{cases}$$

和数塔问题一样，这个问题也可以做类似的预处理：初始化  $opt$  的值是一个很大的数，保证解不会超过他，但要注意不要太大了，太大了可能有 **2 2 5 问题**。  $opt[0,0]=0$ 。这样就可以把方程整理为：

$$opt[i,j] = \min\{opt[i+1,j] + z[i,j], opt[i,j-1] + h[i,j]\}$$

复杂度：状态数  $O(N^2)$  \* 转移代价  $O(1)$  =  $O(N^2)$ 。

这一类问题是很经典的问题。

思考这样一个问题：如果让找出一条最短路径，一条较短路径，且两条路径不重合该怎么办呢？

这个问题先留给大家思考，在后面的 **多维状态** 中会详细的讲。

【源代码】

```
program way;
const
  fin='way.in';
  fout='way.out';
  maxn=1010;
var
  h,z,opt:array[0..maxn,0..max
n] of longint;
  n,m:longint;
procedure init;
var
  i,j:longint;
```

```
begin
  assign(input,fin);
  reset(input);
  assign(output,fout);
  rewrite(output);
  read(n,m);
  for i:=1 to n do
    for j:=2 to m do
      read(h[i,j]);
  for i:=1 to n-1 do
    for j:=1 to m do
      read(z[i,j]);
  close(input);
```

```
end;
function
  min(x,y:longint):longint;
begin
  min:=y;
  if x<y then min:=x;
end;
procedure main;
var
  i,j:longint;
begin
  fillchar(opt,sizeof(opt),$7F);
  opt[n,0]:=0;
```

<pre> for i:=n downto 1 do   for j:=1 to m do     opt[i,j]:=min(opt[i+1,j]+z[i,j] , opt[i,j-1]+h[i,j]);   end; </pre>	<pre> procedure print; begin   writeln(opt[1,m]);   close(output); end; </pre>	<pre> begin   init;   main;   print; end. </pre>
---	--	--

还有一道例题是街道问题的变形在[记忆化搜索](#)处会说。

## 2.3 最长公共子序列问题

和前面讲的有所区别，这个问题不涉及走向。很经典的动态规划问题。

**例题 17 最长公共子序列** (lcs.pas/c/cpp) 来源：《全国青少年信息学奥林匹克联赛培训教材》

### 【问题描述】

一个给定序列的子序列是在该序列中删去若干元素后得到的序列。确切地说，若给定序列  $X = \langle x_1, x_2, \dots, x_m \rangle$ ，则另一序列  $Z = \langle z_1, z_2, \dots, z_k \rangle$  是  $X$  的子序列是指存在一个严格递增的下标序列  $\langle i_1, i_2, \dots, i_k \rangle$ ，使得对于所有  $j=1, 2, \dots, k$  有  $X_{i_j} = Z_j$ 。例如，序列  $Z = \langle B, C, D, B \rangle$  是序列  $X = \langle A, B, C, B, D, A, B \rangle$  的子序列，相应的递增下标序列为  $\langle 2, 3, 5, 7 \rangle$ 。

给定两个序列  $X$  和  $Y$ ，当另一序列  $Z$  既是  $X$  的子序列又是  $Y$  的子序列时，称  $Z$  是序列  $X$  和  $Y$  的公共子序列。

例如，若  $X = \langle A, B, C, B, D, A, B \rangle$ ， $Y = \langle B, D, C, A, B, A \rangle$ ，则序列  $\langle B, C, A \rangle$  是  $X$  和  $Y$  的一个公共子序列，但它不是  $X$  和  $Y$  的一个最长公共子序列。序列  $\langle B, C, B, A \rangle$  也是  $X$  和  $Y$  的一个公共子序列，它的长度为 4，而且它是  $X$  和  $Y$  的最长公共子序列，因为  $X$  和  $Y$  没有长度大于 4 的公共子序列。

给定两个序列  $X = \langle x_1, x_2, \dots, x_m \rangle$  和  $Y = \langle y_1, y_2, \dots, y_n \rangle$ ，要求找出  $X$  和  $Y$  的最长公共子序列。

### 【输入文件】

输入文件共有两行，每行为一个由大写字母构成的长度不超过 200 的字符串，表示序列  $X$  和  $Y$ 。

### 【输出文件】

输出文件第一行为一个非负整数，表示所求得的最长公共子序列的长度，若不存在公共子序列，则输出文件仅有一行输出一个整数 0，否则在输出文件的第二行输出所求得的最长公共子序列(也用一个大写字母组成的字符串表示)。

### 【输入样例】

```

ABCBDAB
BDCBA

```

### 【输出样例】

```

4
BCBA

```

### 【提交链接】

<http://poj.org/problem?id=1458>  
[http://www.rqnoj.cn/Problem\\_164.html](http://www.rqnoj.cn/Problem_164.html)

### 【问题分析】

这个问题也是相当经典的。

这个题目的阶段很不明显，所以初看这个题目没什么头绪，不像前面讲的有很明显的上一步，上一层之类的东西，只是两个字符串而且互相没什么关联。

但仔细分析发现还是有入手点的：

既然说是动态规划，那我们首先要考虑的就是怎么划分子问题，一般对于前面讲到的街道问题和数塔问题涉及走向的，考虑子问题时当然是想上一步是什么？但这个问题没有涉及走向，也没有所谓的上一步，该怎么办呢？

既然是求公共子序列，也就有第一个序列的第  $i$  个字符和第二个序列的第  $j$  个字符相等的情况。

那么我们枚举第一个序列 ( $X$ ) 的字符，和第二个序列 ( $Y$ ) 的字符。

显然如果  $X[i] = Y[j]$  那么起点是 1 (下面说的子序列都是起点为 1 的)，长度为  $i$  的子序列和长度为  $j$  的子序列的最长公共子序列就是长度为  $i-1$  和长度为  $j-1$  的子序列中最长的公共子序列加上  $X[i]$  或  $Y[j]$ 。

那要是不相等呢？

如果不相等，也就是说第一个序列长度为  $i$  的子序列和第二个序列中长度为  $j$  的子序列的公共子序列中  $X[i]$  和  $Y[j]$  不同时出现。也就是说第一个序列长度为  $i$  的子序列和第二个序列中长度为  $j$  的子序列的公共子序列是第一个序列长度为  $i$  的子序列和第二个序列中长度为  $j-1$  的子序列或第一个序列长度为  $i-1$  的子序列和第二个序列中长度为  $j$  的子序列的公共子序列中一个更长的。

设计一个状态  $opt[i,j]$  表示起点为 1，第一序列长度为  $i$ ，第二序列长度为  $j$  的子序列的最长公共子序列。按照上面的分类就可以得到状态转移方程：

$$opt[i,j] = \begin{cases} opt[i-1,j-1] + x[i] & (x[i] = y[j]) \\ \max\{opt[i-1,j] + x[i], opt[i,j-1] + y[j]\} & (length(opt[i-1,j]) \geq length(opt[i,j-1])) \\ opt[i,j-1] + y[j] & (length(opt[i-1,j]) < length(opt[i,j-1])) \end{cases}$$

( $0 < i \leq length(X), 0 < j \leq length(Y)$ )



复杂度：状态数  $O(N^2)$  \* 转移代价  $O(1) = O(N^2)$ 。

【源代码】

```
program LCS;
const
  fin='LCS.in';
  fout='LCS.out';
  maxn=300;
var
  s1,s2:string;
  opt:array[0..maxn,0..maxn]
of string;
  L1,L2:longint;
procedure init;
begin
  assign(input,fin);
  reset(input);
  assign(output,fout);
  rewrite(output);
  readln(s1);
```

```
  readln(s2);
  L1:=length(s1);
  L2:=length(s2);
  close(input);
end;
procedure main;
var
  i,j:longint;
begin
  for i:=1 to L1 do
    for j:=1 to L2 do
      opt[i,j]:= '';
      for i:=1 to L1 do
        for j:=1 to L2 do
          if s1[i-1]=s2[j-1] then
            opt[i,j]:=opt[i-1,j-1]+s1[i-1]
          else
            if
              length(opt[i-1,j])>=length(opt[i,j-
```

```
1]) then
  opt[i,j]:=opt[i-1,j]
else opt[i,j]:=opt[i,j-1];
end;
procedure print;
begin
  writeln(length(opt[L1,L2]));
  write(opt[L1,L2]);
  close(output);
end;
begin
  init;
  main;
  print;
end.
```

**例题 18 回文词** (palin.pas/c/cpp) 来源：IOI 2000

【问题描述】

回文词是一种对称的字符串——也就是说，一个回文词，从左到右读和从右到左读得到的结果是一样的。任意给定一个字符串，通过插入若干字符，都可以变成一个回文词。你的任务是写一个程序，求出将给定字符串变成回文词所需插入的最少字符数。

比如字符串“Ab3bd”，在插入两个字符后可以变成一个回文词（“dAb3bAd”或“Adb3bdA”）。然而，插入两个以下的字符无法使它变成一个回文词。

【输入文件】

第一行包含一个整数  $N$ ，表示给定字符串的长度， $3 \leq N \leq 5000$

第二行是一个长度为  $N$  的字符串，字符串由大小写字母和数字构成。

【输出文件】

一个整数，表示需要插入的最少字符数。

【输入样例】

```
5
Ab3bd
```

【输出样例】

```
2
```

【提交链接】

<http://poj.org/problem?id=1159>

【问题分析】

所谓回文词（正着读和反着读一样），其实就是从中间断开把后面翻转后与前面部分一样（注意奇数和偶数有区别）。例：

回文词：AB3BA

断开：AB BA （奇数个时去掉中间字符）

翻转：AB AB

这个题目要求出最少填几个数可以使一个字符串变成回文词，也就是说从任意点截断，再翻转后面部分后两个序列有相同的部分不用添字符，不一样的部分添上字符就可以了。例：

回文词：Ab3bd

截断：Ab bd

翻转：Ab db

b 在两个序列里都有，在第二个里添 A 在第一个里添 d 就可以了：

Adb Adb

这样添两个就可以了，显然从别的地方截断添的个数要比这样多。

这样就把原问题抽象成求最长公共子序列问题了。枚举截断点，把原串截断，翻转。求最长公共子序列。答案就是  $len - (ans * 2)$   $len$  是翻转后两个序列的长度和。 $ans$  是最长公共子序列的长度。

其实这样求解很麻烦，做了好多重复的工作。仔细想想既然在最后求解  $ans$  还要乘 2 那么在先前计算时直接把原串翻转作为第二个序列和第一个序列求最长公共子序列就可以了。这样最后求解就不用乘 2 了，也不用枚举截断点了。例：

原串：Ab3bd

翻转：db3bA

最长公共子序列 b3b

添加 2 个字符

怎么理解这个优化呢？

其实翻转了序列后字符的先后顺序就变了，求解最长公共子序列中得到的解，是唯一的，也就是说这个序列的顺序是唯一的，如果在翻转后的序列和原串能得到相同的序列，那么这个序列在两个串中字符间的顺序是恒定的，这就满足了回文词的定义（正着读和反着读一样）。所以这个优化是正确的。

注意：

这个问题的数据规模很大，空间复杂度较高（ $O(N^2)$ ）所以要用到滚动数组，如果不知道什么是滚动数组就该往后翻页，因为我在后面的动态规划的优化里会说到。

【源代码 1】

<pre>program P1327; const   maxn=5002; var   a,b:ansistring;   opt:array[0..1,0..maxn] of     longint;   n,ans:longint; function   max(x,y:longint):longint; begin   if x&gt;y then exit(x);   max:=y; end; procedure main;</pre>	<pre>var   i,x,j,k0,k1:longint; begin   fillchar(opt,sizeof(opt),0);   readln(n);   readln(a);   b:= "";   for i:=n downto 1 do     b:=b+a[i];     k0:=0;     k1:=1;     for i:=1 to n do       begin         fillchar(opt[k1],sizeof(opt[k1]           ),0);         for j:=1 to n do</pre>	<pre>begin   opt[k1,j]:=max(opt[k0,j],opt[     k1,j-1]);   if a[i]=b[j] then     opt[k1,j]:=max(opt[k1,j],opt[       k0,j-1]+1);   end;   x:=k0;   k0:=k1;   k1:=x;   end;   writeln(n-opt[k0,n]); end; begin   main; end.</pre>
---	--	--

用这个方法 AC 了就该很高兴了，但不要停止思考的步伐，还有别的方法么？

从原问题出发，找这个问题的子问题。和上面说的最长公共子序列问题一样，设计序列的问题我们一般要考虑它的子序列，也就是更短的序列。

这样就回到了我第一节说的边界条件法了。

显然单独的字符就是边界了，而且单独的字符就是回文词，添加 0 个字符就可以了。

如果是两个字符组成的序列怎么办呢？

只要看他们是否相同就可以了，如果相同那就是回文词了，添加 0 个字符，如果不相同就在它的左边或右边添一个字符，让另外一个当对称轴。

如果是 3 个字符呢？

我们用 S 存这个序列，如果  $S[1]=S[3]$  那么它就是回文词了，如果  $S[1] \neq S[3]$ ，那么就在前面添 S[3] 或后面添 S[1]，剩下的就要考虑 S[1]S[2] 和 S[2]S[3] 这两个序列了。

通过前面的分析我们很容易想到这样的算法：

对于一个序列 S 只要看它的左右端的字符是否相同，如果相同那么就看除掉两端字符的新串要添的字符个数了；如果不同，就在它左面添上右端的字符，然后考虑去掉新序列两端的字符后的串要添的字符。或者在右面添上左端的字符，再考虑去掉添了字符后新串左右两端字符得到的新串要添的字符。

设计一个二维状态  $opt[L,i]$  表示长度是 L+1，起点是 i 的序列变成回文词要添的字符的个数。阶段就是字符的长度，决策要分类，即  $S[i]$  和  $S[i+L]$  是否相等。

状态转移方程：

$$opt[L,i] = \begin{cases} \min(opt[L-1,i]+1, opt[L-1,i+1]+1) & (s[i] \neq s[i+L]) \\ \min(opt[L-1,i]+1, opt[L-1,i+1]+1, opt[L-2,i+1]) & (s[i] = s[i+L]) \end{cases}$$

复杂度：空间复杂度=状态数  $O(N^2)$ ，时间复杂度=状态数  $O(N^2)$  \* 转移代价  $O(1) = O(N^2)$ 。由于空间复杂度较高，仍然要用滚动数组。

【源代码 2】

<pre>program P1327; const   maxn=5002; var   a:array[0..maxn] of char;   opt:array[0..2,0..maxn] of     longint;   n,ans:longint; function   min(x,y:longint):longint; begin</pre>	<pre>min:=y; if x&lt;y then min:=x; end; procedure main; var   i,L,j,k0,k1,k2:longint; begin   fillchar(opt,sizeof(opt),0);   readln(n);   for i:=1 to n do     read(a[i]);     k0:=0;</pre>	<pre>k1:=1; k2:=2; for L:=1 to n-1 do begin   for i:=1 to n-L do   begin     opt[k2,i]:=min(opt[k1,i],opt[       k1,i+1])+1;     if a[i]=a[i+L] then       opt[k2,i]:=min(opt[k2,i],opt[         k0,i+1]);     end;</pre>
--	--	---

j:=k0; k0:=k1; k1:=k2; k2:=j;	end; writeln(opt[k1,1]); end; begin	main; end.
--	--	---------------

### 例题 19 调整队形 (queue.pas/c/cpp) 来源: TJU P1006

#### 【问题描述】

学校艺术节上,规定合唱队要参加比赛,各个队员的衣服颜色不能很混乱:合唱队员应排成一横排,且衣服颜色必须是左右对称的。例如:“红蓝绿蓝红”或“红蓝绿绿蓝红”都是符合的,而“红蓝绿红”或“蓝绿蓝红”就不符合要求。

合唱队人数自然很多,仅现有的同学就可能会有 3000 个。老师希望将合唱队调整得符合要求,但想要调整尽量少,减少麻烦。以下任一动作认为是一次调整:

- 1、在队伍左或右边加一个人(衣服颜色依要求而定);
- 2、在队伍中任两个人中间插入一个人(衣服颜色依要求而定);
- 3、删掉一个人;
- 4、让一个人换衣服颜色;

老师想知道就目前的队形最少的调整次数是多少,请你编一个程序来回答他。

因为加入合唱队很热门,你可以认为人数是无限的,即随时想加一个人都能找到人。同时衣服颜色也是任意的。

#### 【输入文件】

第一行是一个整数  $n(1 \leq n \leq 3000)$ 。

第二行是  $n$  个整数,从左到右分别表示现有的每个队员衣服的颜色号,都是 1 到 3000 的整数。

#### 【输出文件】

一个数,即对于输入队列,要调整得符合要求,最少的调整次数。

#### 【输入样例】

```
5
1 2 2 4 3
```

#### 【输出样例】

```
2
```

#### 【提交链接】

[http://www.rqnoj.cn/Problem\\_478.html](http://www.rqnoj.cn/Problem_478.html)

#### 【问题分析】

读完题目发现很熟悉,仔细想想这个题就是回文词的加强版。不同于回文词的是这个问题的决策多了,不仅可以插入一个人(词),还可以删人,还可以换服装,其实删人和插入是等价的。也就是说比原问题只多了一个条件就是可以换服装。

这样就不能用回文词的第一个方法解了。(因为序列中的元素不固定,可以换)。只能用第二个方法解。

和回文词一样,阶段是序列的长度,状态是  $opt[i,j]$  表示  $[i,j]$  这段区间内要变成回文所需要的最少的调整次数。决策比回文词多了一个,即:如果左右两端不一样还可以通过换服装这种方式只花费一次的代价调整好。

状态转移方程:

$$opt[i,j] = \begin{cases} \min\{opt[i,j-1]+1, opt[i+1,j]+1, opt[i+1,j-1]+1\} & (a[i] \neq a[j], 1 \leq i < j \leq n) \\ \min\{opt[i,j-1]+1, opt[i+1,j]+1, opt[i+1,j-1]\} & (a[i] = a[j], 1 \leq i < j \leq n) \\ opt[i,i] = 0 & (1 \leq i \leq n) \text{ 边界条件} \end{cases}$$

时间复杂度: 状态数  $O(N^2)$  \* 转移代价  $O(1)$  = 总复杂度  $O(N^2)$ 。

#### 【源代码】

<pre>program queue; const fin='queue.in'; fout='queue.out'; maxn=3000; var a:array[0..maxn] of longint; opt:array[0..maxn,0..maxn] of longint; n:longint; procedure init; var i:longint; begin assign(input,fin);</pre>	<pre>reset(input); assign(output,fout); rewrite(output); readln(n); for i:=1 to n do read(a[i]); end; procedure main; var i,j,L:longint; begin fillchar(opt,sizeof(opt),0); for L:=1 to n-1 do for i:=1 to n-L do begin j:=i+L;</pre>	<pre>if opt[i+1,j]+1&lt;opt[i,j-1]+1 then opt[i,j]:=opt[i+1,j]+1 else opt[i,j]:=opt[i,j-1]+1; if a[i]=a[j] then begin if opt[i+1,j-1]&lt;opt[i,j] then opt[i,j]:=opt[i+1,j-1] end else begin if opt[i+1,j-1]+1&lt;opt[i,j] then opt[i,j]:=opt[i+1,j-1]+1; end; end; end; procedure print;</pre>
---	---	---

begin	end;	print;
writeln(opt[1,n]);	begin	end.
close(input);	init;	
close(output);	main;	

## 2.4 背包问题的拓展

前面说的背包问题还有个有趣的变形，可以说是背包问题的拓展吧，下面看一下这个例题：

**例题 20 找啊找啊找 GF** (gf.pas/c/cpp) 来源：MM 群 2007 七夕模拟赛 (RQNOJ 57)

### 【问题描述】

“找啊找啊找 GF，找到一个好 GF，吃顿饭啊拉拉手，你是我的好 GF，再见。”

“唉，别再见啊...”

七夕...七夕...七夕这个日子，对于 sqybi 这种单身的菜鸟来说是多么的痛苦...虽然他听着这首叫做“找啊找啊找 GF”的歌，他还是很痛苦。为了避免这种痛苦，sqybi 决定要给自己找点事情干。他去找到了七夕模拟赛的负责人 zmc MM，让她给自己一个出题的任务。经过几天的死缠烂打，zmc MM 终于同意了。

但是，拿到这个任务的 sqybi 发现，原来出题比单身更让人感到无聊-\_-....所以，他决定了，要在出题的同时去办另一件能够使自己不无聊的事情--给自己找 GF。

sqybi 现在看中了 n 个 MM，我们不妨把她们编号 1 到 n。请 MM 吃饭是要花钱的，我们假设请 i 号 MM 吃饭要花 rmb[i] 块大洋。而希望骗 MM 当自己 GF 是要费人品的，我们假设请第 i 号 MM 吃饭试图让她当自己 GF 的行为(不妨称作泡该 MM)要耗费 rp[i] 的人品。而对于每一个 MM 来说，sqybi 都有一个对应的搞定她的时间，对于第 i 个 MM 来说叫做 time[i]。sqybi 保证自己有足够的魅力用 time[i] 的时间搞定第 i 个 MM^\_^。

sqybi 希望搞到尽量多的 MM 当自己的 GF，这点是毋庸置疑的。但他不希望为此花费太多的时间(毕竟七夕赛的题目还没出)，所以 he 希望在保证搞到 MM 数量最多的情况下花费的总时间最少。

sqybi 现在有 m 块大洋，他也通过一段时间的努力攒到了 r 的人品(这次为模拟赛出题也攒 rp 哦~~)。他凭借这些大洋和人品可以泡到一些 MM。他想知道，自己泡到最多的 MM 花费的最少时间是多少。

注意 sqybi 在一个时刻只能去泡一个 MM--如果同时泡两个或以上的 MM 的话，她们会打起来的...

### 【输入文件】

输入的第一行是 n，表示 sqybi 看中的 MM 数量。

接下来有 n 行，依次表示编号为 1, 2, 3, ..., n 的一个 MM 的信息。每行表示一个 MM 的信息，有三个整数：rmb, rp 和 time。

最后一行有两个整数，分别为 m 和 r。

### 【输出文件】

你只需要输出一行，其中有一个整数，表示 sqybi 在保证 MM 数量的情况下花费的最少总时间是多少。

### 【输入样例】

```
4
1 2 5
2 1 6
2 2 2
2 2 3
5 5
```

### 【输出样例】

```
13
```

### 【数据规模】

对于 20% 数据， $1 \leq n \leq 10$ ;

对于 100% 数据， $1 \leq \text{rmb} \leq 100$ ,  $1 \leq \text{rp} \leq 100$ ,  $1 \leq \text{time} \leq 1000$ ;

对于 100% 数据， $1 \leq m \leq 100$ ,  $1 \leq r \leq 100$ ,  $1 \leq n \leq 100$ 。

### 【提交链接】

[http://www.rqnoj.cn/Problem\\_57.html](http://www.rqnoj.cn/Problem_57.html)

### 【问题分析】

初看问题觉得条件太多，理不出头绪来，所以要将问题简化，看能否找出熟悉的模型来，如果我们只考虑钱够不够，或只考虑 RP 够不够。并且不考虑花费的时间。这样原问题可以简化成下面的问题：

在给定 M 元 RMB (或 R 单位 RP, RP 该用什么单位呢？汗...) 的前提下，去泡足够多的 MM，很显然这个问题就是典型的 0/1 背包问题了。

可以把泡 MM 用的 RMB (或 RP 看做重量)，泡到 MM 的个数看做价值，给定的 M (或 R) 就是背包的载重。求解这个问题很轻松喽。

但是，这个问题既要考虑 RMB 又要考虑 RP 怎么办呢？

解决这个问题很容易啊，要是你有足够的 RMB 去泡第 i 个 MM 而 RP 不够就泡不成了，要是 RP 够就可以。也就是在原来问题的基础上，状态加一维。

那要是再考虑时间上最小怎么办呢？



这个也很好说，在求解过程中如果花  $X$  元 RMB,  $Y$  单位 RP 可以泡到  $Z$  个 MM, 那么在泡第  $i$  个 MM 时, 发现可以用  $X-rmb[i]$  元,  $Y-rp[i]$  单位 RP 泡到的 MM 数加上这个 MM (也就是+1) 比原来  $Z$  多, 就替换它 (因为你的原则是尽量多的泡 MM), 如果和  $Z$  一样多, 这时就要考虑原来花的时间多呢, 还是现在花的时间多。要是原来的多, 就把时间替换成现在用的时间 (因为你既然可以泡到相同数量的 MM 当然要省点时间去出题)。

设计一个二维状态  $opt[j,k]$  表示正好花  $j$  元 RMB,  $k$  单位 RP 可以泡到的最多的 MM 的数量。增加一个辅助的状态  $ct[j,k]$  表示正好花  $j$  元 RMB,  $k$  单位 RP 可以泡到的最多 MM 的情况下花费的最少的时间。

边界条件  $opt[0,0]=1$  (按题意应该是 0, 但为了标记花费是否正好设为 1, 这样,  $opt[j,k]>0$  说明花费正好)

状态转移方程:

$opt[j,k]=\max\{opt[j-rmb[i],k-rp[i]]+1\}$  ( $rmb[i]\leq j\leq m, rp[i]\leq k\leq r, 0<i\leq n, opt[j-rmb[i],k-rp[i]]>0$ )

$ct[j,k]:=\min\{ct[j-rmb[i],k-rp[i]]+time[i]\}$  ( $opt[j,k]=opt[j-rmb[i],k-rp[i]]+1$ )

时间复杂度: 阶段数  $O(N)$  \* 状态数  $O(MR)$  \* 转移代价  $O(1) = O(NMR)$ 。

注: 数据挺小的。

问题拓展:

如果要加入别的条件, 比如泡 MM 还要一定的 SP 等, 也就是说一个价值要不同的条件确定, 那么这个问题的状态就需要再加一维, 多一个条件就多一维。

【源代码】

```
program gf;
const
  fin='gf.in';
  fout='gf.out';
  maxn=110;
var
  rmb,rp,time:array[0..maxn]
of longint;
  opt,ct:array[0..maxn,0..maxn]
of longint;
  n,m,r,ans,max:longint;
procedure init;
var
  i:longint;
begin
  assign(input,fin);
  reset(input);
  assign(output,fout);
  rewrite(output);
  read(n);
  for i:=1 to n do
    read(rmb[i],rp[i],time[i]);
  read(m,r);
  close(input);
end;
```

```
procedure main;
var
  i,j,k:longint;
begin
  fillchar(opt,sizeof(opt),0);
  fillchar(ct,sizeof(ct),0);
  opt[0,0]:=1;
  for i:=1 to n do
    for j:=m downto rmb[i] do
      for k:=r downto rp[i] do
        if opt[j-rmb[i],k-rp[i]]>0 then
          begin
            if
              opt[j-rmb[i],k-rp[i]]+1>opt[j,k]
            then
              begin
                opt[j,k]:=opt[j-rmb[i],k-rp[i]]
                +1;
                ct[j,k]:=ct[j-rmb[i],k-rp[i]]+ti
                me[i];
              end
            else if
              (opt[j-rmb[i],k-rp[i]]+1=opt[j,k])
              and
              (ct[j-rmb[i],k-rp[i]]+time[i]<ct[j,k]
              ) then
```

```

              ct[j,k]:=ct[j-rmb[i],k-rp[i]]+ti
              me[i];
            end;
            max:=0;
            for j:=1 to m do
              for k:=1 to r do
                if opt[j,k]>max then
                  begin
                    max:=opt[j,k];
                    ans:=ct[j,k];
                  end
                else if (opt[j,k]=max) and
                  (ct[j,k]<ans) then
                    ans:=ct[j,k];
                  end;
            procedure print;
            begin
              writeln(ans);
              close(output);
            end;
            begin
              init;
              main;
              print;
            end.
```

### 例题 21 多多看 DVD (加强版) (watchdvd.pas/c/cpp) 来源: 本人原创

【问题描述】

多多进幼儿园了, 今天报名了。只有今晚可以好好放松一下了 (以后上了学后会很忙)。她的叔叔决定给他买一些动画片 DVD 晚上看。可是爷爷规定他们只能在一定的时间段  $L$  看完。(因为叔叔还要搞 NOIP 不能太早陪多多看碟, 而多多每天很早就困了所以只能在一定的时间段里看碟)。多多列出一张表要叔叔给她买  $N$  张 DVD 碟, 大多都是多多爱看的动画片 (福音战士, 机器猫, 火影忍者, 樱桃小丸子……)。这  $N$  张碟编号为  $(1, 2, 3, \dots, N)$ 。多多给每张碟都打了分  $M_i$  ( $M_i>0$ ), 打分越高的碟说明多多越爱看。每张碟有播放的时间  $T_i$ 。多多想在今晚爷爷规定的时间内看的碟总分最高。(必须把要看的碟看完, 也就是说一张碟不能只看一半)。显然叔叔在买碟时没必要把  $N$  张全买了, 只要买要看的就 OK 了, 这样节省资金啊。而且多多让叔叔惯的特别任性, 只要他看到有几张就一定看完。

可是出现了一个奇怪的问题, 买碟的地方只卖给顾客  $M$  ( $M<N$ ) 张碟, 不会多也不会少。这可让多多叔叔为难了。怎么可以在  $N$  张碟中只买  $M$  张而且在规定时间内看完, 而且使总价值最高呢?

聪明的你帮帮多多的叔叔吧。

【输入说明】(watchdvd.in)

输入文件有三行

第一行: 三个空格隔开的正整数,  $N, M, L$  (分别表示叔叔给多多买的碟的数量, 商店要卖给叔叔的



碟的数量，爷爷规定的看碟的时间段)。

第二行到第 N 行，每行两个数：T, M，给出多多列表中 DVD 碟的信息。

【输出说明】(watchdvd.out)

单独输出一行

表示多多今晚看的碟的总分。

如果商店卖给叔叔的 M 张碟无法在爷爷规定的时间看完输出 0。

【输入样例】

3 2 10

11 100

1 2

9 1

【输出样例】

3

【数据范围】

20%的数据  $N \leq 20$ ;  $L \leq 2000$ ;

100%的数据  $N \leq 100$   $L \leq 2000$ ;  $M < N$

【时限】

1S

【提交链接】

[http://www.rqnoj.cn/Problem\\_95.html](http://www.rqnoj.cn/Problem_95.html)

【问题分析】

这道题目是本人在学习背包问题时发现的一个拓展内容，所以就编了道题目，这道题比原问题多了一个条件：就是说不仅背包重量有限，连个数也有限。其实也可以说成在“找啊找啊找 GF”那道题目所说的增加了别的条件。

很多人这么想：那还不简单，在 DP 的过程中记录一下方案背包的个数，求解时只要个数等于 M 就 OK 了。

其实这个想法是错误的，因为由于 M 的不同，最优解就不同。对应不同的 M 有不同的最优解，也就是一个不限制 M 的较优解可能是限制了 M 以后的最优解。

正确的解法：在原来背包问题求解的基础上状态多加一维，表示不同的 M 对应的不同的最优解。

设计状态  $opt[i,j]$  表示背包载重是 j 时，选取物品限制 i 个的最优解。

状态转移方程：

$opt[i,j] = \max(opt[i-1,j-w[i]] + v[i])$

时间复杂度：阶段数  $O(N)$  \* 状态数  $O(LM)$  + 转移代价  $O(1) = O(NML)$ 。

【源代码】

```
program bb;
const
  fin='watchdvd.in';
  fout='watchdvd.out';
  maxn=100;
  maxL=1000;
var
  opt:array[0..maxn,0..maxL]
of longint;
  w,val:array[0..maxn] of
longint;
  n,m,v,ans:longint;
procedure init;
var
  i:longint;
begin
  assign(input,fin);
  reset(input);
  assign(output,fout);
```

```
  rewrite(output);
  readln(n,m,v);
  for i:=1 to n do
    read(w[i],val[i]);
  close(input);
end;
function
  max(x,y:longint):longint;
begin
  max:=y;
  if x>y then max:=x;
end;
procedure main;
var
  i,j,k:longint;
begin
  fillchar(opt,sizeof(opt),0);
  for i:=1 to n do
    for j:=m downto 1 do
      if j<=i then
```

```
        for k:=v downto w[i] do
          if (opt[j-1,k-w[i]]>0) or ((j=1)
and (k=w[i])) then
            opt[j,k]:=max(opt[j-1,k-w[i]]
+val[i],opt[j,k]);
          ans:=-maxlongint;
          for i:=0 to v do
            if opt[m,i]>ans then
              ans:=opt[m,i];
          end;
        procedure print;
        begin
          write(ans);
          close(output);
        end;
      begin
        init;
        main;
        print;
      end.
```

## 2.5 石子合并问题

也有人把这类问题叫做是区间上的动态规划。

**例题 22 石子合并** (stone.pas/c/cpp) 来源：某年 NOI（去巴蜀交）

【问题描述】

在一个操场上摆放着一行共 n 堆的石子。现要将石子有序地合并成一堆。规定每次只能选相邻的两堆

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。请编程计算出将  $n$  堆石子合并成一堆的最小得分和将  $n$  堆石子合并成一堆的最大得分。

【输入文件】

输入第一行为  $n(n < 1000)$ ，表示有  $n$  堆石子，第二行为  $n$  个用空格隔开的整数，依次表示这  $n$  堆石子的石子数量 ( $\leq 1000$ )

【输出文件】

输出将  $n$  堆石子合并成一堆的最小得分和将  $n$  堆石子合并成一堆的最大得分。

【输入样例】

3

1 2 3

【输出样例】

9 11

【提交链接】

[http://www.rqnoj.cn/Problem\\_490.html](http://www.rqnoj.cn/Problem_490.html)

【问题分析】

人们拿到这类题目的第一想法是贪心（找最大/最小的堆合并），但是很容易找到贪心的反例：（以求最小为例）

贪心：

3 4 6 5 4 2

5 4 6 5 4      得分：5

9 6 5 4      得分：9

9 6 9      得分：9

15 9      得分：15

24      得分：24

总得分：62

合理方案

3 4 6 5 4 2

7 6 5 4 2      得分：7

7 6 5 6      得分：6

7 11 6      得分：11

13 11      得分：13

24      得分：24

总得分：61

也就是说第二个 4 和 2 合并要比和 5 合并好，但是由于一开始 2 被用了没办法它只好和 5 合并了……

既然贪心不行，数据规模又很大，我们自然就想到用动态规划了。

\*不过要知道某个状态下合并后得分最优就需要知道合并它的总得分，只有对子问题的总得分进行最优的判断，设计的状态才有意义。

\*但要是想知道总分，就要知道合并一次的得分，显然这个含义不能加到动态规划的状态中，因为一般一个状态只代表一个含义（也就是说  $OPT[i]$  的值只能量化一个问题）（上面带\*号的两段比较抽象，不懂可以不看。我只是为了标注自己的理解加的，不理解的也没必要理解。）

先要把题目中的环变成链，这样好分析问题。具体方法：把环截断，复制一份放到截断后形成的链的后面形成一个长度是原来两倍的链（只有环中的元素在处理时不随着变化，就可以这样做。其实读入数据已经帮你截断了）；

例： 3 4 5

变成 3 4 5 3 4 5

对于这样一个链，我们设计一个状态  $opt[i,j]$  表示起点为  $i$  终点为  $j$  的链合并成一堆所得到的最优得分。

要合并一个区间里的石子无论合并的顺序如何它的得分都是这个区间内的所有石子的和，所以可以用一个数组  $sum[i]$  存合并前  $i$  个石子的得分。

因为合并是连续的所以决策就是把某次合并看作是把某个链分成两半，合并只想把两半的好多堆分别合并成一堆的总得分+最后合并这两半的得分。

状态转移方程：

$maxopt[i,j] = \max \{ maxopt[i,k] + maxopt[k+1,j] \} + sum[j] - sum[i-1]$

$minopt[i,j] = \min \{ minopt[i,k] + minopt[k+1,j] \} + sum[j] - sum[i-1]$

复杂度：状态数  $O(N^2)$  \* 决策数  $O(N) = O(N^3)$

【源代码】

program stone;

const

maxn=1010;

var

a,sum:array[0..maxn]

of

longint;

minopt,maxopt:array[0..max  
n\*2,0..maxn\*2] of longint;

n:longint;

minans,maxans:longint;

procedure init;

var

i:longint;

begin

read(n);

for i:=1 to n do

begin

<pre> read(a[i]); a[n+i]:=a[i]; end; for i:=1 to n*2 do sum[i]:=sum[i-1]+a[i]; end; function max(x,y:longint):longint; begin max:=y; if x&gt;y then max:=x; end; function min(x,y:longint):longint; begin min:=y; if (x&lt;y) and (x&gt;0) then min:=x; end; procedure main; var i,j,L,k:longint; </pre>	<pre> begin fillchar(minopt,sizeof(minopt ),200); fillchar(maxopt,sizeof(maxop t),0); for i:=1 to 2*n do minopt[i,i]:=0; for L:=1 to n-1 do for i:=1 to 2*n-L do begin j:=i+L; for k:=i to j-1 do begin maxopt[i,j]:=max(maxopt[i,j] ,maxopt[i,k]+maxopt[k+1,j]); minopt[i,j]:=min(minopt[i,j], minopt[i,k]+minopt[k+1,j]); end; inc(maxopt[i,j],sum[j]-sum[i- 1]); inc(minopt[i,j],sum[j]-sum[i- 1]); </pre>	<pre> end; maxans:=-maxlongint; minans:=maxlongint; for i:=1 to n do maxans:=max(maxans,maxo pt[i,i+n-1]); for i:=1 to n do minans:=min(minans,minopt[ i,i+n-1]); {for i:=1 to n*2 do begin for j:=1 to n*2 do write(maxopt[i,j],' '); writeln; end;} end; begin init; main; writeln(minans,' ',maxans); end. </pre>
--	---	--

### 例题 23 能量项链 (energy.pas/c/cpp) 来源 NOIP2006 (提高组)

#### 【问题描述】

在 Mars 星球上，每个 Mars 人都随身佩带着一串能量项链。在项链上有  $N$  颗能量珠。能量珠是一颗有头标记与尾标记的珠子，这些标记对应着某个正整数。并且，对于相邻的两颗珠子，前一颗珠子的尾标记一定等于后一颗珠子的头标记。因为只有这样，通过吸盘（吸盘是 Mars 人吸收能量的一种器官）的作用，这两颗珠子才能聚合成一颗珠子，同时释放出可以被吸盘吸收的能量。如果前一颗能量珠的头标记为  $m$ ，尾标记为  $r$ ，后一颗能量珠的头标记为  $r$ ，尾标记为  $n$ ，则聚合后释放的能量为（Mars 单位），新产生的珠子的头标记为  $m$ ，尾标记为  $n$ 。

需要时，Mars 人就用吸盘夹住相邻的两颗珠子，通过聚合得到能量，直到项链上只剩下一颗珠子为止。显然，不同的聚合顺序得到的总能量是不同的，请你设计一个聚合顺序，使一串项链释放出的总能量最大。

例如：设  $N=4$ ，4 颗珠子的头标记与尾标记依次为 (2, 3) (3, 5) (5, 10) (10, 2)。我们用记号  $\oplus$  表示两颗珠子的聚合操作，(j k) 表示第  $j$ ,  $k$  两颗珠子聚合后所释放的能量。则第 4、1 两颗珠子聚合后释放的能量为：(4 1)= $10*2*3=60$ 。

这一串项链可以得到最优值的一个聚合顺序所释放的总能量为：

((4 1) 2) 3) = $10*2*3+10*3*5+10*5*10=710$ 。

#### 【输入文件】

输入文件 energy.in 的第一行是一个正整数  $N$  ( $4 \leq N \leq 100$ )，表示项链上珠子的个数。第二行是  $N$  个用空格隔开的正整数，所有的数均不超过 1000。第  $i$  个数为第  $i$  颗珠子的头标记 ( $1 \leq i \leq N$ )，当  $i < N$  时，第  $i$  颗珠子的尾标记应该等于第  $i+1$  颗珠子的头标记。第  $N$  颗珠子的尾标记应该等于第 1 颗珠子的头标记。

至于珠子的顺序，你可以这样确定：将项链放到桌面上，不要出现交叉，随意指定第一颗珠子，然后按顺时针方向确定其他珠子的顺序。

#### 【输出文件】

输出文件 energy.out 只有一行，是一个正整数  $E$  ( $E \leq 2.1*10^9$ )，为一个最优聚合顺序所释放的总能量。

#### 【输入样例】

```

4
2 3 5 10

```

#### 【输出样例】

```

710

```

#### 【提交链接】

[http://www.rqnoj.cn/Problem\\_5.html](http://www.rqnoj.cn/Problem_5.html)

#### 【问题分析】

这道题应该算是本次考试的拿分题目，大多选手都做了这道题。可是大多数人都想简单了。其实它就是经典的石子合并的变形。

#### (1) 标准算法

这道题的考点应该是区间上的动态规划，思考这个问题之前先得解决项链的环状怎么处理。按照题意我们可以枚举切断点，把环状处理成链状。当然更好的方法是把环从任意一点切断，复制成两条链把这两条链首尾向接，针对题的读入我们直接把读入数据复制后连起来即可如：

2 3 5 10 ----->2 3 5 10 2 3 5 10

这样处理后其中任意长度为  $N+1$  的链就可代表一个环，那么问题就转化成合并任意长度为  $N+1$  的链所能释放的总能量最大。

也就是说从任意一点( $i < k < j$ )把链拆成两段问题的解就是合并这两段释放出最大能量在加上合并后这两颗珠子再一次合并释放的能量。将这个子问题进一步分解就是分解到链长度为 1 也就是就有两颗珠子时，生成这两颗珠子没有释放能量，而合并他们释放的能量是  $m * r * n$ 。(这就是边界条件)。

我们设计一个状态  $opt[i, j]$  表示合并头为  $i$ ，尾为  $j$  的链状项链所能释放的最多的能量值。边界条件是  $opt[i, i] = 0$  ( $1 \leq i \leq n * 2$ )。

根据定义不难得到动规的状态转移方程：

$opt[i, j] = \max\{opt[i, j], opt[i, k] + opt[k, j] + a[i] * a[k] * a[j]\} (i < k < j)$

(2) 复杂度：这个题有  $2N^2$  个状态，每个状态转移近似为  $N$  所以时间复杂度为  $O(N^3)$ ，由于  $N$  很小所以瞬间就可以出解。

#### 【源代码】

```
program energy;
const
  fin='energy.in';
  fout='energy.out';
  maxn=300;
var
  a:array[0..maxn] of longint;
  opt:array[0..maxn,0..maxn]
of longint;
  n,ans:longint;
procedure init;
var
  i:longint;
begin
  assign(input,fin);
  reset(input);
  assign(output,fout);
  rewrite(output);
  readln(n);
```

```
  for i:=1 to n do
  begin
    read(a[i]);
    a[n+i]:=a[i];
  end;
  close(input);
end;
function
  max(x,y:longint):longint;
begin
  if x>y then exit(x);
  exit(y);
end;
procedure main;
var
  i,j,k,L:longint;
begin
  fillchar(opt,sizeof(opt),0);
  for L:=2 to n do
    for i:=1 to n*2-L+1 do
```

```
  begin
    j:=i+L;
    for k:=i+1 to j-1 do
      opt[i,j]:=max(opt[i,j],opt[i,k]
+opt[k,j]+a[i]*a[j]*a[k]);
    end;
    for i:=1 to n do
      ans:=max(ans,opt[i,i+n]);
    end;
  procedure print;
  begin
    writeln(ans);
    close(output);
  end;
begin
  init;
  main;
  print;
end.
```

### 例题 24 统计单词个数 (.pas/c/cpp) 来源：NOIP2001（提高组）

#### 【问题描述】

给出一个长度不超过 200 的由小写英文字母组成的字母串(约定：该字母串以每行 20 个字母的方式输入，且保证每行一定为 20 个)。要求将此字母串分成  $k$  份( $1 < k \leq 40$ )，且每份中包含的单词个数加起来总数最大(每份中包含的单词可以部分重叠。当选用一个单词之后，其第一个字母不能再选。例如字符串 this 中可包含 this 和 is，选用 this 之后就不能包含 th)。

单词在给出的一个不超过 6 个单词的字典中。要求输出最大的个数。

#### 【输入文件】

输入数据放在文本文件 input3.dat 中，其格式如下：

每组的第一行有二个正整数( $p, k$ )， $p$  表示字串的行数； $k$  表示分为  $k$  个部分。

接下来的  $p$  行，每行均有 20 个字符。

再接下来有一个正整数  $s$ ，表示字典中单词个数。( $1 \leq s \leq 6$ )

接下来的  $s$  行，每行均有一个单词。

#### 【输出文件】

结果输出至屏幕，每行一个整数，分别对应每组测试数据的相应结果。

#### 【输入样例】

```
1 3
thisisabookyouareaoh
4
is
a
ok
sab
```

#### 【输出样例】

```
7
```

#### 【提交链接】

[http://www.rqnoj.cn/Problem\\_302.html](http://www.rqnoj.cn/Problem_302.html)

### 【问题分析】

刚看到这个题目觉得很迷茫，没入手点但是突然看到了闪亮的突破口：题目中说 this 包含 this 和 is 但不包含 th 这也就是说在一个串内对于一个固定了起点的单词只能用一次，即使他还可以构成别的单词但他还是用一次。比如：串：thisa

字典：this is th

串中有 this is th 这三个单词，但是对于 this 和 th 只用一次，也就是说枚举一下构成单词的起点，只要以该起点的串中包含可以构成一个以该起点开头的单词，那么就说明这个串中多包含一个单词。

这样可以得出下面的结果：

枚举的起点

结论：

t	至少包含 1 个
h	至少包含 1 个
i	至少包含 2 个
s	至少包含 2 个
a	至少包含 2 个

考虑到这里，就有点眉目了。

题目中要将串分 k 个部分也就是说从一个点截断后一个单词就未必可以构成了。比如上例要分 3 个部分，合理的其中的一个部分至多有 3 个字母，这样 this 这个单词就构不成了。

要是分 5 个部分，那就连一个单词都构不成了。

这样就需要对上面做个改动，上面的只控制了起点，而在题目中还需要限制终点，分完几个部分后，每部分终点不同可以构成的单词就不同了。

这样就需要再枚举终点了。

设计一个二维数组  $sum[i,j]$  统计从 i 到 j 的串中包含的单词的个数

状态转移方程：

$sum[i+1,j]+1$	(s[i,j] 中包含以 s[i] 开头的单词)
$sum[i,j]=sum[i+1,j]$	(与上面相反)

注：(1)这里枚举字符的起点的顺序是从尾到头的。

(2)有人把上面这次也看做是一次动态规划，但我觉得更准确的说是递推。

求出所有的 sum 还差一步，就是不同的划分方法显然结果是不一样的，但是对于求解的问题我们可以这样把原问题分解成子问题：求把一个串分成 k 部分的最多单词个数可以看做是先把串的最后一部分分出来，再把前面一部分分解成 k-1 个部分，显然决策就是找到一种划分的方法是前面的 k-1 部分的单词+最后一部分的单词最多。

显然这个问题满足最优化原理，那满不满足无后效性呢？

对于一个串分解出最后一部分在分解前面的那部分是根本就不会涉及分好的这部分，换句话说每次分解都会把串分解的更小，对于分解这个更小的串不会用到不属于这个小串的元素。这就满足无后效性。

具体求解过程：

设计一个状态  $opt[i,j]$  表示把从 1 到 j 的串分成 i 份可以得到最多的单词的个数。决策就是枚举分割点使当前这种分割方法可以获得最多的单词。

状态转移方程： $opt[i,j]=\max(opt[i-1,t]+sum[t+1,j])$  ( $i < t < j$ )

边界条件： $opt[1,i]=sum[1,i]$  ( $0 < i \leq L$ )

时间复杂度：状态数  $O(N^2)$  \* 决策数  $O(N) = O(N^3)$ ，空间复杂度： $O(N^2)$ 。

### 【源代码】

```
program P3;
const
  fin='input3.dat';
  fout='output3.dat';
  maxn=210;
var
  s,ss:string;
  opt,sum:array[0..maxn,0..maxn] of longint;
  a:array[0..maxn] of string;
  n,ii,P,k,L,nn:longint;
procedure init;
var
  i:longint;
begin
  readln(p,k);
  s:="";
  for i:=1 to p do
    begin
```

```
      readln(ss);
      s:=s+ss;
    end;
  readln(n);
  for i:=1 to n do
    readln(a[i]);
  end;
  function
    find(i,j:longint):boolean;
  var
    t:longint;
  begin
    for t:=1 to n do
      if pos(a[t],copy(s,i,j-i+1))=1
    then exit(true);
    find:=false;
  end;
  function
    max(x,y:longint):longint;
  begin
```

```
    max:=y;
    if x>y then max:=x;
  end;
procedure main;
var
  i,j,t:longint;
begin
  L:=length(s);
  for i:=L downto 1 do
    for j:=i to L do
      if find(i,j) then
        sum[i,j]:=sum[i+1,j]+1
      else sum[i,j]:=sum[i+1,j];
    fillchar(opt,sizeof(opt),0);
    opt[1]:=sum[1];
    for i:=2 to k do
      for j:=i+1 to L do
        for t:=i+1 to j-1 do
          opt[i,j]:=max(opt[i,j],opt[i-1,t]
        ]+sum[t+1,j]);
```



```
writeln(opt[k,L]);
end;
begin
assign(input,fin);
```

```
reset(input);
assign(output,fout);
rewrite(output);
init;
```

```
main;
close(input);
close(output);
end.
```

## 2.6 其他问题

还有一些题目虽和一写基本模型相似但又有区别，我也就不总结共性了，列出它们，看看他们的状态又是怎么设计的：

**例题 25 花店橱窗设计** (flower.pas/c/cpp) 来源：IOI 或巴蜀评测系统

【问题描述】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，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 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： $1 \leq F \leq 100$ ， $F \leq V \leq 100$ ， $-50 \leq A_{ij} \leq 50$ ，其中  $A_{ij}$  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

【输入文件】

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

【输出文件】

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。

【输入样例】

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

【输出样例】

```
53
2 4 5
```

【题目链接】

<http://poj.org/problem?id=1157>

[http://www.rqnoj.cn/Problem\\_496.html](http://www.rqnoj.cn/Problem_496.html)

【问题分析】

这个问题很奇怪题目中给定的条件是花瓶和花束，似乎是两个没有关联的事物啊，但着两个看似没关联的东西，却有一点联系：不同的花放在不同的花瓶中产生不同的美学价值。

一般人的思维都是拿来花一个一个的放，假设要放第  $i$  束花时，把它放到哪里好呢？

很容易想到一个贪心的策略：找到一个符合条件（第  $i$  束花要放在前  $i-1$  束花的后面）下的它放如后能产生最大的美学价值的花瓶放。但和容易举出反例：

	花瓶 1	花瓶 2	花瓶 3
--	------	------	------

杜鹃花	1	2	-5
秋海棠	5	10	1

按照贪心策略是：杜鹃花放在 2 号瓶里，秋海棠放在 3 号瓶里，美学值：3

答案是：杜鹃花放在 1 号瓶里，秋海棠放在 2 号瓶里，美学值：11

数据量很大搜索显然不行。

那要是动态规划，阶段，状态，决策有是什么呢？

既然要拿来花束一个一个的放，我们就以花束划分阶段。设计一个状态  $opt[i,j]$  表示将第  $i$  束花放在第  $j$  个花瓶中可使前  $i$  束花或得的最大美学价值，那么决策就很容易想到了：将第  $i$  束花放在第  $j$  个瓶中，那么第  $i-1$  束花只能放在前  $j-1$  个瓶里，显然我们要找到一个放在前  $j-1$  个瓶中的一个最大的美学价值在加上当前第  $i$  束放在第  $j$  个瓶中的美学价值就是  $OPT[I, J]$  的值。

显然符合最优优化原理和无后效性。

状态转移方程：

$opt[i,j]=\max\{opt[i-1,k]\}+a[i,j]$  ( $i \leq k \leq j-1$ ) 为什么是  $i \leq$  呢？

复杂度：状态数  $O(FV) * 转移代价 O(V) = O(FV^2)$

数据范围很小，可以在瞬间出解。

回顾刚才的解题过程，贪心和动态规划都是找前一阶段的最大值，为什么贪心是错的，而动态规划是对的？

着就要读者自己反思，总结了。

【源代码】

```
const
  maxn=110;
var
  a,opt,path:array[0..maxn,0..maxn] of longint;
  n,m,ans:longint;
procedure init;
var
  i,j:longint;
begin
  read(n,m);
  for i:=1 to n do
    for j:=1 to m do
      read(a[i,j]);
  end;
procedure main;
var
  i,j,k:longint;
begin
  for i:=1 to n do
```

```
    for j:=1 to m do
      opt[i,j]:=-maxlongint;
      for i:=1 to n do
        for j:=i to m-n+i do
          begin
            for k:=i-1 to j-1 do
              if opt[i-1,k]>opt[i,j] then
                begin
                  opt[i,j]:=opt[i-1,k];
                  path[i,j]:=k;
                end;
            end;
            ans:=n;
            for i:=n+1 to m do
              if opt[n,i]>opt[n,ans] then
                ans:=i;
            end;
          procedure
            outputway(i,j:longint);
          begin
```

```
            if i>0 then
              begin
                outputway(i-1,path[i,j]);
                write(j,' ');
              end;
            end;
          procedure print;
          var
            i:longint;
          begin
            writeln(opt[n,ans]);
            outputway(n,ans);
            writeln;
          end;
          begin
            init;
            main;
            print;
          end.
```

**例题 26 Divisibility** 来源：ZJU2042

【问题描述】

Consider an arbitrary sequence of integers. One can place + or - operators between integers in the sequence, thus deriving different arithmetical expressions that evaluate to different values. Let us, for example, take the sequence: 17, 5, -21, 15. There are eight possible expressions:

```
17 + 5 + -21 + 15 = 16
17 + 5 + -21 - 15 = -14
17 + 5 - -21 + 15 = 58
17 + 5 - -21 - 15 = 28
17 - 5 + -21 + 15 = 6
17 - 5 + -21 - 15 = -24
17 - 5 - -21 + 15 = 48
17 - 5 - -21 - 15 = 18
```

We call the sequence of integers divisible by  $K$  if + or - operators can be placed between integers in the sequence in such way that resulting value is divisible by  $K$ . In the above example, the sequence is divisible by 7 ( $17+5+-21-15=-14$ ) but is not divisible by 5.

You are to write a program that will determine divisibility of sequence of integers.

译题：

给出  $N$  个数，你可以在这  $N$  个数中任意地添加+号或-号，求解能不能使算出的结果被  $K$  整除。可以

则打印 “Divisible”，否则打印 “Not divisible”。( $1 \leq N \leq 10000$ ,  $2 \leq K \leq 100$ )

下面是一个例子：有 4 个数，分别是 17 5 -21 15，有 8 种加法，其中第二种求出的 -14 能被 7 整除。

$$17 + 5 + -21 + 15 = 16$$

$$17 + 5 + -21 - 15 = -14$$

$$17 + 5 - -21 + 15 = 58$$

$$17 + 5 - -21 - 15 = 28$$

$$17 - 5 + -21 + 15 = 6$$

$$17 - 5 + -21 - 15 = -24$$

$$17 - 5 - -21 + 15 = 48$$

$$17 - 5 - -21 - 15 = 18$$

【输入文件】

The first line of the input contains two integers, N and K ( $1 \leq N \leq 10000$ ,  $2 \leq K \leq 100$ ) separated by a space.

The second line contains a sequence of N integers separated by spaces. Each integer is not greater than 10000 by its absolute value.

注意第一个数是测试数据的组数，多组测试数据一起测。

【输出文件】

Write to the output file the word "Divisible" if given sequence of integers is divisible by K or "Not divisible" if it's not.

The first line of a multiple input is an integer N, then a blank line followed by N input blocks. Each input block is in the format indicated in the problem description. There is a blank line between input blocks.

The output format consists of N output blocks. There is a blank line between output blocks.

注意：输出每组结果之间有空格，最后一行无空格，格式不对不能 AC，我就是因为格式不对调了一上午...

【输入样例】

2

4 7

17 5 -21 15

4 5

17 5 -21 15

【输出样例】

Divisible

Not divisible

【提交链接】

<http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=2042>

<http://poj.org/problem?id=1745>

【问题分析】

看到题目第一个反映就是枚举中间添的运算符，算出值在 MOD K 如果有一个值 MOD K=0 则输出 “Divisible”。

时间复杂度是  $O(2N-1)$ 。

但是题目给出的数据量很大，这样做效率太低了。因为题目涉及 MOD 运算，要想简化问题就需要知道一些基本的 MOD 运算性质：

$$A*B \bmod C = (A \bmod C * B \bmod C) \bmod C$$

$$(A+B) \bmod C = (A \bmod C + B \bmod C) \bmod C$$

有了这个性质，我们就可以把累加后求余转化成求余后累加（我们把减法看作加负数以后分析只说加法）再求余。这样我们的读入数据就控制在了 1-K 到 K-1 的范围内了。

我们要判断的就是：所有结果的累加和 MOD K 是否为 0。简记为：

$$(A+B) \bmod K = 0 \text{ or } (A+B) \bmod K <> 0$$

如果我们按数的个数划分阶段，前 N-1 个数的运算结果 MOD K 看做 A，第 N 个数看作 B 就 OK 了。于是我们想到了这样的状态：opt[i,j] 表示前 i 个数是否可以得到余数为 j 的结果。

那么状态转移方程就是：

$$\text{opt}[i, (j-a[i] \bmod k) \bmod k] = \text{opt}[i-1, j] \quad (\text{opt}[i-1, j] = \text{true});$$

$$\text{opt}[i, (j+a[i] \bmod k) \bmod k] = \text{opt}[i-1, j] \quad (\text{opt}[i-1, j] = \text{true});$$

如果 opt[n,0]=true 就输出 ‘Divisible’。

【源代码】

program P2042;

const

maxk=110;

maxn=10010;

var

a:array[0..maxn] of longint;  
opt:array[1..2,-maxk..maxk]  
of boolean;  
n,k,tim,ii:longint;  
vis:array[0..maxn] of  
boolean;

procedure init;  
var  
i:longint;  
begin  
read(n,k);  
for i:=1 to n do

<pre> read(a[i]); end; procedure main; var i,j,p1,p2,p3:longint; begin fillchar(opt,sizeof(opt),false); fillchar(vis,sizeof(vis),false); for i:=1 to n do if a[i] mod k=0 then vis[i]:=true; for i:=1 to n do a[i]:=a[i] mod k; opt[1,a[1]]:=true; p1:=1; </pre>	<pre> p2:=2; for i:=2 to n do if not vis[i] then begin fillchar(opt[p2],sizeof(opt[p2 ]),false); for j:=1-k to k-1 do if opt[p1,j] then begin opt[p2,(j-a[i]) mod k]:=true; opt[p2,(j+a[i]) mod k]:=true; end; p3:=p1; p1:=p2; p2:=p3; </pre>	<pre> end; if opt[p1,0] then writeln('Divisible') else writeln('Not divisible'); end; begin read(tim); for ii:=1 to tim do begin if ii&gt;1 then writeln; init; main; end; end. </pre>
--	---	--

### 3.多维状态和动态规划的优化

一般多维动态规划的时，空间复杂度较高，所以我们要想办法将其优化，我就把多维动态规划和动态规划的优化放到一起了……

多维动态规划以三，四维最为常见，在多的也没有太大的研究价值，其实多维动态规划大多也就是上面的一维，和二维的加一些条件，或是在多进程动态规划中要用到。当然除了这些特点外，状态的表示也有一些共性。

三维：状态  $opt[i,j,k]$  一般可表示下面的含义：

(1) 二维状态的基础上加了某个条件，或其中一维变成两个。

比如  $opt[L,i]$  表示起点为  $I$ ，长度为  $L$  的序列的最优值。 $opt[L,i,j]$  就可表示起点是  $i$  和起点是  $j$ ，长度是  $L$  的两个序列的最优值。

(2)  $I, J, K$  组合表示一个正方形 ( $(i,j)$  点为一角，边长为  $K$ )。

(3)  $I, J, K$  组合表示一个等边三角形 ( $(i,j)$  点为一角，边长为  $K$ )。

四维：除了二维和三维加条件外，还可以用  $i,j,k,t$  组合来描述一个矩形，

$(i,j)$  点和  $(k,t)$  点是两个对顶点。

四维以上的一般都是前几维加了条件了，这里就不多说了。

动态规划的优化：

动态规划的优化往往需要较强的数学功底。

常见空间优化：

滚动数组，滚动数组在前面也提到过，其实很简单，如果一个状态的决策的步长为  $N$  就只保留以求出的最后  $N$  (一般  $N=1$ ) 个阶段的状态，因为当前状态只和后  $N$  个阶段中的状态有关，再以前的已经利用过了，没用了就可以替换掉了。具体实现是最好只让下标滚动 (这样更省时间)。

$X := K1, K1 := K2, K2 := K3, K3 := X$  这样就实现了一个  $N=3$  的下标的滚动，在滚动完如果状态是涉及累加，累乘类的操作要注意将当前要求的状态初始化。

常见时间优化：利用一些数据结构 (堆，并查集，HASH) 降低查找复杂度。

时间空间双重优化：改变状态的表示法，降低状态维数。

具体的多维动态规划和动态规划的优化，我们从题目里体会吧！

#### 3.1 矩阵问题

先看一道题

**例题 27 盖房子** 来源：Vijos P1057

【问题描述】

永恒の灵魂最近得到了面积为  $n*m$  的一大块土地 (高兴 ING^\_^)，他想在这块土地上建造一所房子，这个房子必须是正方形的。但是，这块土地并非十全十美，上面有很多不平坦的地方 (也可以叫瑕疵)。这些瑕疵十分恶心，以至于根本不能在上面盖一砖一瓦。他希望找到一块最大的正方形无瑕疵土地来盖房子。不过，这并不是什么难题，永恒の灵魂在 10 分钟内就轻松解决了这个问题。现在，您也来试试吧。

【输入文件】

输入文件第一行为两个整数  $n,m$  ( $1 \leq n,m \leq 1000$ )，接下来  $n$  行，每行  $m$  个数字，用空格隔开。0 表示该块土地有瑕疵，1 表示该块土地完好。

【输出文件】

一个整数，最大正方形的边长。

【输入样例】

4 4

0 1 1 1

1 1 1 0

0 1 1 0

1 1 0 1

【输出样例】

2

【提交链接】

[http://www.vijos.cn/Problem\\_Show.asp?id=1057](http://www.vijos.cn/Problem_Show.asp?id=1057)

【问题分析】

题目中说要求一个最大的符合条件的正方形，所以就想到判断所有的正方形是否合法。

这个题目直观的状态表示法是  $opt[i,j,k]$  基类型是 `boolean`，判断以  $(i,j)$  点为左上角（其实任意一个角都可以，依据个人习惯），长度为  $K$  的正方形是否合理，再找到一个  $K$  值最大的合法状态就可以了（用 `true` 表示合理，`false` 表示不合理）。其实这就是递推，（决策唯一）。

递推式：

$opt[i,j,k] = opt[i+1,j+1,k-1] \text{ and } opt[i+1,j,k-1] \text{ and } opt[i,j+1,k-1] \text{ and } (a[i,j]=1)$

时间复杂度：

状态数  $O(N^3)$  \* 转移代价  $O(1)$  = 总复杂度  $O(N^3)$

空间复杂度：

$O(N^3)$

由于空间复杂度和时间复杂度都太高，不能 AC，我们就再想想怎么优化？

显然何以用滚动数组优化空间，但是时间复杂度仍然是  $O(N^3)$ 。这就需要我们找另外一种简单的状态表示法来解了。

仔细分析这个题目，其实我们没必要知道正方形的所有长度，只要知道以一个点为左上角的正方形的最大合理长度就可以了。

如果这个左上角是 0 那么它的最大合理长度自然就是 0（不可能合理）。

如果这个左上角是 1 呢？

回顾上面的递推式，我们考虑的是以它的右面，下面，右下这个三个方向的正方形是否合理，所以我们还是要考虑这三个方向。具体怎么考虑呢？

如果这三个方向合理的最大边长中一个最小的是  $X$ ，那么它的最大合理边长就是  $X+1$ 。为什么呢？

看个例子：

0 1 1 1 1 1

1 1 1 1 1 1

0 1 0 1 1 0

1 1 0 1 1 1

上例中红色的正方形，以  $(1,3)$  点为左上角，以  $(1,4)$ ， $(2,3)$ ， $(2,4)$  这三个点的最大合理边长分别是 2，1，2。其中最小的是以  $(2,3)$  为左上角的正方形，最大合理边长是 1。因为三个方向的最大合理边长大于等于 1，所以三个方向上边长为 1 的正方形是合理的，即上面递推式中：

$opt[1,3,2] = opt[1,4,1] \text{ and } opt[2,3,1] \text{ and } opt[2,4,1] \text{ and } (a[1,3]=1) = \text{true}$  成立

这样就把一个低推判定性问题转化成最优化问题从而节省空间和时间。

具体实现：

设计一个状态  $opt[i,j]$  表示以  $(i,j)$  为左上角的正方形的最大合理边长。

状态转移方程：

$\min\{opt[i+1,j], opt[i,j+1], opt[i+1,j+1]\} + 1 \quad (a[i,j]=1)$

$opt[i,j] = 0 \quad (a[i,j]=0)$

时间复杂度：状态数  $O(N^2)$  \* 转移代价  $O(1)$  = 总代价  $O(N^2)$

空间复杂度：  $O(N^2)$

【源代码】

```
program P1057;
const
  maxn=1010;
var
  opt,a:array[0..maxn,0..maxn]
of longint;
  n,m,ans:longint;
procedure init;
var
  i,j:longint;
begin
  read(n,m);
```

```
  for i:=1 to n do
  for j:=1 to m do
    read(a[i,j]);
  end;
procedure main;
var
  i,j:longint;
begin
  fillchar(opt,sizeof(opt),0);
  for i:=n downto 1 do
  for j:=m downto 1 do
    if a[i,j]<>0 then
      begin
```

```
        opt[i,j]:=opt[i+1,j];
        if opt[i,j+1]<opt[i,j] then
          opt[i,j]:=opt[i,j+1];
        if opt[i+1,j+1]<opt[i,j] then
          opt[i,j]:=opt[i+1,j+1];
        inc(opt[i,j]);
      end;
  ans:=0;
  for i:=1 to n do
  for j:=1 to m do
    if opt[i,j]>ans then
      ans:=opt[i,j];
  writeln(ans);
```



```
end;  
begin
```

```
init;  
main;
```

```
end.
```

### 3.2 多进程动态规划

从字面上就可以看出，所谓多进程就是在原文题的基础上要求将这个问题重复多次的总和最大。具体怎么做看个例题吧。

**例题 28 方格取数** (fgqs.pas/c/cpp) 来源：NOIP2000(提高组)

#### 【问题描述】

设有  $N \times N$  的方格图 ( $N \leq 10$ ), 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):

		→ 向右								
	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	↓
向 下										B

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。

#### 【输入文件】

输入的第一行为一个整数  $N$  (表示  $N \times N$  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。

#### 【输出文件】

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
```

#### 【提交链接】

[http://www.rqnoj.cn/Problem\\_314.html](http://www.rqnoj.cn/Problem_314.html)

#### 【问题分析】

这是一道很经典的多进程动态规划题, 如果只取一次, 它的模型就是我们前面讲的街道问题了, 很简单就可以实现。现在要求取两次, 怎么办呢?

一个自然的想法是: 将前面的取过的数全赋成 0, 然后在取一次, 感觉挺对的, 样例也过了。

但这样做是错的, 第一次取的显然是最大值, 但第二次取的未必次大, 所以也许两条非最大的路, 也许比一条最大一条小一点的路更优。

看个例子:

002030	002030
002000	002000
002020	002020
000020	000020
000020	000020
002020	002020

图 1

图 2

如上图，图 1 是按找上诉思路求得的解。图中红色路线是第一求得的最大值，显然图 1 红色和紫色两条路径不如图 2 蓝色和绿色两条路径大。

既然这样做不行，我们还得回到动态规划的本质来看代问题，我们在想想这个问题的状态，对于走一次，走到矩阵的任意一个位置就是一个状态，而要是走两次，显然走到矩阵的某个位置只是一个状态的一部分，不能完整的描述整个状态。那另一部分显然就是第二次走到的位置了。如果我们把这两部分合起来就是一个完整的状态了。

于是，设计一个状态  $opt[i1,j1,i2,j2]$  表示两条路分别走到  $(i1,j1)$  点和  $(i2,j2)$  点时取到的最大值。显然决策有 4 中（乘法原理一个点两种\*另一个点的两中）

即（上，上）（上，左）（左，上）（左，左）上和左表示从哪个方向走到该点，当然要注意走到同行，同列，同点时的情况（因为要求路径不重复）。

状态转移方程：

$$\max(opt[i1-1,j1,i2-1,j2],opt[i1,j1-1,i2-1,j2])+a[i1,j1]+a[i2,j2] \quad (1 \leq i1=i2 \leq n, 1 \leq j1 \leq j2 \leq n)$$

$$\max(opt[i1-1,j1,i2,j2-1],opt[i1,j1-1,i2,j2-1])$$

$$(1 \leq j1=j2 \leq n, 1 \leq i1 \leq i2 \leq n)$$

$$opt[i,j] = \max(opt[i1-1,j1,i2-1,j2],opt[i1-1,j1,i2,j2-1],opt[i1,j1-1,i2-1,j2],$$

$$opt[i1,j1-1,i2,j2-2])+a[i1,j1]+a[i2,j2] \quad (1 \leq i1,j1 \leq i2,j2 \leq n)$$

$$\max(opt[i1-1,j1,i2-1,j2],opt[i1-1,j1,i2,j2-1],opt[i1,j1-1,i2-1,j2],$$

$$opt[i1,j1-1,i2,j2-2])+a[i1,j1] \quad (1 \leq i1=i2 \leq n, 1 \leq j1=j2 \leq n)$$

时间复杂度：状态数  $O(N^4)$  \* 转移代价  $O(1)$  = 总复杂度  $O(N^4)$

空间复杂度： $O(N^4)$

由于数据很小所以这样做虽然时间和空间复杂度都很高但还是可以 AC 的。

【源代码 1】

<pre> program fgqs; const fin='fgqs.in'; fout='fgqs.out'; maxn=11; var a:array[0..maxn,0..maxn] of longint; opt:array[0..maxn,0..maxn,0.. maxn,0..maxn] of longint; n:longint; procedure init; var i,j,w:longint; begin assign(input,fin); reset(input); assign(output,fout); rewrite(output); read(n); repeat readln(i,j,w); a[i,j]:=w; until i=0; close(input); </pre>	<pre> end; function max(x,y:longint):longint; begin max:=y; if x&gt;y then max:=x; end; procedure main; var i1,i2,j1,j2:longint; begin for i1:=1 to n do for j1:=1 to n do for i2:=i1 to n do for j2:=1 to j1 do if (i1=i2) and (j1=j2) then opt[i1,j1,i2,j2]:=opt[i1-1,j1,i2 ,j2-1]+a[i1,j1] else if (i1=i2-1) and (j1=j2) then opt[i1,j1,i2,j2]:=max(opt[i1-1 ,j1,i2,j2-1],opt[i1,j1-1,i2,j2-1]) +a[i1,j1]+a[i2,j2] else if (i1=i2) and (j1=j2+1) then </pre>	<pre> opt[i1,j1,i2,j2]:=max(opt[i1-1 ,j1,i2,j2-1],opt[i1-1,j1,i2-1,j2]) +a[i1,j1]+a[i2,j2] else begin opt[i1,j1,i2,j2]:=max(opt[i1-1 ,j1,i2,j2-1],opt[i1-1,j1,i2-1,j2]); opt[i1,j1,i2,j2]:=max(opt[i1,j 1,i2,j2],opt[i1,j1-1,i2,j2-1]); opt[i1,j1,i2,j2]:=max(opt[i1,j 1,i2,j2],opt[i1,j1-1,i2-1,j2]); inc(opt[i1,j1,i2,j2],a[i1,j1]+a[ i2,j2]); end; end; procedure print; begin writeln(opt[n,n,n,n]); close(output); end; begin init; main; print; end. </pre>
--	---	---

如果这个题的数据范围在大点就得优化了，怎么优化这个程序呢？

上面说过对于时间空间都大的时候，首先想到的就是寻找特点，改变状态的表示法，减少状态的维数。

仔细分析我们发现，处于同行，同列的状态，等价于另外一个点在对角线上的状态。而这条对角线正是此题的阶段。因为在状态转移的时候后面的哪个点总是从固定的一个方向转移来的。也就是说我们只要对角先上的状态就可以省掉那些同行同列的状态了。

做过 N 皇的同学一定知道怎么表示右上到左下的这几条对角线，不知道的同学也没关系，对于一个点  $(i,j)$  他对角右上角的点就是  $(i-1, j+1)$  所以可以看出这些点的和是定值，且值从 2 到  $N*2$ 。

这样用三个变量就可以表示这两个点了，于是设计状态  $opt[k,i1,i2]$  表示处于阶段 K 时走到  $i1,i2$  的两条路径所取得的数的最大和。

用上面的思维不难想出动态转移方程：

$$\max(opt[k-1,i1-1,i2-1],opt[k-1,i1-1,i2],opt[k-1,i1,i2-1],opt[k-1,i1,i2])+a[i1,k-i1]+a[i2,k-i2] \quad (1 \leq i1,i2 \leq n, 2 \leq k \leq n*2, i1 \leq i2)$$

opt[k,i1,i2]=opt[k-1,i1-1,i2]+a[i1,k-i1](1<=i1,i2<=n,2<=k<=n\*2,i1=i2)

#### 【源代码 2】

```
program fgqs;
const
  fin='fgqs.in';
  fout='fgqs.out';
  maxn=11;
var
  a:array[0..maxn,0..maxn] of
    longint;
  opt:array[0..maxn*2,0..maxn,
    0..maxn] of longint;
  n:longint;
procedure init;
var
  i,j,w:longint;
begin
  assign(input,fin);
  reset(input);
  assign(output,fout);
  rewrite(output);
  read(n);
  repeat
    readln(i,j,w);
    a[i,j]:=w;
```

```
until i=0;
close(input);
end;
function
  max(x,y:longint):longint;
begin
  max:=y;
  if x>y then max:=x;
end;
procedure main;
var
  k,i1,i2,j1,j2,mid:longint;
begin
  for k:=2 to n*2 do
    begin
      for i1:=1 to n do
        if (k-i1>0) and (k-i1<=n)
        then
          for i2:=1 to n do
            if (k-i2>0) and (k-i2<=n)
            then
              begin
                if i1=i2 then
                  opt[k,i1,i2]:=opt[k-1,i1-1,i2]
```

```
+a[i1,k-i1]
                else begin
                  opt[k,i1,i2]:=max(opt[k-1,i1,i
                    2],opt[k-1,i1,i2-1]);
                  opt[k,i1,i2]:=max(opt[k,i1,i2]
                    ,opt[k-1,i1-1,i2]);
                  opt[k,i1,i2]:=max(opt[k,i1,i2]
                    ,opt[k-1,i1-1,i2-1]);
                  inc(opt[k,i1,i2],a[i1,k-i1]+a[i
                    2,k-i2]);
                end;
              end;
            end;
          end;
        end;
      procedure print;
      begin
        writeln(opt[n*2,n,n]);
        close(output);
      end;
    end;
  begin
    init;
    main;
    print;
  end.
```

注：如果取多次，和取两次道理一样，只不过有多一次，状态就多加一维，如果数据范围很大，时间空间复杂度太高时可以用网络流解这个问题，只是本人才疏学浅不能和大家分享了。

## 4.树型动态规划

由于树型动态规划比较特殊，要借助树这种数据结构，所以很多地方都把它独立看做一类，我也不例外。

一般树型动态规划都是建立在二叉树上的，对于普通的树和森林要把它们转化成二叉树，以前认为数据结构中将的转化没什么用处，现在终于用到了。

树型动态规划是动态规划类中最好分析的，因为树本来就是一个低归的结构，阶段，状态，决策都很明显，子问题就是子树的最优值，也很明显。

较其他类型的动态规划不同的是，求解树型动态规划的第一步一般要先建立数据结构，考虑问题的数据结构是二叉树呢？还是多叉树，还是森林……

其他类型的动态规划一般都是用逆向递推实现，而树型动态规划一般要正向求解，为了保证时间效率要加记忆化（即长说的记忆化搜索）。

树型动态规划的三要素：

阶段：树的层数

状态：树中的结点

决策：某个子数的最优，或所有子树的最优和，或某几个子树的最优

通过上面的决策，发现如果是一棵多叉树，针对求某几个（>=1）子树的最优解，决策会很多。以至于我们没发写出准确的状态转移方程。这就是我们为什么要把多叉数转化成二叉数的原因。（当然，如果问题是求所有子树的和，就没必要转化了，如 URAL P1039 没有上司的舞会）。

如果把多叉树或森林转化成二叉树要注意，左儿子和根结点是父子关系，而右儿子在原问题中和跟结点是兄弟关系。（这个数据结构掌握扎实的应该能明白，不理解的就去看数据结构方面的知识）。

用邻接矩阵存多叉树，转化成二叉树的部分代码（可能有语法错误）

G: 存图，F[i] 表示第 i 个结点的儿子应该插入的位置

W: 结点的值 BT: 二叉树

Procedure creat\_tree(T:tree);

Var

i:longint;

Begin

for i:=1 to n do

Begin

```

for j:=1 to n do
  If g[i,j] then
    Begin
      If F[i]=0 then
        BT[i].L:=j
      Else BT[F[i]].r:=j;
      F[i]:=j;
    End;
  End;
End;

```

下面同过例题看一下树型动态规划的具体实现：

### 例题 29 加分二叉树 (binary.pas/c/cpp) 来源：NOIP2003（提高组）

#### 【问题描述】

设一个  $n$  个节点的二叉树  $tree$  的中序遍历为  $(1,2,3,\dots,n)$ ，其中数字  $1,2,3,\dots,n$  为节点编号。每个节点都有一个分数（均为正整数），记第  $i$  个节点的分数为  $di$ ， $tree$  及它的每个子树都有一个加分，任一棵子树  $subtree$ （也包含  $tree$  本身）的加分计算方法如下：

$subtree$  的左子树的加分  $\times$   $subtree$  的右子树的加分  $+$   $subtree$  的根节点的分数

若某个子树为空，规定其加分为 1，叶子的加分就是叶节点本身的分数。不考虑它的空子树。

试求一棵符合中序遍历为  $(1,2,3,\dots,n)$  且加分最高的二叉树  $tree$ 。要求输出：

(1)  $tree$  的最高加分

(2)  $tree$  的前序遍历

#### 【输入格式】

第 1 行：一个整数  $n$  ( $n < 30$ )，为节点个数。

第 2 行： $n$  个用空格隔开的整数，为每个节点的分数（分数  $< 100$ ）。

#### 【输出格式】

第 1 行：一个整数，为最高加分（结果不会超过 4,000,000,000）。

第 2 行： $n$  个用空格隔开的整数，为该树的前序遍历。

#### 【输入样例】

```

5
5 7 1 2 10

```

#### 【输出样例】

```

145
3 1 2 4 5

```

#### 【提交链接】

[http://www.rqnoj.cn/Problem\\_49.html](http://www.rqnoj.cn/Problem_49.html)

#### 【问题分析】

根据题目描述就可以看出是典型的树型动态规划，而且题目中已经给出了加分的求法：

$subtree$  的左子树的加分  $\times$   $subtree$  的右子树的加分  $+$   $subtree$  的根节点的分数

这估计是最简单的题目了。

但是有一点需要注意：我们应该怎么建树？

其实这个题不用建树，这就需要对数据结构掌握的很扎实。

题目中说这个树的中序遍历是  $1, 2, 3, \dots, N$ ，我们要求的是树的先序，这样马上就想到怎样用中序和先序确定一棵树。

枚举树根  $i$  那么  $1, 2, \dots, i-1$  就是它的左子树中的结点， $i+1, \dots, N$  就是它右子树中的结点。这样一颗树按这样的递归定义就构造出来了，（当然我们只要这个过程并不需要存储这棵树）。在构造过程中顺便求出加分来，在一个序列里不同的元素做根显然加分不同，我们只要记录一个最大的就可以了。

具体实现方法：

设计状态  $opt[L,r]$  表示以  $L$  为起点，以  $r$  为终点的结点所组成的树的最高加分，阶段就是树的层数。决策就是在这些结点中找一个结点做根使树的加分最大，状态转移方程：

1  $(L > r)$

$opt[L,r] = a[L] \quad (L = r)$

$\max\{opt[L,i-1]*opt[i+1,r]+a[i]\} \quad (L < r, L \leq i \leq r)$

在保存最优解的过程用  $path[i,j]$  记录以  $i$  为起点，以  $j$  为终点的中序结点中的根就可以了。

由于树型动态规划的阶段不明显所以一般用记忆化搜索好实现。

时间复杂度：状态数  $O(N^2)$  \* 转移代价  $O(N) = O(N^3)$

#### 【源代码】

```

program binary;
const

```

```

  fin='binary.in';
  fout='binary.out';
  maxn=100;

```

```

var
  a:array[0..maxn] of longint;
  path,opt:array[0..maxn,0..ma

```

<pre> xn] of longint; n:longint; procedure init; var i:longint; begin assign(input,fin); reset(input); assign(output,fout); rewrite(output); read(n); for i:=1 to n do read(a[i]); close(input); fillchar(opt,sizeof(opt),0); end; function TreeDp(L,r:longint):longint; var i,x:longint; begin </pre>	<pre> if opt[L,r]=0 then begin if L&gt;r then begin opt[L,r]:=1; path[L,r]:=0; end else if L=r then begin opt[L,r]:=a[L]; path[L,r]:=L; end else begin for i:=L to r do begin x:=TreeDp(L,i-1)*TreeDp(i+ 1,r)+a[i]; if x&gt;opt[L,r] then begin opt[L,r]:=x; path[L,r]:=i; </pre>	<pre> end; end; end; end; TreeDp:=opt[L,r]; end; procedure print(L,r:longint); begin if path[L,r]&gt;0 then begin write(path[L,r],' '); print(L,path[L,r]-1); print(path[L,r]+1,r); end; end; begin init; writeln(TreeDp(1,n)); print(1,n); close(output); end. </pre>
--	---	--

### 例题 30 A Binary Apple Tree 苹果二叉树 来源: URAL P1018

#### 【问题描述】

设想苹果树很象二叉树，每一枝都是生出两个分支。我们用自然数来数这些枝和根那么必须区分不同的枝（结点），假定树根编号都是定为 1，并且所用的自然数为 1 到 N。N 为所有根和枝的总数。例如下图的 N 为 5，它是有 4 条枝的树。

```

2 5
 \ /
3 4
 \ /
1

```

当一棵树太多枝条时，采摘苹果是不方便的，这就是为什么有些枝要剪掉的原因。现在我们关心的是，剪枝时，如何令到损失的苹果最少。给定苹果树上每条枝的苹果数目，及必须保留的树枝的数目。你的任务是计算剪枝后，能保留多少苹果。

#### 【输入文件】

首行为 N, Q ( $1 \leq Q \leq N$ ,  $1 < N \leq 100$ ), N 为一棵树上的根和枝的编号总数, Q 为要保留的树枝的数目。以下 N-1 行为每条树枝的描述, 用 3 个空格隔开的整数表示, 前 2 个数为树枝两端的编号, 第三个数为该枝上的苹果数。假设每条枝上的苹果数不超过 3000 个。

#### 【输出文件】

输出能保留的苹果数。(剪枝时, 千万不要连根拔起哦)

#### 【输入样例】

```

5 2
1 3 1
1 4 10
2 3 20
3 5 20

```

#### 【输出样例】

```

21

```

#### 【问题分析】

和上一道题一样，题目描述就很明确的说是关于树的题目，在加上是求最优值，和上一道题不同的是，这个题目边有值，并非点有值，还有一个不同点就是这个题需要建树。

建树是要注意，给每个结点增加一个域：SUM 用来存以它为根的树中的边的数目。

其实树型动态规划是最好分析的，因为树这种数据结构本来就符合递归定义，这样的话子问题很好找，显然这个问题的子问题就是一棵树要保留 M 个枝分三种情况：

剪掉左子树：让右子树保留 M-1 个枝可保留最多的苹果数+连接右子树的枝上的苹果数

剪掉右子树：让左子树保留 M-1 个枝可保留最多的苹果数+连接左子树的枝上的苹果数

都不剪掉：让左子树保留 i 个枝，让右子树保留 M-2-i 个枝可保留最多的苹果数+连接右子树的枝上的苹果数+连接左子树的枝上的苹果数

显然边界条件就是如果要保留的枝子数比当前的子树的枝多，或着一个树要保留 0 个枝子，则结果就



是 0。

应为一颗树中根接点是对子树的完美总结，所以满足最优化原理。

没次求解只和子树有关所以也满足无后效性，可见用动态规划做是正确的。

设计一个状态  $opt[num, i]$  表示要让结点编号为  $i$  的树保留  $num$  个枝可得到的最优解。

状态转移方程：

$opt[num, i] = \max \{ opt[num-1, BT[i].L] + T[i, BT[i].L], opt[num-1, BT[i].r] + T[i, BT[i].r], opt[k, BT[i].L] + opt[num-2-k, BT[i].r] + T[i, BT[i].L] + T[i, BT[i].r] \}$

( $0 \leq k \leq n-2$ , BT: 树, T: 读入时记录的枝上的苹果数);

时间复杂度: 状态数  $O(NM)$  \* 转移代价  $O(M) = O(NM^2)$

#### 【源代码】

```
program P1018;
const
  fin='P1018.in';
  fout='P1018.out';
  maxn=110;
type
  treetype=record
    l,r,sum:longint;
  end;
  var
    T,opt:array[0..maxn,0..maxn]
  of longint;
  BT:array[0..maxn]
  of treetype;
  vis:array[0..maxn]
  of boolean;
  n,m,p:longint;
procedure init;
var
  i,j,k,w:longint;
begin
  assign(input,fin);
  reset(input);
  assign(output,fout);
  rewrite(output);
  fillchar(T,sizeof(T),0);
  read(n,m);
  for k:=1 to n-1 do
  begin
```

```
    read(i,j,w);
    T[i,j]:=w;
    T[j,i]:=w;
  end;
  close(input);
  fillchar(vis,sizeof(vis),false);
end;
Procedure
  creat_tree(i:longint);
var
  j:longint;
begin
  vis[i]:=true;
  for j:=1 to n do
    if (T[i,j]>0) and (not vis[j])
  then
    begin
      creat_tree(j);
      BT[i].L:=Bt[i].r;
      Bt[i].r:=j;
      inc(Bt[i].sum,BT[j].sum+1);
    end;
  end;
Function
  max(x,y:longint):longint;
begin
  max:=y;
  if x>y then max:=x;
end;
Function
```

```
  F(num,i:longint):longint;
var
  k:longint;
begin
  if opt[num,i]<0 then
  begin
    if (num>BT[i].sum) or
    (num=0) then opt[num,i]:=0
    else begin
      opt[num,i]:=F(num-1,BT[i].L
    )+T[i,BT[i].L];
      opt[num,i]:=max(opt[num,i],
    F(num-1,BT[i].r)+T[i,BT[i].r]);
      for k:=0 to num-2 do
        opt[num,i]:=max(opt[num,i],
    F(k,BT[i].L)+F(num-2-k,BT[i].r
    )+T[i,BT[i].L]+T[i,BT[i].r]);
    end;
  end;
  F:=opt[num,i];
end;
begin
  init;
  creat_tree(1);
  fillchar(opt,sizeof(opt),200);
  writeln(F(m,1));
  close(output);
end.
```

# 动态规划经典问题

刘汝佳

# 目录

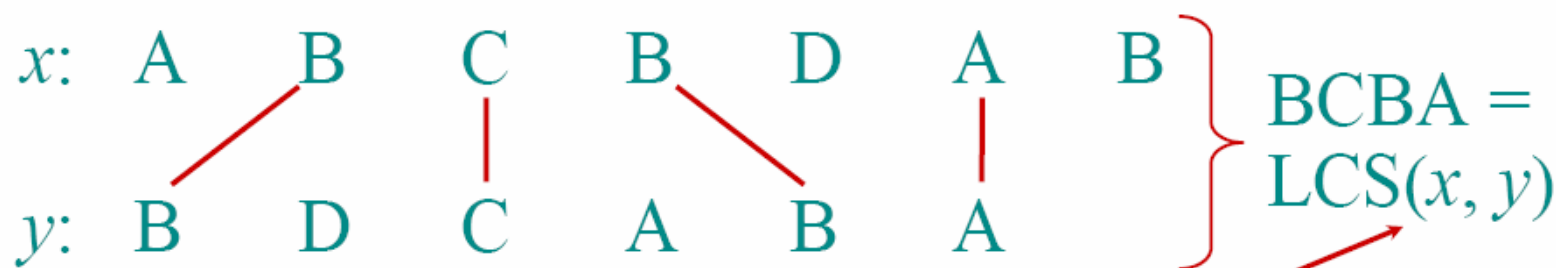
- 一、最长公共子序列 $O(mn)$
- 二、最优排序二叉树 $O(n^3)$
- 三、最长上升子序列 $O(n\log n)$
- 四、最优三角剖分 $O(n^3)$
- 五、最大 $m$ 子段和 $O(mn)$
- 六、0-1背包问题 $O(\min\{nc, 2^n, n1.44^n\})$
- 七、最优排序二叉树 $O(n^2)$
- 八、最优合并问题 $O(n\log n)$

# 一、最长公共子序列

- **Longest Common Subsequence(LCS)**

- Given two sequences  $x[1 \dots m]$  and  $y[1 \dots n]$ , find a longest subsequence common to them both.

“a” *not* “the”



functional notation,  
but not a function

# 分析

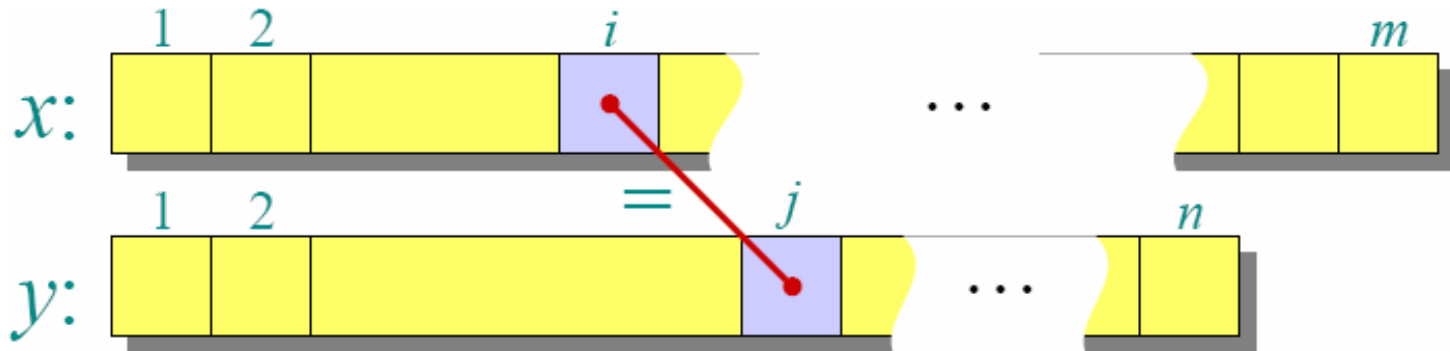
- 考虑前缀 $x[1..i]$ 和 $y[1..j]$ , 定义

$$c[i,j] = |\text{LCS}(x[1..i], y[1..j])|$$

- 则 $c[m,n] = |\text{LCS}(x, y)|$ . 递推公式为

$$c[i,j] = \begin{cases} c[i-1,j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise.} \end{cases}$$

- 很直观. 考虑 $x[i]=y[j]$ 的情形:





# 关键点一：最优子结构

- 为了使用动态规划, 问题需具备最优子结构 (**Optimal Substructure**)

## ***Optimal substructure***

*An optimal solution to a problem (instance) contains optimal solutions to subproblems.*

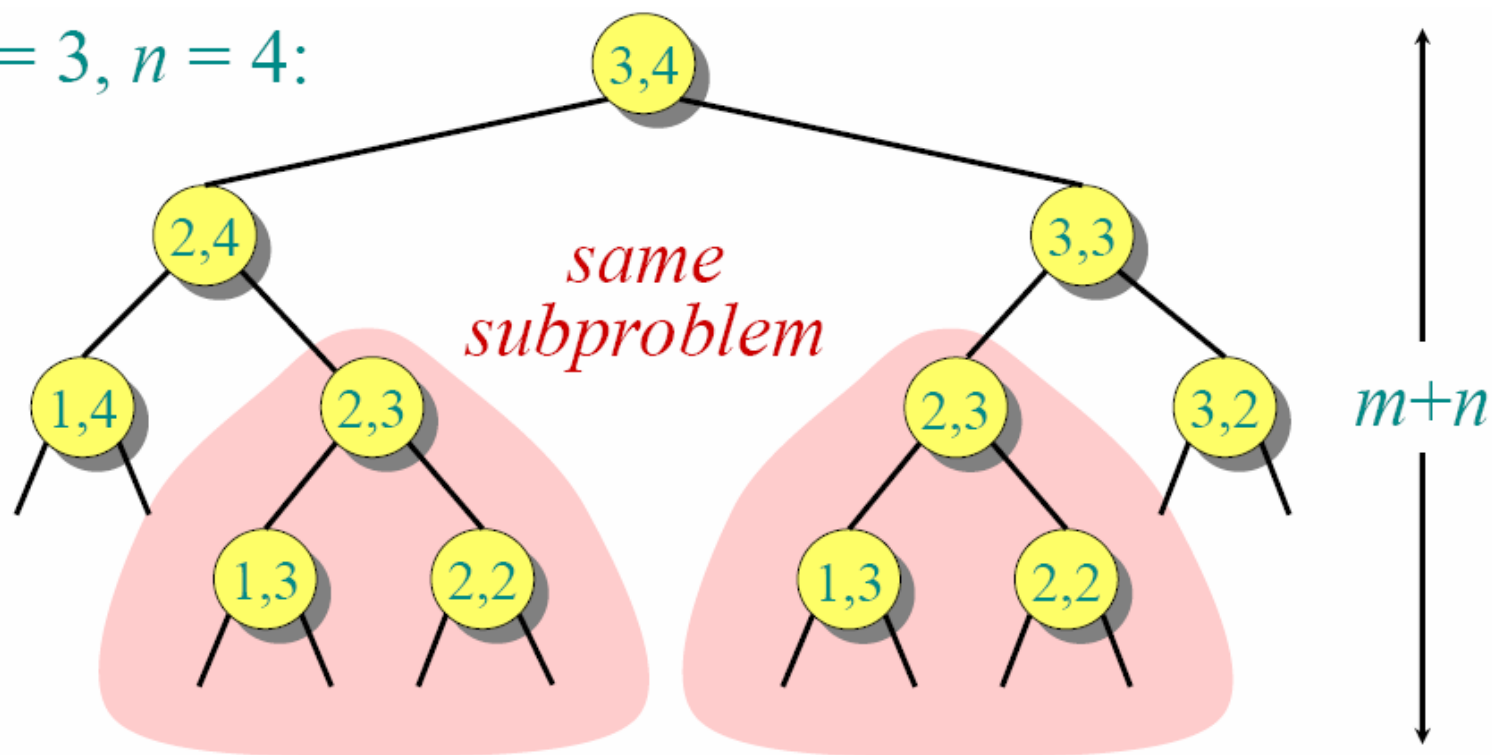
If  $z = \text{LCS}(x, y)$ , then any prefix of  $z$  is an LCS of a prefix of  $x$  and a prefix of  $y$ .

# 直接书写的程序

```
LCS( $x, y, i, j$ )  
  if  $x[i] = y[j]$   
    then  $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$   
    else  $c[i, j] \leftarrow \max \{ \text{LCS}(x, y, i-1, j),$   
                                    $\text{LCS}(x, y, i, j-1) \}$ 
```

# 递归树分析

$m = 3, n = 4$ :



Height =  $m + n \Rightarrow$  work potentially exponential,  
but we're solving subproblems already solved!

## 关键点二：重叠子问题

- 为了让动态规划确实发挥功效, 问题应该包含尽量多的重叠子问题(**overlapping subproblems**)

### *Overlapping subproblems*

*A recursive solution contains a “small” number of distinct subproblems repeated many times.*

The number of distinct LCS subproblems for two strings of lengths  $m$  and  $n$  is only  $mn$ .

# 解决方法: 记忆化

- 注意memoization不是memorization

***Memoization:*** After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

```
LCS( $x, y, i, j$ )  
  if  $c[i, j] = \text{NIL}$   
    then if  $x[i] = y[j]$   
      then  $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$   
      else  $c[i, j] \leftarrow \max \{ \text{LCS}(x, y, i-1, j),$   
                                 $\text{LCS}(x, y, i, j-1) \}$ 
```

*} same as before*

Time =  $\Theta(mn)$  = constant work per table entry.  
Space =  $\Theta(mn)$ .



# 自底向上递推

## IDEA:

Compute the table bottom-up.

Time =  $\Theta(mn)$ .

Reconstruct LCS by tracing backwards.

Space =  $\Theta(mn)$ .

Exercise:

$O(\min\{m, n\})$ .

	A	B	C	B	D	A	B
B	0	0	0	0	0	0	0
D	0	0	1	1	1	2	2
C	0	0	1	2	2	2	2
A	0	1	1	2	2	3	3
B	0	1	2	2	3	3	4
A	0	1	2	2	3	4	4

# 空间优化

- 如果只需要最优值, 可以用滚动数组实现
- 按照 $i$ 递增的顺序计算,  $d[i,j]$ 只和 $d[i-1,j]$ 和 $d[i,j-1]$ 以及 $d[i-1,j-1]$ 有关系, 因此只需要保留相邻两行, 空间复杂度为 $O(\min\{m,n\})$
- 更进一步的, 可以只保留一行, 每次用单独的变量 $x$ 保留 $d[i-1,j]$ , 则递推方程为

**if( $i==j$ )  $d[j]=x$ ;**

**else {  $x = d[j]$ ;  $d[j]=\max\{d[j-1], d[j]\}$  };**

# 变形. 回文词

- 给一个字符串**a**, 保持原字符的顺序不变, 至少要加几个字符才能变成回文词?
- 例: **abfcbfa** → **afb****c****fc****bfa**

# 分析

- 红、绿色表示原字符, 白色为新增字符
- 显然, **s**和**s'**在任何一个位置不可能都是白色(不需要加那个字符!)
- 应该让红色字符尽量多! 相当于求**s**和逆序串**s'**的**LCS**, 让**LCS**中的对应字符(红色)对齐, 中间的每个绿色字符都增加一个字符和它相等

1		2	3		4	5
5	4		3	2		1

## 二、最优排序二叉树

- 给 $n$ 个关键码和它们的频率，构造让期望比较次数最小的排序二叉树



# 分析

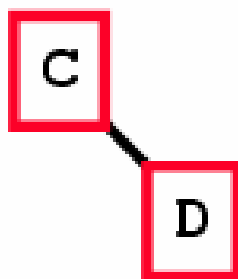
- 定理：最优排序二叉树的子树也是最优排序二叉树
- 给出关键码-频率对照表（升序排列）
- 问题：把哪个关键码做为根？则左右子树可以递归往下做

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	..
23	10	8	12	30	5	14	18	20	2	4	11	7	22	22	10	..

# 分析

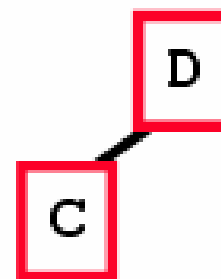
- 用递归来思考，但用递推来做
- 先考虑两个结点的情形

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	..
23	10	8	12	30	5	14	18	20	2	4	11	7	22	22	10	..



**Cost**

$$8 \times 1 + 12 \times 2 = 32$$



$$8 \times 2 + 12 \times 1 = 28$$

Min

# 分析

- 可以用矩阵来保存结果
- $C[j,k]$ 表示从j到k的关键码组成的最优排序二叉树
- $Root[j,k]$ 记录这棵排序二叉树的根

	A	B	C	D	E	F	G	H	I	J	K
A	23										
B	43	10									
C		26	8								
D			28	12							
E				54	30						
F					40	5					
G						24	14				
H							46	18			
I								56	20		
J									24	2	

[illegible]

# 分析

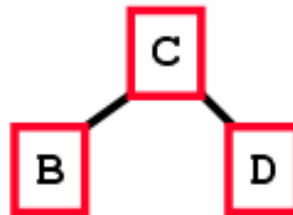
- 考虑三个结点的情形
- 最优值放在 $C[B,D]$ 中，根放在 $root[B,D]$ 中

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	..
23	10	8	12	30	5	14	18	20	2	4	11	7	22	22	10	..

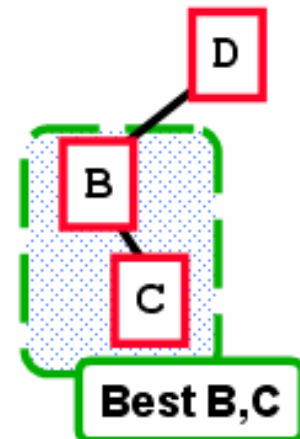
★ Root = B



★ Root = C



★ Root = D



# 分析

- 类似地，更新所有 $C[j-2,j]$ 和 $root[j-2,j]$

	A	B	C	D	E	F	G	H	I	J	K
A	23										
B	43	10									
C	67	26	8								
D		52	28	12							
E			78	54	30						
F				64	40	5					
G					73	24	14				
H						60	46	18			
I							86	56	20		
J								60	24	2	

**Costs**

	A	B	C	D	E	F	G	H	I	J	K
A	0										
B	0	1									
C	0	1	2								
D		2	3	3							
E			4	4	4						
F				4	4	5					
G					4	6	6				
H						6	7	7			
I							7	8	8		
J								8	8	9	

**Roots**



# 分析

- 四个结点的情形（如A-D）

- **Choose A as root**

**Use 0 for left**  
**Best B-D is known**

- **Choose B as root**

**A-A is in C[0,0]**  
**Best C-D is known**

- **Choose C as root**


**A-B is in C[0,1]**  
**D is in C[3,3]**

- **Choose D as root**

**A-C is in C[0,2]**  
**Use 0 in C[4,3] for right**

# 分析

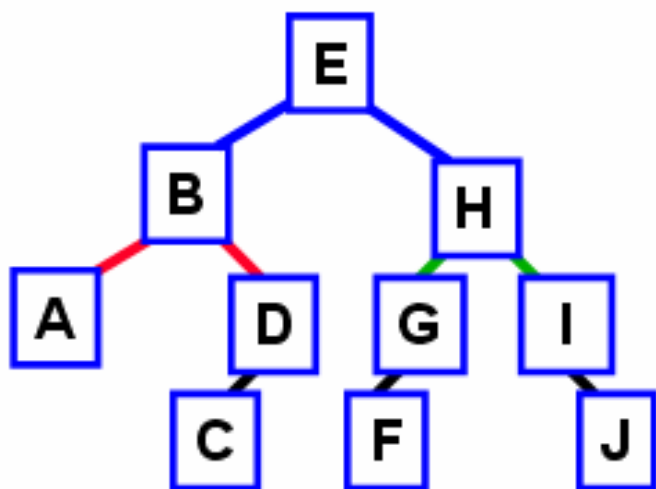
- 最终计算结果为



	A	B	C	D	E	F	G	H	I	J
A	23									
B	43	10								
C	67	26	8							
D	104	52	28	12						
E	180	112	78	54	30					
F	195	122	88	64	40	5				
G	230	155	121	97	73	24	14			
H	284	209	175	151	125	60	46	18		
I	345	270	236	212	180	101	86	56	20	
J	353	278	244	220	186	107	92	60	24	2

# 分析

- 可以利用root矩阵递归地构造出最优树



	A	B	C	D	E	F	G	H	I	J
A	0									
B	0	1								
C	0	1	2							
D	1	2	3	3						
E	2	4	4	4	4					
F	2	4	4	4	4	5				
G	4	4	4	4	4	6	6			
H	4	4	4	4	6	6	7	7		
I	4	4	4	4	7	7	7	8	8	
J	4	4	4	4	7	7	8	8	9	

# 分析

- 时间复杂度：计算每个 $C[i,j]$ 和 $root[i,j]$ 需要枚举根结点，故为 $O(n^3)$
- 空间复杂度：需要两个 $n*n$ 矩阵， $O(n^2)$

### 三、最长上升子序列

- 最长上升子序列问题 (**LIS**) 给一个序列, 求它的一个递增子序列, 使它的元素个数尽量多。例如序列**1,6,2,5,4,7**的最长上升子序列是**1,2,5,7** (还有其他的, 这里略去)

# 分析

- 定义 $d[i]$ 是从第1个元素到第 $i$ 个元素为止的最长子序列长度, 则状态转移方程为

$$d[i] = \min_{k < i \text{ 且 } a[k] < a[i]} \{d[k] + 1\}$$

- 直接使用这个方程得到的是 $O(n^2)$ 算法
- 下面把它优化到 $O(n \log n)$



# 状态的组织

- d值相同的a值只需要保留最小的, 因此用数组g[i]表示d值为i的数的a最小值, 显然

$$g[1] \leq g[2] \leq \dots \leq g[k]$$

- **计算d[i]:** 需要在g中找到大于等于a[i]的第一个数j, 则d[i]=j
- **更新g:** 由于g[j]>a[i], 需要更新g[j]=a[i]

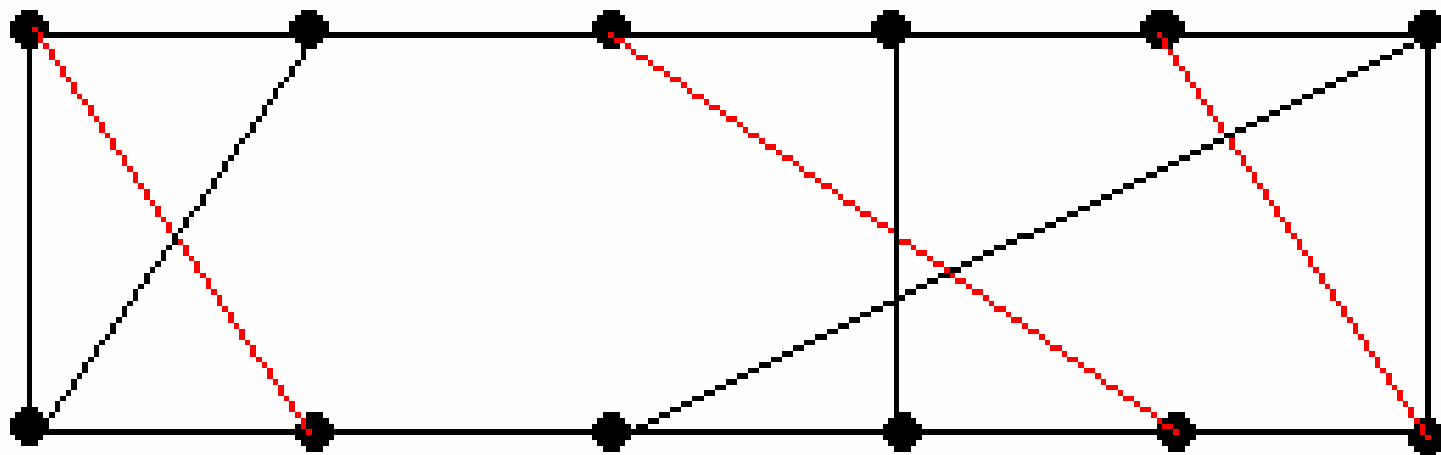
# 代码

- 使用STL的lower\_bound可以直接求出比a[i]大的第一个数, 用二分查找实现, 每次转移时间 $O(\log n)$ , 总时间 $O(n \log n)$

```
fill(g, g + n, infinity);  
for(int i = 0; i < n; i++){  
    int j = lower_bound(g, g + n, a[i]) - g;  
    d[i] = j + 1;  
    g[j] = a[i];  
}
```

# 变形1: 航线问题

- 有两行点, 每行 $n$ 个. 第一行点和第二行点是一一对应的, 有线连接, 如下图所示
- 选择尽量多的线, 两两不交叉



# 分析

- 设与第1行第 $i$ 个点对应的是第2行第 $f[i]$ 个点
- 假设 $i < j$ , 两条线 $(i, f[i])$ 和 $(j, f[j])$ 的充要条件是 $f[i] < f[j]$ , 因此问题变成了

求 $f$ 的最长上升子序列

- 时间复杂度为 $O(n \log n)$

# 变形2: 两排列的LCS

- 给 $1 \sim n$ 的两个排列 $p_1, p_2$
- 求 $p_1$ 和 $p_2$ 的最长公共子序列
- 例: **1 5 3 2 4**  $\Leftrightarrow$  **5 3 4 2 1**

# 分析

- 算法一: 直接套用LCS算法, 时间 $O(n^2)$
- 算法二: 注意到把两个排列做相同的置换, LCS不变, 可以先把 $p_1$ 排列为 $1, 2, 3, \dots, n$

**1 5 3 2 4**  $\Leftrightarrow$  **1 2 3 4 5**

- 即映射 $5 \rightarrow 2, 2 \rightarrow 4, 4 \rightarrow 5$ ,  $p_2$ 作同样置换

**5 3 4 2 1**  $\Leftrightarrow$  **2 3 5 4 1**

- 与 $1, 2, 3, \dots, n$ 的LCS显然是最长上升子序列, 时间降为 $O(n \log n)$

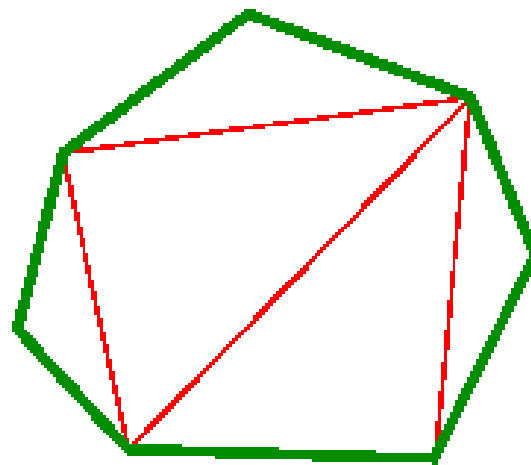
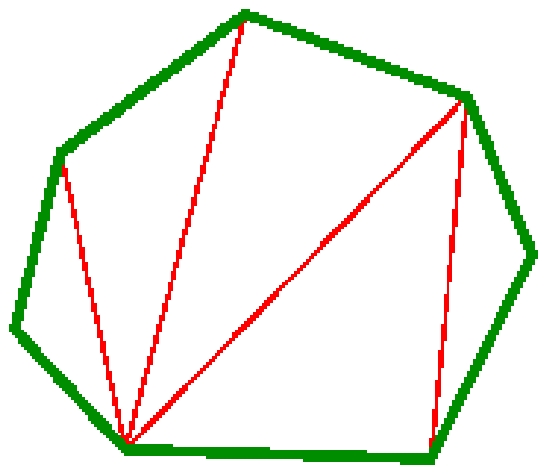
# 推广: DAG上的最短路

- “上升”依赖于序关系 $\leq$ , 它具有一般性
- **DAG**(有向无环图)的最长路径问题: 把有向边看成偏序关系, 则本题的算法一仍然适用, 时间复杂度为 $O(n^2)$ . 如果图的边数 $m$ 比较, 可进一步优化到 $O(m)$ , 因为每条边恰好考虑一次(用邻接表或前向星, 而不是邻接矩阵)
- 算法二不再适用! 因为 $d$ 值相同的 $a$ 不一定可以两两相互比较, 不一定存在最小值.



## 四、最优三角剖分

- 给一个 $n$ 个顶点的凸多边形, 有很多方法对它进行三角剖分(polygon triangulation)
- 每个三角形有一个权计算公式(如周长, 顶点权和), 求总权最小(大)的三角剖分方案

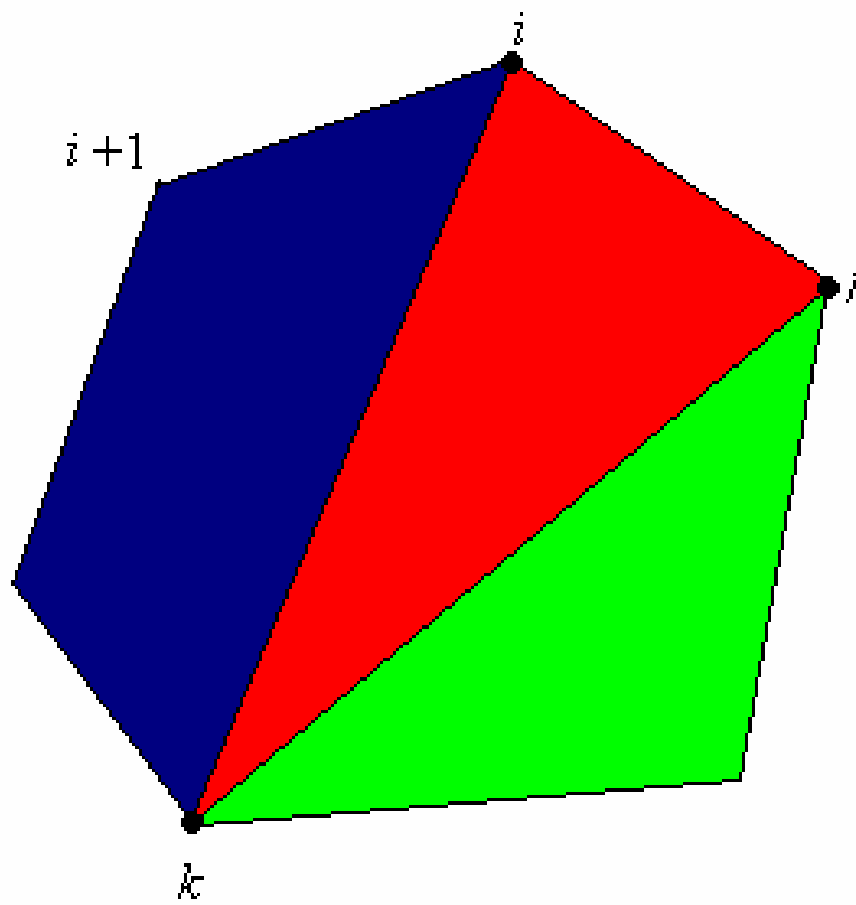


# 分析

- 用 $d[i,j]$ 表示由顶点 $i, i+1, \dots, j$ 组成的多边形(注意 $i$ 可以大于 $j$ ) 的最小代价
  - 方案一: 枚举三个顶点, 组成一个三角形, 决策是 $O(n^3)$ 的
  - 方案二: 边 $(i,j)$ 一定属于一个唯一的三角形, 设第三个顶点为 $k$ , 则决策仅为 $O(n)$
- 采用方案二, 状态 $O(n^2)$ , 决策 $O(n)$ , 总 $O(n^3)$

# 分析

- 确定 $k$ 后, 最优方案应该是 $d[i,k]+d[k,j]+w(i,k,j)$
- 因此转移方程 $d[i,j]=\max\{d[i,k]+d[k,j]+w(i,k,j)\}$

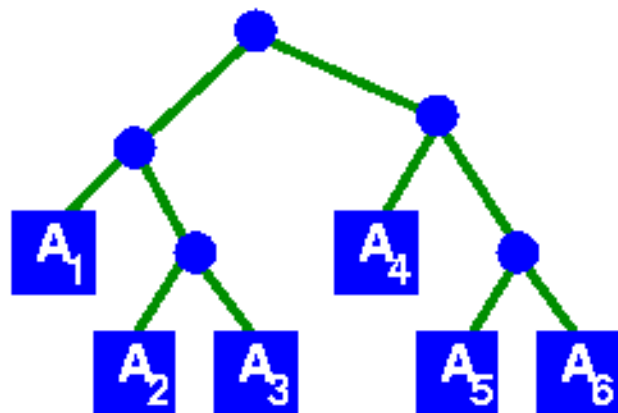


# 变形1: 最优矩阵乘法链

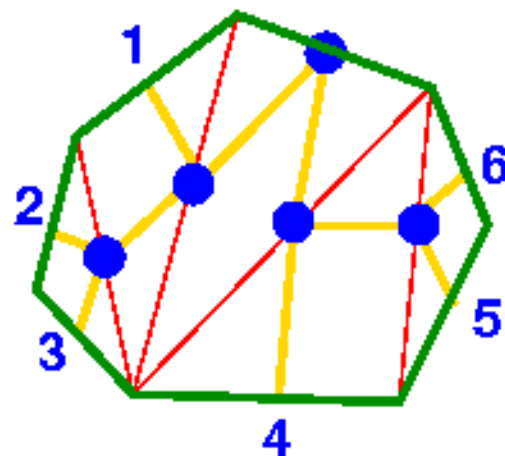
- 需要计算n个矩阵的乘积 $A_1A_2\dots A_n$
- 由于矩阵乘法满足结合律, 可以有多种计算方法. 例如 $A_1$ 是 $10*100$ ,  $A_2$ 是 $100*5$ ,  $A_3$ 是 $5*50$ , 则
  - 顺序1:  $(A_1A_2)A_3$ , 代价为 $10*100*5+10*5*50=7500$
  - 顺序2:  $A_1(A_2A_3)$ , 代价为 $100*5*10+10*100*50=75000$
- 求代价最小的方案(加括号方法)

# 共同的结构

- 用二叉树(**binary tree**)可以表示两个问题相同的结构, 每个结点表示一个区间(结点区间 / 矩阵区间), 左子树和右子树表示分成的两个序列
- 如果在原问题中让 $d[i,j]$ 表示 $i-1 \sim j$ 的最优值, 则在方程形式上也完全等价



$(A_1(A_2A_3))(A_4(A_5A_6))$

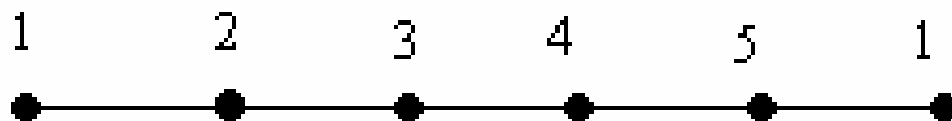


## 变形2. 决斗

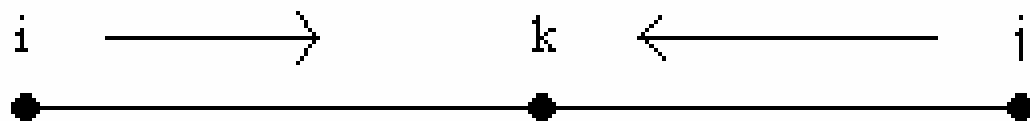
- 编号为 $1\sim n$ 的 $n$ 个人按逆时针方向排成一圈，他们要决斗 $n-1$ 场。每场比赛在某相邻两人间进行，败者退出圈子，紧靠败者右边的人成为与胜者直接相邻的人。
- 任意两人之间决斗的胜负都将在一矩阵中给出（如果 $A[i,j]=1$ 则 $i$ 与 $j$ 决斗 $i$ 总是赢，如果 $A[i,j]=0$ 则 $i$ 与 $j$ 决斗时 $i$ 总是输），
- 求出所有可能赢得整场决斗的人的序号

# 分析

- 首先把圈想象成一条链



- 设 $d[i,j]$ 表示 $i$ 是否能和 $j$ 相遇, 则相遇的充要条件是存在 $k$ ,  $i$ 和 $k$ ,  $k$ 和 $j$ 都能相遇, 且 $i$ 或 $j$ 能打败 $k$



- 同样是 $O(n^2)$ 个状态, 决策 $O(n)$ , 总 $O(n^3)$



## 五、最大m子段和

- 给一个序列 $a_1, a_2, \dots, a_n$
- 求m个不相交(可以相接)的连续序列, 总和尽量大
- 例如 $m=2$ , **1 2 -3 4 5 -6 7**

# 分析

- 设 $d[i,j]$ 为以 $j$ 项结尾的 $i$ 段和的最大值, 则需要枚举此段开头 $y$ 和上一段结尾 $x$ , 即

$$d[i,j]=\max\{d[i-1,x] + a[y..j]\}$$

- 每次需要枚举 $x < y \leq j$ , 决策量为 $O(n^2)$ , 状态为 $O(nm)$ , 共 $O(n^3m)$
- 注意到如果 $a[j-1]$ 也是本段的, 答案变成成为 $d[i,j-1]+a[j]$ , 因此方程优化为

$$d[i,j]=\max\{d[i,j-1]+a[j], d[i-1,x]+a[j]\}, x < j$$

# 分析

- 优化后状态仍然是二维的，但决策减少为 $O(n)$ ，总 $O(n^2m)$
- 可以继续优化. 注意到时间主要耗费在对 $x$ 的枚举上, 计算 $\max\{d[i-1, x]\}$ . 这个值...
- 我们把 $d$ 的第一维称为“阶段”，则本题是典型的多阶段决策问题
  - 计算一个阶段时, 顺便记录本阶段最大值
  - 只保留相邻两个阶段(滚动数组)
- 则时间降为 $O(nm)$ , 空间降为 $O(n)$

## 六、0-1背包问题

- 给定 $n$ 种物品和一个背包, 物品 $i$ 的重量是 $w_i$ , 价值是 $v_i$ , 背包容量为 $c$
- 对于每个物品, 要么装背包, 要么不装
- 选择装背包的物品集合, 使得物品总重量不超过背包容量 $c$ , 且价值和尽量大

# 分析

- 设 $d[i,j]$ 为背包容量为 $j$ 时, 只考虑前 $i$ 个物品时的最大价值和
  - 如果装第 $i$ 个物品, 背包容量只剩 $j-w_i$
  - 如果不装, 背包容量不变
- 因此 $d[i,j]=\max\{d[i,j-w_i]+v_i, d[i-1,j]\}$
- 状态有 $nc$ 个, 每个状态决策只有两个, 因此总时间复杂度为 $O(nc)$ . 用滚动数组后, 空间复杂度只有 $O(c)$

# 分析

- 当 $c$ 大时, 算法效率非常低. 事实上, 由于 $c$ 是数值范围参数, 一般不把它看作输入规模. 这样的 $O(nc)$ 只是一个伪多项式算法
- 事实上, 如果物品重量和背包容量都是实数时, 算法将失败, 因为看起来物品的重量和可以是“任何实数”.
- 但事实是: 物品重量和只有 $2^n$ 种可能的取值, 并不是无限多种

# 分析

- 算法一：枚举 $2^n$ 个子集合，再计算，枚举 $2^n$ ，计算 $n$ ，共 $n2^n$
- 算法二：采用递归枚举，共 $2^n$
- 算法三：先考虑一半元素，保存 $2^{n/2}$ 个和。再考虑后一半元素，每计算出一个和 $w$ ，查找重量 $\leq C-w$ 的元素中价值的最大值。

下面考虑实现细节



# 算法三

- 前一半元素的 $2^{n/2}$ 个和按重量从小到大排序后放在表**a**里. 对于任何两个和 $i, j$ , 如果 $w_i < w_j$ 且 $v_i > v_j$ , 则 $j$ 是不需要保存的, 因此按重量排序好以后也是按价值排序的
- 考虑后一半元素时, 每得到一个重量 $w$ , 用二分查找得到重量不超过 $c-w$ 的最大元素, 则它的价值也最大.
- 预处理时间复杂度 $2^{n/2} \log 2^{n/2}$ , 每个重量 $w$ 二分查找时间为 $\log 2^{n/2}$ , 因此总 $2^{n/2} \log 2^{n/2} = O(n^{1.44^n})$

## 七、再谈最优排序二叉树

- 给 $n$ 个关键码和它们的频率，构造让期望比较次数最小的排序二叉树

# 基本分析

- 设结点*i..j*的最优代价为*d[i,j]*, 则

$$d[i, j] = \min_{i \leq k \leq j} \{d[i, k-1] + d[k+1, j]\} + w[i, j]$$

- 其中*w[i,j]=f<sub>i</sub>+f<sub>i+1</sub>+...+f<sub>j</sub>*, (如果认为根结点的代价为0, 则*w[i,j]*要减去*f<sub>k</sub>*). 直接计算是*O(n<sup>3</sup>)*的
- 设*d[i,j]*的最优决策为*K[i,j]*, 下面证明
$$K[i, j-1] \leq K[i, j] \leq K[i+1, j]$$
- 从而把时间复杂度降到*O(n<sup>2</sup>)*

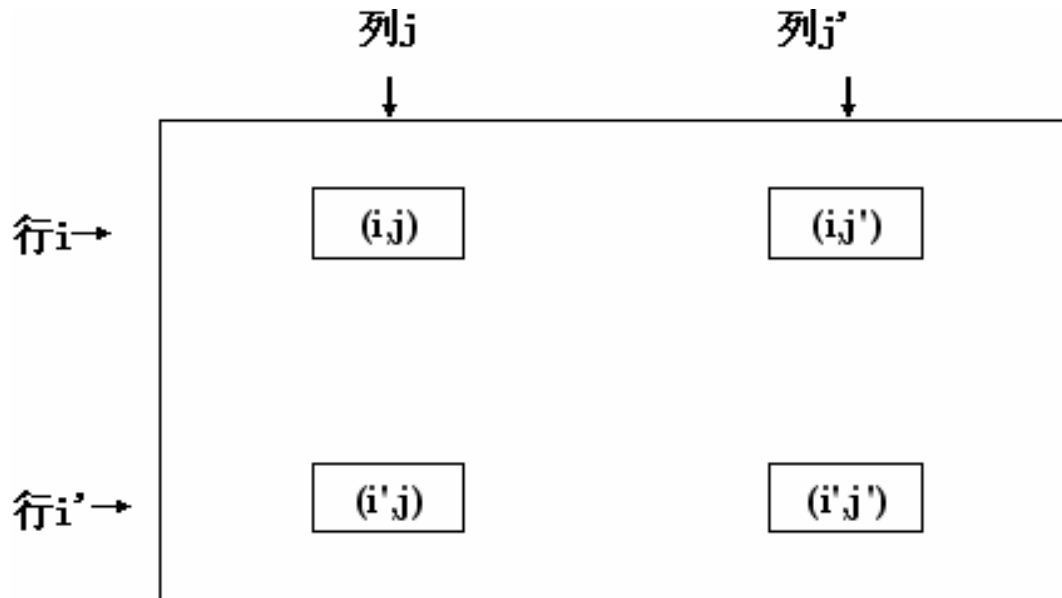
# 四边形不等式

- 凸性(Monge condition/quadrangle inequality)

$$w[i,j]+w[i',j'] \leq w[i',j]+w[i,j'], i \leq i' < j \leq j'$$

- 单调性(区间包含格上)

$$w(i',j) \leq w(i,j'), i \leq i' < j \leq j'$$



# 验证四边形不等式

- 只需验证

$$w[i,j]+w[i+1,j+1]\leq w[i+1,j]+w[i,j+1]$$

- 移项得

$$w[i+1,j+1]-w[i+1,j]\leq w[i,j+1]-w[i,j]$$

- 当j固定时记函数 $f(x) = w[x,j+1]-w[x,j]$ , 则上式变为:  $f(i+1)\leq f(i)$ , 因此

$f(i)$ 是减函数,  $w$ 为凸;  $f(i)$ 是增函数,  $w$ 为凹

- 固定i有同样的结论(减函数时为凸)

# 本题中的w

- 本题中, w的凸性更好证明:

$$\begin{aligned} & w[i,j] + w[i+1,j+1] \\ &= \textcolor{red}{w[i,j]} + (\textcolor{teal}{w[i,j]} + \textcolor{teal}{f[j+1]} - \textcolor{red}{f[i]}) \\ &= w[i+1,j] + w[i,j+1] \end{aligned}$$

- 两边是完全相等的. 或者计算

$$f(x) = w[x,j+1] - w[x,j] = f[i+1] = \text{常数}$$

- 常数既是增函数也是减函数, 因此  
本题中, w既为凸也为凹

# 定理

- 考虑递归式  $d[i,j] = \min\{d[i,k-1] + d[k,j] + w[i,j]\}$
- 定理(**F.Yao**): 若  $w$  满足四边形不等式, 则  $d$  也满足四边形不等式, 即

$$d[i,j] + d[i',j'] \leq d[i',j] + d[i,j'], \quad i \leq i' \leq j \leq j'$$

- 证明: 对长度  $i=j'-i$  归纳, 显然  $i \leq 1$  时正确.  $i=i'$  或  $j=j'$  时(同一行或同一列), 等式显然成立
  - 情形1:  $i'=j$ , 退化为反三角不等式
  - 情形2:  $i' < j$



# 情形1. 反三角不等式

- $i'=j$ 时,  $d[i,j]+d[i',j'] \leq d[i',j]+d[i,j']$ 退化为
$$d[i,j]+d[j,j'] \leq d[i,j']$$
- 设 $k$ 为让 $d[i,j']$ 取最小值的决策(有多个时取最大的一个 $k$ , 后同).
- 若 $k \leq j$ , 则 $k$ 是计算 $d[i,j]$ 考虑过的合法决策
$$d[i,j] \leq w[i,j]+d[i,k-1]+d[k,j]$$
- 两边加上 $d[j,j']$ , 得
$$d[i,j]+d[j,j'] \leq w[i,j]+d[i,k-1]+d[k,j]+d[j,j']$$

# 情形1. 反三角形不等式

- 设 $k$ 为让 $d[i,j']$ 取最小值的决策.  $k \leq j$ 时有
$$d[i,j] + d[j,j'] \leq w[i,j] + d[i,k-1] + d[k,j] + d[j,j']$$
- 用单调性和反三角形不等式(归纳假设), 有
$$d[i,j] + d[j,j'] \leq w[i,j'] + d[i,k-1] + d[k,j']$$
- 由于 $k$ 是最佳决策, 右边恰好是 $d[i,j']$ , 这就证明了情形1中 $k \leq j$ 的情形,  $k > j$ 时类似

## 情形2. 非退化的情形

- $i' < j$ 时, 右边保留两项 $d[i', j]$ 和 $d[i, j']$ . 设二者取最小值时的决策分别为 $y$ 和 $z$ , 仍需分 $z \leq y$ 和 $z > y$ 两种情况(对称). 下面只考虑 $z \leq y$ 时
- $y$ 和 $z$ 是合法决策, 因此 $y \leq j$ ,  $z > i$ , 且
$$d[i, j] \leq w[i, j] + d[i, z-1] + d[z, j]$$
$$d[i', j'] \leq w[i', j'] + d[i', y-1] + d[y, j']$$
- 两式相加并整理, 对应项写在一起, 得

## 情形2. 非退化的情形

- 两式相加并整理, 对应项写在一起, 右边 $\leq$   
 $w[i,j]+w[i',j']+\cancel{d[i,z-1]}+\cancel{d[i',y-1]}+d[z,j]+d[y,j']$
- 因 $z \leq y$ , 红蓝色分别用四边形不等式, 右边 $\leq$   
 $w[i,j']+\cancel{w[i',j]}+\cancel{d[i,z-1]}+\cancel{d[i',y-1]}+d[z,j']+\cancel{d[y,j]}$
- 按红蓝色分别组合, 得
$$d[i,j]+d[i',j'] \leq d[i,j']+\cancel{d[i',j]}$$
- $z \leq y$ 时命题得证.  $z > y$ 时类似

# 决策单调性

- 进一步地,  $d$  的凸性可以推出决策的单调性
- 设  $k[i,j]$  为让  $d[i,j]$  取最小值的决策, 下面证明
$$k[i,j] \leq k[i,j+1] \leq k[i+1,j+1], i \leq j$$
- 即:  $k$  在同行同列上都是递增的
- 证明:  $i=j$  时显然成立. 由对称性, 只需证明  $k[i,j] \leq k[i,j+1]$ . 记  $d_k[i,j] = d[i,k-1] + d[k,j] + w[i,j]$ , 则只需要证明对于所有的  $i < k \leq k' \leq j$ , 有

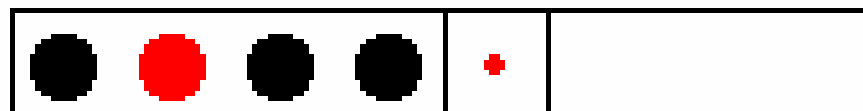
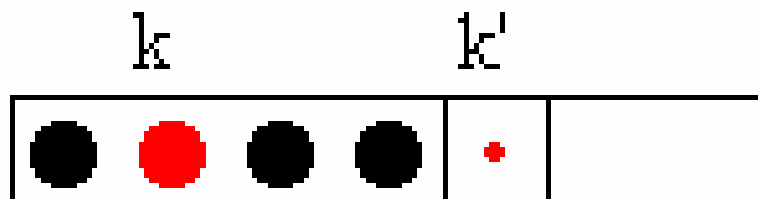
$$d_{k'}[i,j] \leq d_k[i,j] \rightarrow d_{k'}[i,j+1] \leq d_k[i,j+1]$$

# 决策单调性

- 事实上，我们可以证明一个更强的式子

$$d_k[i,j]-d_{k'}[i,j] \leq d_k[i,j+1]-d_{k'}[i,j+1] \quad (i \leq k \leq k' \leq j)$$

- $k'$ 在 $[i,j]$ 更优(左 $\geq 0$ ) $\rightarrow k'$ 在 $[i,j+1]$ 上也更优(右 $\geq 0$ )
- 设 $k'$ 是 $[i,j]$ 的最优值，则对于它左边的任意 $k$ ,  $k'$ 在 $[i,j]$ 更优可推出 $k'$ 在 $[i,j+1]$ 上也更优, 因此在区间 $[i,j+1]$ 上,  $k'$ 左边的值都比它大, 如下图



# 决策单调性

- 欲证 $d_k[i,j]-d_{k'}[i,j]\leq d_k[i,j+1]-d_{k'}[i,j+1]$ , 移项得
$$d_k[i,j]+d_{k'}[i,j+1]\leq d_k[i,j+1]+d_{k'}[i,j]$$
- 按定义展开, 两边消去 $w[i,j]+w[i,j+1]$ , 得
$$d[i,k-1]+d[k,j]+d[i,k'-1]+d[k',j+1]\leq d[i,k-1]+d[k,j+1]+d[i,k'-1]+d[k',j]$$
- 同时消去红色部分, 得
$$d[k,j]+d[k',j+1]\leq d[k,j+1]+d[k',j]$$
- 这就是 $k\leq k'\leq j<j+1$ 时 $d$ 的凸性

# 算法优化

- 由于 $k$ 是单调, 计算 $d[i,j]$ 时决策只需从 $k[i,j-1]$ 枚举到 $k[i+1,j]$ 即可
- 当 $L=j-i$ 固定时,
  - $d[1,L+1]$ 的决策是 $k[1,L]$ 到 $k[2,L+1]$
  - $d[2,L+2]$ 的决策是 $k[2,L+1]$ 到 $k[3,L+2]$
  - $d[3,L+3]$ 的决策是 $k[3,L+2]$ 到 $k[4,L+3]$
  - ...
  - 总决策是从 $k[1,L]$ 到 $k[n-L+1,n]$ , 共 $O(n)$
- 对于 $O(n)$ 个 $L$ , 共 $O(n^2)$ 个决策.
- 本题方程略有不同, 但也可用类似方法证明



## 八、最优合并问题

- 有 $n$ 个正整数, 每次可以合并两个相邻的数(相加), 代价为相加后的新数.
- 按如何的顺序把所有的数合并成一个, 使得代价总和尽量小?

# 分析

- 假设数 $i \sim j$ 的最小合并代价为 $d[i,j]$ , 考虑最后一次合并, 有

$$d[i,j] = \min\{d[i,k-1]+d[k,j]+w[i,j]\}$$

- 其中 $w[i,j] = a[i]+\dots+a[j]$
- 显然 $w[i,j]$ 是凸的和增的, 因此用四边形不等式优化后时间复杂度降为 $O(n^2)$
- 下面进一步优化到 $O(n \log n)$

# 错误的贪心法

- 贪心法: 每次采取代价最少的合并方案
- 不一定得到最优解! 最优解为74

5	3	4	1	3	2	3	4
5	3	4	4		2	3	4
5	3	4	4		5		4
5	7		4		5		4
5	7		9				4
12			9				4
12			13				
25							

$$4+5+7+9+12+13+25=75$$

(a)

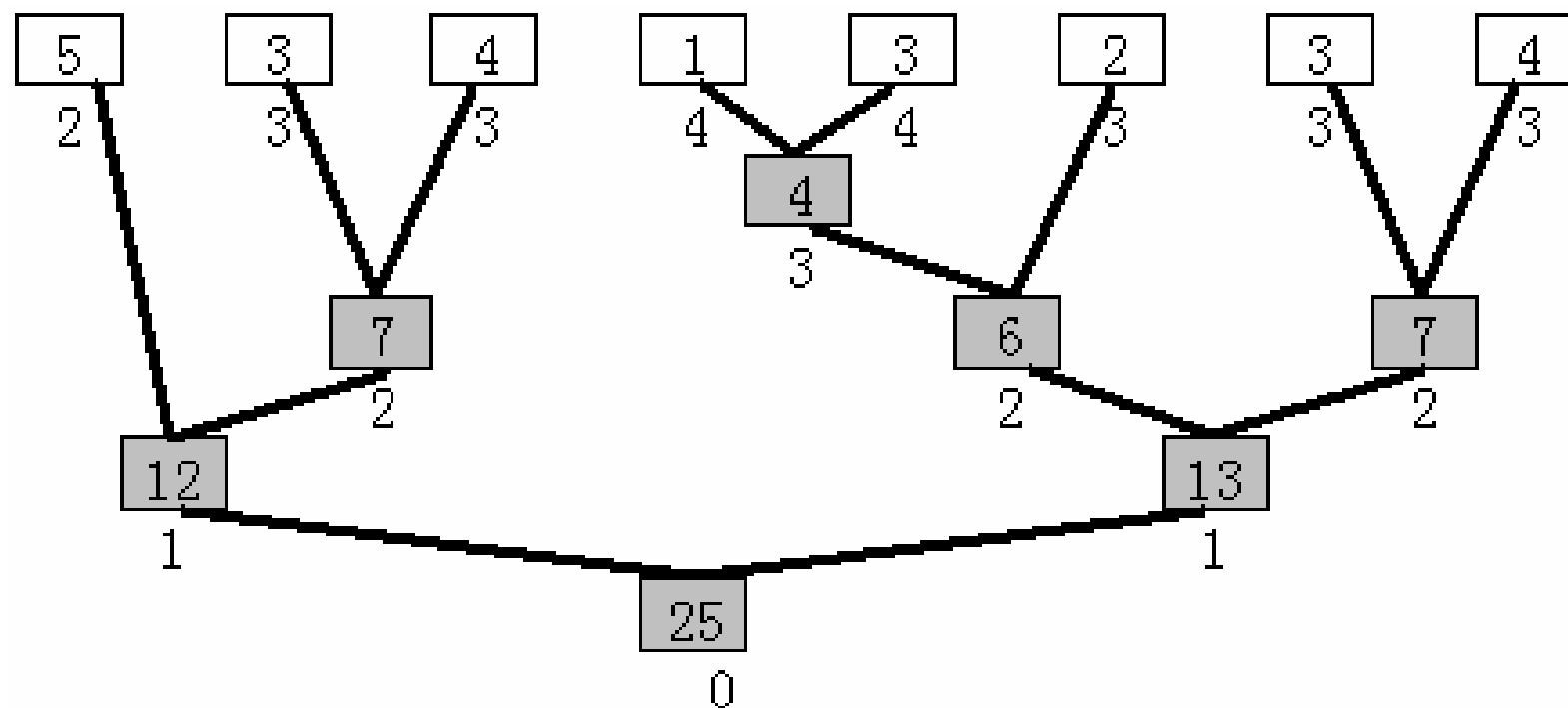
5	3	4	1	3	2	3	4
5	3	4	4		2	3	4
5	3	4	4		5		4
5	7		4		5		4
5	7		4		9		
5	11				9		
16					9		
25							

$$4+5+7+9+11+16+25=77$$

(b)

# 分析

- 可以把合并过程画成一棵树, 标记结点深度

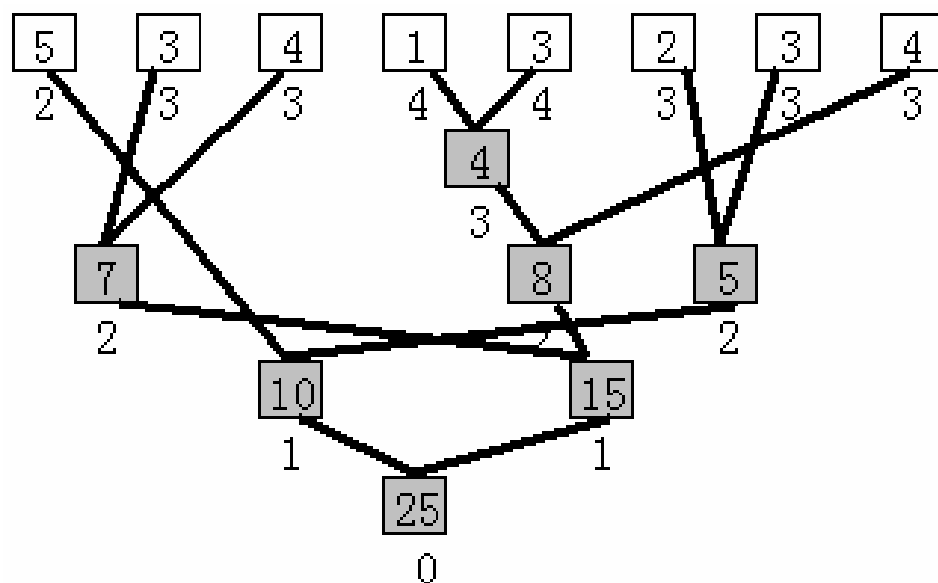


计算方法一:  $4+7+6+7+12+13+25=74$

计算方法二:  $5*2+3*3+4*3+1*4+3*4+2*3+3*3+4*3=74$

# 相容结点贪心法

- 相容结点对: 中间没有原始结点的结点对
- 修改的贪心法:** 每次合并和最小的相容结点*i*, *j*. 如果有多个, 因此让*i*和*j*尽量小



计算方法一:  $4+7+8+5+10+15+25=74$

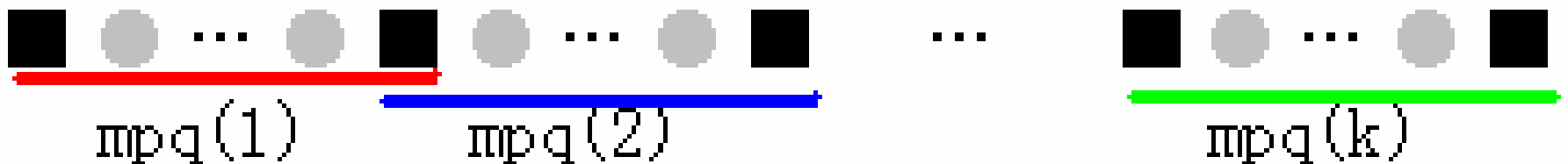
计算方法二:  $5*2+3*3+4*3+1*4+3*4+2*3+3*3+4*3=74$

# 重组

- 修改的贪心法没有按照题目要求合并, 得到的合并树不一定是合法的答案, 但它同样能得到了一棵树, 代价和仍然是 $\sum\{d_i * a_i\}$ , 其中 $d_i$ 是叶子 $i$ 的深度
- **定理:** 用相容结点贪心法得到的树, 在保持各叶子的深度 $d_i$ 不变的情况下可以重组成一棵满足题目要求的合并树
- **算法:** 用修改贪心法求出深度序列, 再重组

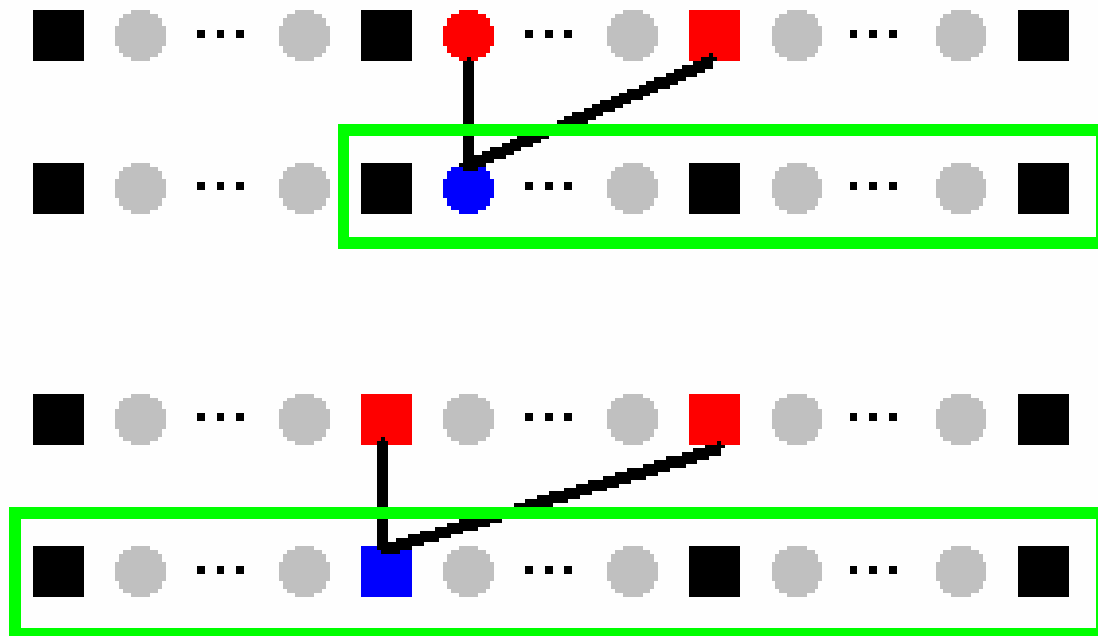
# 实现上的考虑

- 合并结点用圆形表示, 原始结点用正方形表示
- 第 $i$ 个圆形序列片段连同它左边、右边的正方形组成一个结点集合 $mpq(i)$
- 每次合并的两个结点一定是某一个 $mpq$ 的最小值 $min1(i)$ 和次小值 $min2(i)$
- **贪心过程:** 每次选择使 $min1(i)+min2(i)$ 最小的 $i$ , 合并结点 $min1(i)$ 和 $min2(i)$



# 分析

- 合并操作
  - 两个圆形: 没有影响
  - 一个圆形和一个正方形: 合并两个mpq
  - 两个正方形: 三个mpq合并





# 算法梗概

- 贪心过程:  $O(n \log n)$ , 使用可并优先队列
- 标记深度:  $O(n)$
- 树重组:  $O(n)$