Pólya原理及其应用

华东师大二附中 符文杰

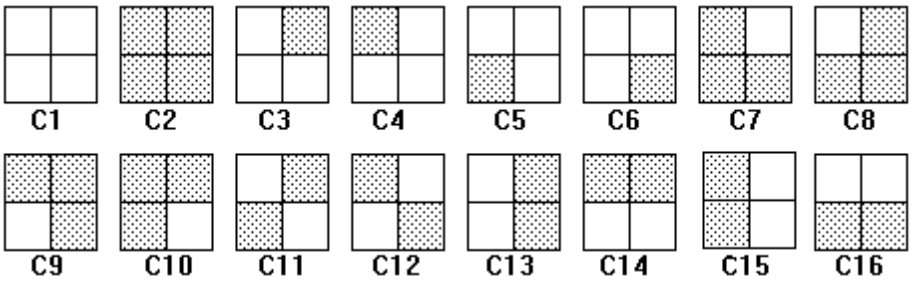
Pólya原理是组合数学中，用来计算全部互异的组合状态的个数的一个十分高效、简便的工具。下面，我就向大家介绍一下什么是Pólya原理以及它的应用。请先看下面这道例题：

【例题1】

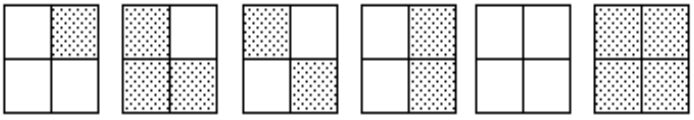
对2*2的方阵用黑白两种颜色涂色，问能得到多少种不同的图像？经过旋转使之吻合的两种方案，算是同一种方案。

【问题分析】

由于该问题规模很小，我们可以先把所有的涂色方案列举出来。



一个2*2的方阵的旋转方法一共有4种：旋转0度、旋转90度、旋转180度和旋转270度。(注：本文中默认旋转即为顺时针旋转) 我们经过尝试，发现其中互异的一共只有6种：C3、C4、C5、C6是可以通过旋转相互变化而得，算作同一种；C7、C8、C9、C10是同一种；C11、C12是同一种；C13、C14、C15、C16也是同一种； C1和C2是各自独立的两种。于是，我们得到了下列6种不同的方案。



但是，一旦这个问题由2*2的方阵变成20*20甚至200*200的方阵，我们就不能再一一枚举了，利用Pólya原理成了一个很好的解题方法。在接触Pólya原理之前，首先简单介绍Pólya原理中要用到的一些概念。

群：给定一个集合$G=\{a,b,c,\ldots\}$和集合G上的二元运算，并满足:

(a) 封闭性：$\forall a,b\in G, \exists c\in G, a*b=c$。

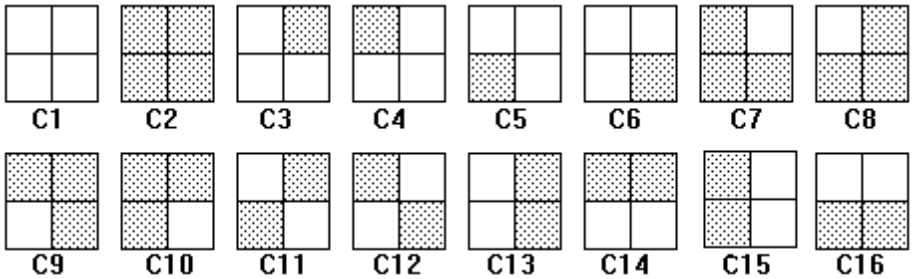
(b) 结合律：$\forall a,b,c\in G, (a*b)*c=a*(b*c)$。

(c) 单位元：$\exists e\in G, \forall a\in G, a*e=e*a=a$。

(d) 逆元：$\forall a\in G, \exists b\in G, a*b=b*a=e$，记$b=a^{-1}$。

则称集合G在运算$*$之下是一个群，简称G是群。一般$a*b$简写为ab。

置换：n个元素1,2,…,n之间的一个置换$\begin{pmatrix}1 & 2 & \cdots & n\\ a_1 & a_2 & \cdots & a_n\end{pmatrix}$表示1被1到$n$中的某个数$a_1$取代，2被1到$n$中的某个数$a_2$取代，直到$n$被1到$n$中的某个数$a_n$取代，且$a_1,a_2,\ldots,a_n$互不相同。本例中有4个置换：



转0° a1=$\begin{pmatrix}1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16\\1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16\end{pmatrix}$

转90° a2=$\begin{pmatrix}1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16\\1&2&6&3&4&5&10&7&8&9&12&11&16&13&14&15\end{pmatrix}$

转180° a3=$\begin{pmatrix}1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16\\1&2&5&6&3&4&9&10&7&8&11&12&15&16&13&14\end{pmatrix}$

转270° a4=$\begin{pmatrix}1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16\\1&2&4&5&6&3&8&9&10&7&12&11&14&15&16&13\end{pmatrix}$

置换群：置换群的元素是置换，运算是置换的连接。例如：

$$\begin{pmatrix}1&2&3&4\\3&1&2&4\end{pmatrix}\begin{pmatrix}1&2&3&4\\4&3&2&1\end{pmatrix}=\begin{pmatrix}1&2&3&4\\3&1&2&4\end{pmatrix}\begin{pmatrix}3&1&2&4\\2&4&3&1\end{pmatrix}=\begin{pmatrix}1&2&3&4\\2&4&3&1\end{pmatrix}$$

可以验证置换群满足群的四个条件。

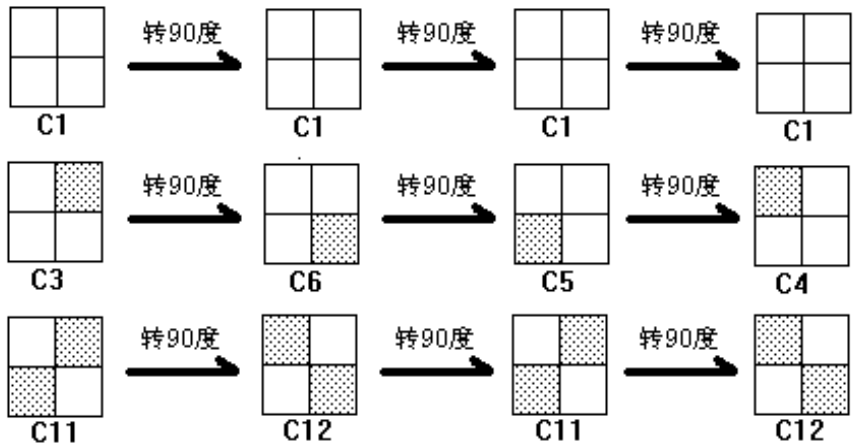
本题中置换群G={转0°、转90°、转180°、转270°}

我们再看一个公式：$|E_k|\cdot|Z_k|=|G|$ k=1…n

该公式的一个很重要的研究对象是群的元素个数，有很大的用处。

Z_k (K不动置换类)：设G是1…n的置换群。若K是1…n中某个元素，G中使K保持不变的置换的全体，记以Z_k，叫做G中使K保持不动的置换类，简称K不动置换类。

*如本例中：*G是涂色方案1~16的置换群。对于方案1，四个置换都使方案1保持不变，所以$Z_1=\{a_1, a_2, a_3, a_4\}$；对于方案3，只有置换$a_1$使其不变，所以$Z_3=\{a_1\}$；对于方案11，置换$a_1$和$a_3$使方案其保持不变，



所以$Z_{11}=\{a_1, a_3\}$。

E_k(等价类)：设G是1…n的置换群。若K是1…n中某个元素，K在G作

用下的轨迹，记作E_k。即K在G的作用下所能变化成的所有元素的集合。

*如本例中：*方案1在四个置换作用下都是方案1，所以$E_1=\{1\}$；方案3，在a_1下是3，在a_2下变成6，在a_3下变成5，在a_4下变成4，所以$E_3=\{3,4,5,6\}$；方案11，在a_1、a_3下是11，在a_2、a_4下变成12，所以$E_{11}=\{11,12\}$。

本例中的数据，也完全符合这个定理。*如本例中：*

$$|E_1|\cdot|Z_1| = 1\times4 = 4 = |G|$$

$$|E_3|\cdot|Z_3| = 4\times1 = 4 = |G|$$

$$|E_{11}|\cdot|Z_{11}| = 2\times2 = 4 = |G|$$

限于篇幅，这里就不对这个定理进行证明。

接着就来研究每个元素在各个置换下不变的次数的总和。见下表：

置换\S_{ij}\元素j a_I	1	2	……	16	$D(a_i)$
a1	$S_{1,1}$	$S_{1,2}$	……	$S_{1,16}$	$D(a_1)$
a2	$S_{2,1}$	$S_{2,2}$	……	$S_{2,16}$	$D(a_2)$
a3	$S_{3,1}$	$S_{3,2}$	……	$S_{3,16}$	$D(a_3)$
a4	$S_{4,1}$	$S_{4,2}$	……	$S_{4,16}$	$D(a_4)$
$\sum_{j=1}^{16}\|Z_j\| = \sum_{j=1}^{4} D(a_j)$					
$\|Z_j\|$	$\|Z_1\|$	$\|Z_2\|$	……	$\|Z_{16}\|$	

其中

$$S_{ij}=\begin{cases}0 & \text{当}a_i\notin Z_j\text{,即}j\text{在}a_i\text{的变化下变动了}\\ 1 & \text{当}a_i\in Z_j\text{,即}j\text{在}a_i\text{的变化下没有变}\end{cases}$$

$D(a_j)$ 表示在置换a_j下不变的元素的个数

*如本题中：*涂色方案1在a_1下没变动，$S_{1,1}=1$；方案3在a_3变动了，$S_{3,3}=0$；在置换a_1的变化下16种方案都没变动，$D(a_1)=16$；在置换a_2下只有1、2这两种方案没变动，$D(a_2)=2$。

一般情况下，我们也可以得出这样的结论：$\sum_{j=1}^{n}|Z_j|=\sum_{i=1}^{s}D(a_i)$

我们对左式进行研究。

不妨设N={1，.....，n}中共有L个等价类，$N=E_1+E_2+\ldots\ldots+E_L$，则当j和k属于同一等价类时，有$|Z_j|=|Z_k|$。所以

$$\sum_{k=1}^{n}|Z_k|=\sum_{i=1}^{L}\sum_{k\in E_i}|Z_k|=\sum_{i=1}^{L}|E_i|\cdot|Z_i|=L\cdot|G|$$

这里的L就是我们要求的互异的组合状态的个数。于是我们得出：

$$L=\frac{1}{|G|}\sum_{k=1}^{n}|Z_k|=\frac{1}{|G|}\sum_{j=1}^{s}D(a_j)$$

利用这个式子我们可以得到本题的解 L=(16+2+4+2)/4=6 与前面枚举得到的结果相吻合。这个式子叫做Burnside引理。

但是，我们发现要计算$D(a_j)$的值不是很容易，如果采用搜索的方法，总的时间规模为O(n×s×p)。(n表示元素个数，s表示置换个数，p表示格子数，这里n的规模是很大的) 下一步就是要找到一种简便的$D(a_j)$的计算方法。先介绍一个循环的概念：

循环：记

$$(a_1a_2\cdots a_n)=\begin{pmatrix}a_1 & a_2 & \cdots & a_{n-1} & a_n\\ a_2 & a_3 & \cdots & a_n & a_1\end{pmatrix}$$

称为n阶循环。每个置换都可以写若干互不相交的循环的乘积，两个循环$(a_1a_2\ldots a_n)$和$(b_1b_2\ldots b_n)$互不相交是指$a_i\neq b_j$, $i,j=1,2,\ldots,n$。例如：

$$\begin{pmatrix}1&2&3&4&5\\3&5&1&4&2\end{pmatrix}=(13)(25)(4)$$

这样的表示是唯一的。置换的循环节数是上述表示中循环的个数。例如(13)(25)(4)的循环节数为3。

有了这些基础，就可以做进一步的研究，我们换一个角度来考虑这个问题。我们给2*2方阵的每个方块标号，如下图：

2	1
3	4

构造置换群$G'=\{g_1,g_2,g_3,g_4\}$, $|G'|=4$，令g_i的循环节数为$c(g_i)$ $(i=1,2,3,4)$

在G'的作用下，其中

g_1表示转0°， 即$g_1=(1)(2)(3)(4)$ $c(g_1)=4$

g_2表示转90°， 即$g_2=(4\ 3\ 2\ 1)$ $c(g_2)=1$

g_3表示转180°， 即$g_3=(1\ 3)(2\ 4)$ $c(g_3)=2$

g_4表示转270°， 即$g_4=(1\ 2\ 3\ 4)$ $c(g_4)=1$

我们可以发现，g_i的同一个循环节中的对象涂以相同的颜色所得的图像数$m^{c(g_i)}$正好对应G中置换a_i作用下不变的图象数，即

$2^{c(g_1)}=2^4=16=D(a_1)$ $2^{c(g_2)}=2^1=2=D(a_2)$

$2^{c(g_3)}=2^2=4=D(a_3)$ $2^{c(g_4)}=2^1=2=D(a_4)$

由此我们得出一个结论：

设G是p个对象的一个置换群，用m种颜色涂染p个对象，则不同

$$L=\frac{1}{|G|}(m^{c(g_1)}+m^{c(g_2)}+\cdots+m^{c(g_s)})$$

染色方案为：

其中$G=\{g_1,\ldots g_s\}$ $c(g_i)$为置换g_i的循环节数$(i=1\ldots s)$

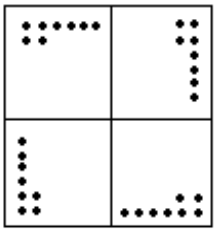
这就是所谓的Pólya定理。我们发现利用Pólya定理的时间复杂度为$O(s\times p)$ (这里s表示置换个数，p表示格子数)，与前面得到的Burnside引理相比之下，又有了很大的改进，其优越性就十分明显了。Pólya定理充分挖掘了研究对象的内在联系，总结了规律，省去了许多不必要的盲目搜索，把解决这类问题的时间规模降到了一个非常低的水平。

现在我们把问题改为：$n\times n$的方阵，每个小格可涂m种颜色，求在旋转操作下本质不同的解的总数。

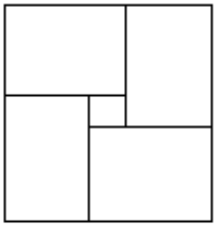
【问题分析】

先看一个很容易想到的搜索的方法。(见附录)

这样搜索的效率是极低的，它还有很大的改进的余地。前面，我们采用的方法是先搜后判，这样的盲目性极高。我们需要边搜边判，避免过多的不必要的枚举，我们更希望把判断条件完全融入到搜索的边界中去，消灭无效的枚举。这个美好的希望是可以实现的。



n为偶数时



n为奇数时

我们可以在方阵中分出互不重叠的长为$[(n+1)/2]$，宽为$[n/2]$的四个矩阵。当n为偶数时，恰好分完；当n为奇数时，剩下中心的一

个格子，它在所有的旋转下都不动，所以它涂任何颜色都对其它格子没有影响。令m种颜色为0~m-1，我们把矩阵中的每格的颜色所代表的数字顺次(左上角从左到右，从上到下；右上角从上到下，从右到左；.....)排成m进制数，然后就可以表示为一个十进制数，其取值范围为0~$m^{[n_2/4]}$-1。(因为[n/2]*[(n+1)/2]=[n^2/4]) 这样，我们就把一个方阵简化为4个整数。我们只要找到每一个等价类中左上角的数最大的那个方案(如果左上角相同，就顺时针方向顺次比较) 这样，在枚举的时候其它三个数一定不大于左上角的数，效率应该是最高的。

进一步考虑，当左上角数为i时,(0≤i≤R-1) 令R=$m^{[n_2/4]}$
可分为下列的4类：

① 其它三个整数均小于i，共i^3个。

② 右上角为i，其它两个整数均小于i，共i^2个。

③ 右上角、右下角为i，左下角不大于i，共i+1个。

④ 右下角为i，其它两个整数均小于i，且右上角的数不小于左下角的，共i(i+1)/2个。

$$L=\sum_{i=0}^{R-1}(i^3+i^2+i+1+\frac{1}{2}i(i+1))=\sum_{i=0}^{R-1}(i^3+\frac{3}{2}i^2+\frac{3}{2}i+1)$$
$$=\sum_{i=1}^{R}((i-1)^3+\frac{3}{2}(i-1)^2+\frac{3}{2}(i-1)+1)=\sum_{i=1}^{R}(i^3-\frac{3}{2}i^2+\frac{3}{2}i)$$
$$=\frac{1}{4}R^2(R+1)^2-\frac{3}{2}\times\frac{1}{6}R(R+1)(2R+1)+\frac{3}{2}\times\frac{1}{2}R(R+1)$$
$$=\frac{1}{4}(R^4+R^2+2R)$$

因此，

当n为奇数时，还要乘一个m。

由此我们就巧妙地得到了一个公式。但是，我们应该看到要想到这个公式需要很高的智能和付出不少的时间。另一方面，这种方法只能对这道题有用而不能广泛地应用于一类试题，具有很大的不定性因素。因此，如果能掌握一种适用面广的原理，就会对解这一类题有很大的帮助。

下面我们就采用Pólya定理。我们可以分三步来解决这个问题。

1．确定置换群

在这里很明显只有4个置换：转0°、转90°、转180°、转270°。所以，置换群G={转0°、转90°、转180°、转270°}。

2．计算循环节个数

首先，给每个格子顺次编号（1~n^2），再开一个二维数组记录置换后的状态。最后通过搜索计算每个置换下的循环节个数，效率为一次方级。

3．代入公式

即利用Pólya定理得到最后结果。

$$L=\frac{1}{|G|}(m^{c(g1)}+m^{c(g2)}+\cdots+m^{c(gs)})$$

【程序题解】

```
const
  maxn=10;
var
  a,b:array[1..maxn,1..maxn] of integer;{记录方阵的状态}
  i,j,m,n:integer;{m颜色数;n方阵大小}
  l,l1:longint;
```

```
procedure xz;{将方阵旋转90°}
var
  i,j:integer;
begin
  for i:=1 to n do
    for j:=1 to n do
      a[j,n+1-i]:=b[i,j];
  b:=a
end;
procedure xhj;{计算当前状态的循环节个数}
var
  i,j,i1,j1,k,p:integer;
begin
  k:=0;{用来记录循环节个数，清零}
  for i:=1 to n do
    for j:=1 to n do
      if a[i,j]>0 then{搜索当前尚未访问过的格子}
        begin
          inc(k);{循环节个数加1}
          i1:=(a[i,j]-1) div n;
          j1:=(a[i,j]-1) mod n+1;{得到这个循环的下一个格子}
          a[i,j]:=0;{表示该格已访问}
          while a[i1,j1]>0 do begin
            p:=a[i1,j1];{暂存当前格的信息}
            a[i1,j1]:=0;{置已访问标志}
            i1:=(p-1) div n+1;
            j1:=(p-1) mod n+1{得到这个循环的下一个格子}
          end{直到完整地访问过这个循环后退出}
        end;
  l1:=1;
  for i:=1 to k do l1:=l1*m;{计算m的k次方的值}
```

```
  l:=l+l1{进行累加}
end;
begin
  writeln('Input m,n=');
  readln(m,n);{输入数据}
  for i:=1 to n do
    for j:=1 to n do a[i,j]:=(i-1)*n+j;{对方阵的状态进行初始化}
  b:=a;
  xhj;{计算转0°状态下的循环节个数}
  xz;{转90°}
  xhj; {计算转90°状态下的循环节个数}
  xz;{再转90°}
  xhj; {计算转180°状态下的循环节个数}
  xz;{再转90°}
  xhj; {计算转270°状态下的循环节个数}
  l:=l div 4;
  writeln(l);{输出结果}
  readln
end.
```

在上面的程序中，我暂时回避了高精度计算，因为这和我讲的内容关系不大。

如果大家再仔细地考虑一下，就会发现这个题解还可以继续优化。对n分情况讨论：

① n为偶数：在转0°时，循环节为n^2个，转180°时，循环节为$n^2/2$个，转90°和转270°时，循环节为$n^2/4$个。

② n为偶数：在转0°时，循环节为n^2个，转180°时，循环节为$(n^2+1)/2$个，转90°和转270°时，循环节为$(n^2+3)/4$个。

把这些综合一下就得到：在转0°时，循环节为n^2个，转180°时，循环节为$[(n^2+1)/2]$个，转90°和转270°时，循环节为$[(n^2+3)/4]$个。(其中，方括号表示取整)于是就得到：

$$L=\frac{1}{4}(m^{n^2}+m^{[\frac{n^2+3}{4}]}+m^{[\frac{n^2+1}{2}]}+m^{[\frac{n^2+3}{4}]})$$

这和前面得到的结果完全吻合。

经过上述一番分析，使得一道看似很棘手的问题得以巧妙的解决，剩下的只要做一点高精度计算即可。

通过这几个例子，大家一定对Pólya原理有了八九成的了解，通过和搜索方法的对比，它的优越性就一目了然了。它不仅极大地提高了程序的时间效率，甚至在编程难度上也有减无增。所以，我们在智能和经验不断增长的同时，也不能忽视了原理性的知识。智能和经验固然重要，但是掌握了原理就更加踏实。因此，我们在解题之余，也要不忘对原理性知识的学习，不停给自己充电，使自己的水平有更大的飞跃。

附录 (搜索方法的程序)

```
const
  maxn=10;
type
  sqtype=array[1..maxn,1..maxn] of byte;
var
  n,total,m:integer;
  sq:sqtype;
```

```
function big:boolean;(检验当前方案是否为同一等价类中最大的)
var
  units:array[2..4] of sqtype;(记录三种旋转后的状态)
  i,j,k:integer;
begin
  for i:=1 to n do
    for j:=1 to n do begin
      units[2,j,n+1-i]:=sq[i,j];
      units[3,n+1-i,n+1-j]:=sq[i,j];
      units[4,n+1-j,i]:=sq[i,j]
    end;
  big:=false;
  for k:=2 to 4 do (进行比较)
    for i:=1 to n do begin
      j:=1;
      while (j<=n)and(sq[i,j]=units[k,i,j]) do inc(j);
      if j<=n then
        if sq[i,j]<units[k,i,j]
          then exit
          else break
    end;
  big:=true
end;
procedure make(x,y:byte);(枚举每个格子中涂的颜色)
var i:integer;
begin
  if x>n then begin
    if big then inc(total);
    exit
  end;
  for i:=1 to m do begin
    sq[x,y]:=i;
    if y=n then make(x+1,1) else make(x,y+1)
  end
end;
begin
  writeln('Input m,n=');readln(m,n);
  total:=0;
  make(1,1);
  writeln(total);readln
```

end.

从圆桌问题谈数据结构的综合运用

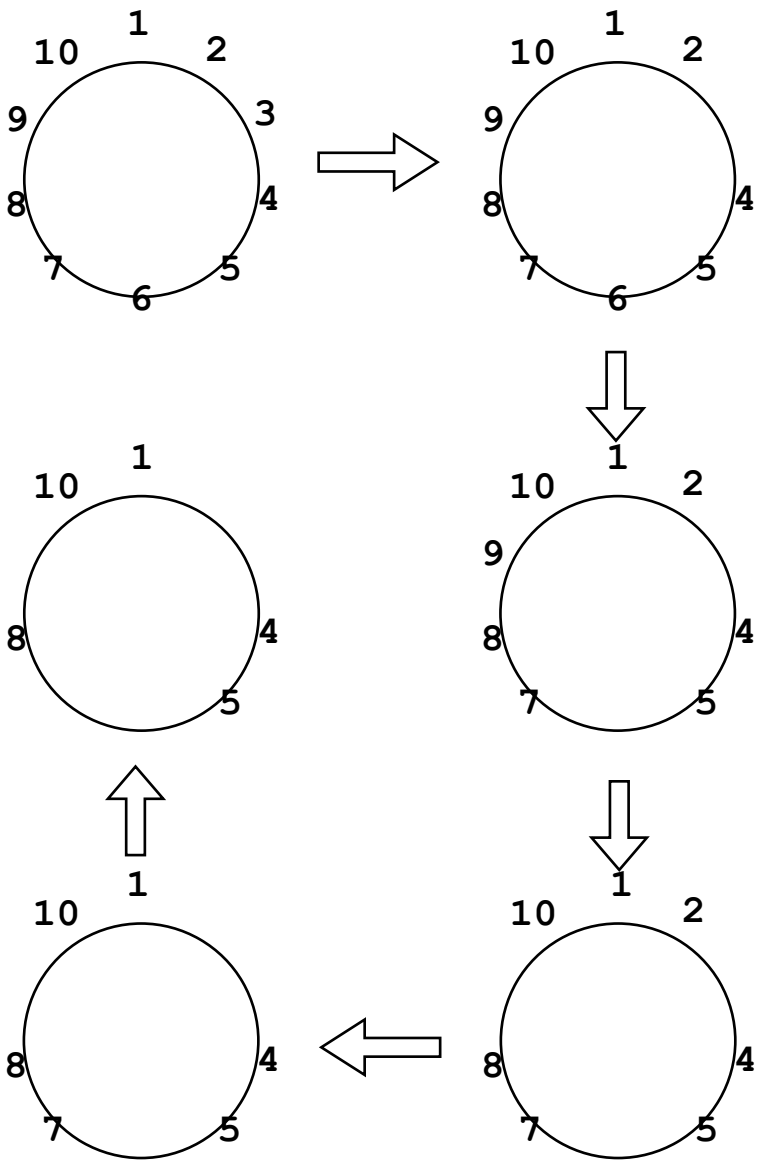
圆桌问题

题目：圆桌上围坐着 2n 个人。其中 n 个人是好人，另外 n 个人是坏人。如果从第一个人开始数数，数到第 m 个人，则立即处死该人；然后从被处死的人之后开始数数，再将数到的第 m 个人处死...依此方法不断处死围坐在圆桌上的人。试问预先应如何安排这些好人与坏人的座位，能使得在处死 n 个人之后，圆桌上围坐的剩余的 n 个人全是好人。

输入：文件中的每一行都有两个数，依次为 n 和 m，表示一个问题的描述信息， n≤32767，m≤32767。

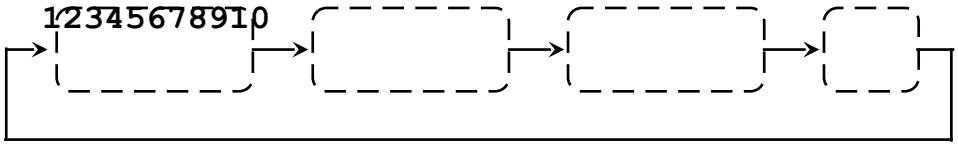
输出：依次输出每一个问题的解。每一个问题的解可以用连续的若干行字符来表示，每行的字符数量不超过 50。但是在一个问题的解中不允许出现空白字符和空行，相邻的两个问题的解之间用空行隔开。用大写字母 G 表示好人，大写字母 B 表示坏人。

圆桌问题实现思想图示（$n=5,m=3$）

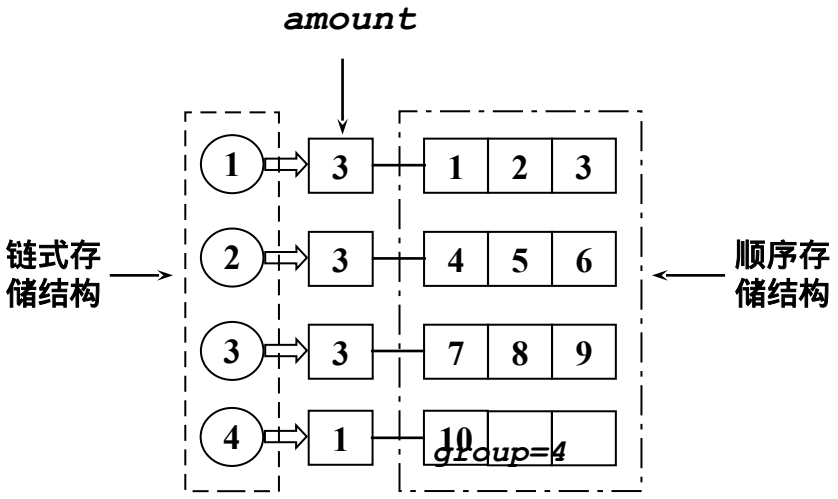


分段式数据结构示意

(思想模型)



(实际模型)



共进行 1+2+2+3+5=13 次操作

改进前后程序效率比较

（测试机器：*P166*）

测试数据	线性表 “查找”法	“优化直接定位”法	
		amount=400	改进前用时是改进后的多少倍
n=200 m=100	0.000s	0.000s	/
n=1000 m=50	0.440s	0.000s	/
n=32767 m=200	5.870s	0.930s	6.312
n=32767 m=1000	29.440s	0.980s	30.041
n=32767 m=10000	294.120s	1.260s	233.43
n=32767 m=20000	588.530s	1.590s	370.14
n=32767 m=32767	963.560s	1.970s	489.12

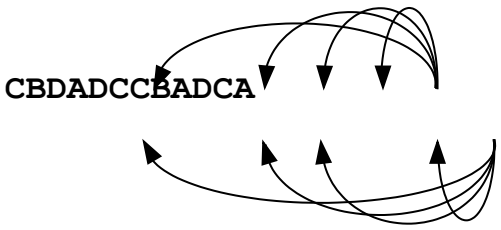
引申

- 横向延伸——约瑟夫环类的问题

  如：《翻牌游戏》、《猴子选大王》

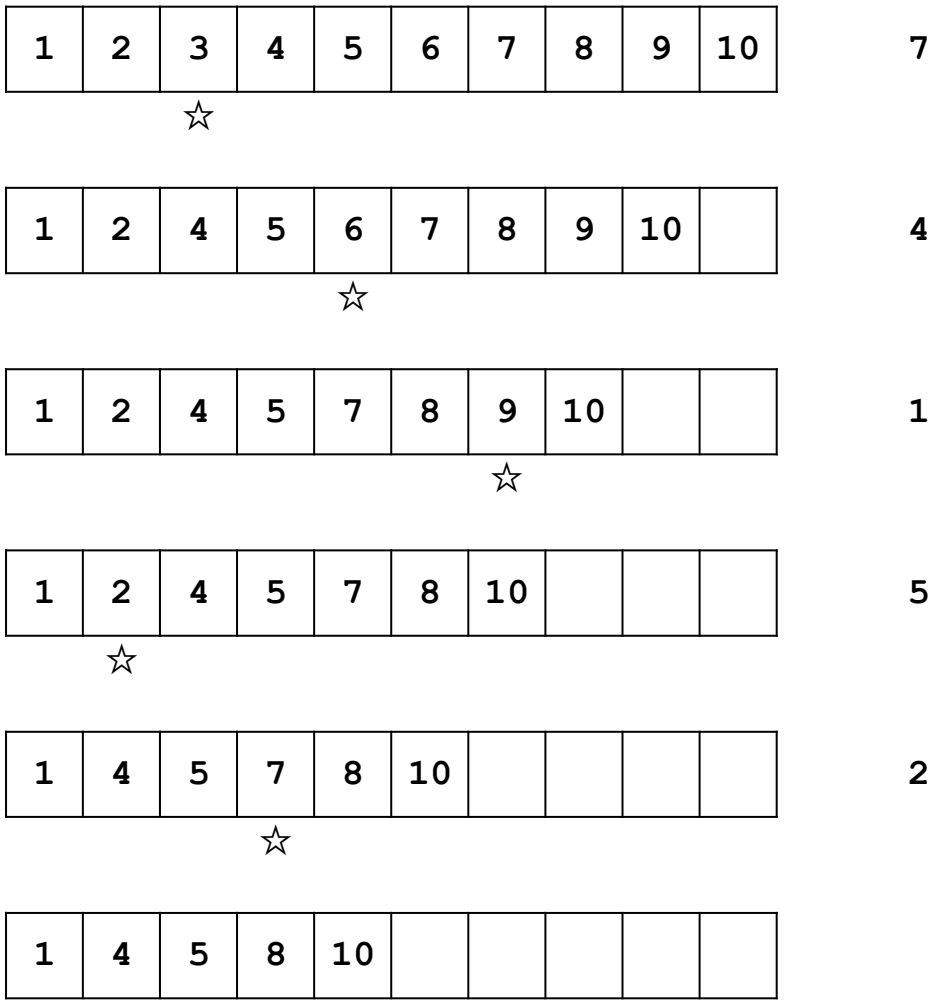
- 纵向延伸——数据结构的综合运用

  在解决一些数据规模较大的问题时有很好的效用。如《隐藏的码字》*(IOI'99)*。在解决这道题目时，如果建立起链式和顺序相结合的数据结构（如下图），程序效率就比较高。



链式和顺序相结合的数据结构实现简单，效果显著，应用比较广泛。当然还有其它的结合，比如二叉堆和顺序结构的一一映射（单射），在解决某些问题时会有很好的效果。

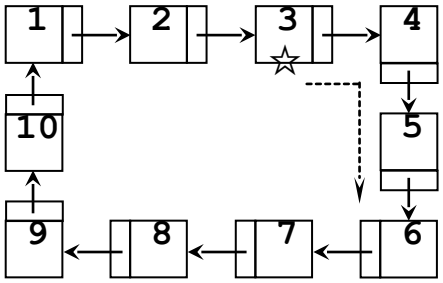
顺序存储结构操作示意



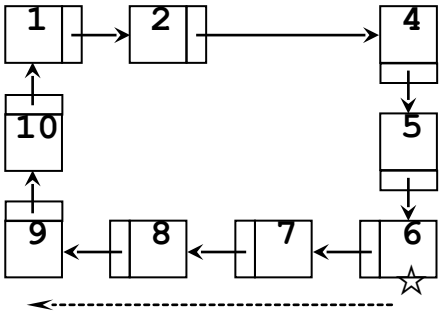
共进行 7+4+1+5+2=19 次操作，时间复杂度 $O(n^2)$。

链式存储结构操作示意

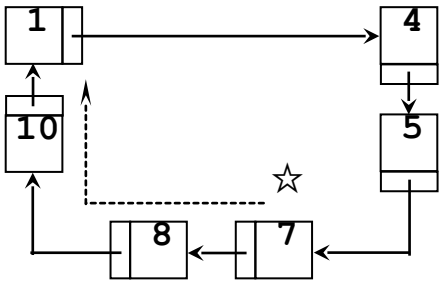
Step 1



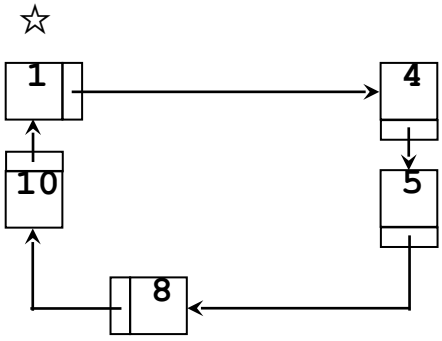
Step 2



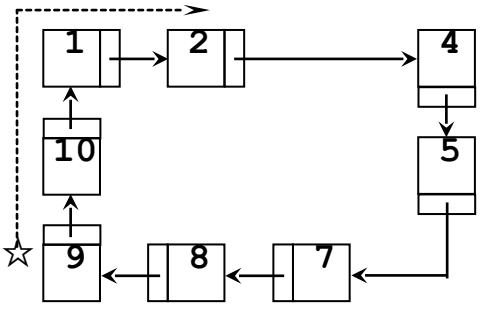
Step 3



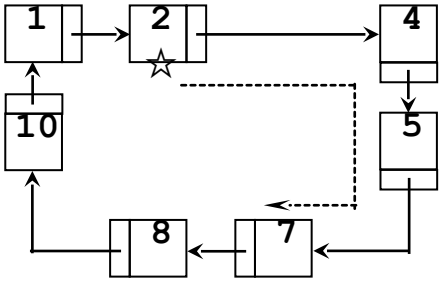
Step 4



Step 5



Step 6



共进行 5×3=15 次操作，时间复杂度 *O(nm)*。

从圆桌问题谈数据结构的综合运用

例．圆桌问题*(AH'99)*

题目： 圆桌上围坐着 $2n$ 个人。其中 n 个人是好人，另外 n 个人是坏人。如果从第一个人开始数数，数到第 m 个人，则立即处死该人；然后从被处死的人之后开始数数，再将数到的第 m 个人处死……依此方法不断处死围坐在圆桌上的人。试问预先应如何安排这些好人与坏人的座位，能使得在处死 n 个人之后，圆桌上围坐的剩余的 n 个人全是好人。

输入： 文件中的每一行都有两个数，依次为 n 和 m ，表示一个问题的描述信息。约束条件： $n \le 32767$ ， $m \le 32767$ 。

输出： 依次输出每一个问题的解。每一个问题的解可以用连续的若干行字符表示，每行字符数量不超过 50。但是在一个问题的解中不允许出现空白字符和空行，相邻的两个问题的解之间用空行隔开。用大写字母 G 表示好人，大写字母 B 表示坏人。

解法：

思想：模拟实际过程，寻找前 n 个被"处死"的人的位置。

1． 普通解法——线性表"查找"法

1 用顺序存储结构实现

用数组记录当前所有未被处死的人原来的位置，初始值为 1..2n。可根据前一个被处死的人在数组中的位置（即下标）直接定位，找到下一个应该被处死的人在数组中的位置，然后删去，并将它后面的元素全部前移一次。

2 用链式存储结构实现

用链表记录当前所有未被处死的人原来的位置，初始值为 1..2n。每处死一个人后，只要将这个结点直接从链表中删去即可，然后指针后移 (m-1) 次，找到下一个应该被处死的人。

2． 改进解法——"优化直接定位"法

总体思想就是在较好地实现"直接定位"的基础上，尽量避免大规模的元素移动。

设计出的数据结构如图 1 所示：其中 group 表示将原来的数据分为几段存储；每一段的开头记下的 amount 值表示此段中现有元素的个数。随程序的运行，amount 值是不断减小的。

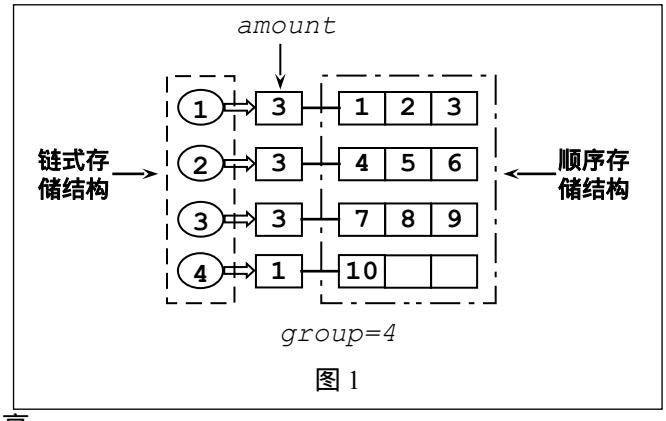


图 1

这种结构可以看作是**链式**存储结构和**顺序**存储结构的结合产物，兼具这两种存储结构的优点。运用了这种存储结构后，程序效率显著提高。

引申

- 横向延伸——其它约瑟夫环问题

  如：《翻牌游戏》、《猴子选大王》

- 纵向延伸——数据结构的综合运用

  在解决一些数据规模较大的题目时有很好的应用。如《隐藏的码字》*（IOI'99）*。在解决这道题目时，如果运用链式和顺序相结合的数据结构（如图 2 所示），程序效率就比较高。

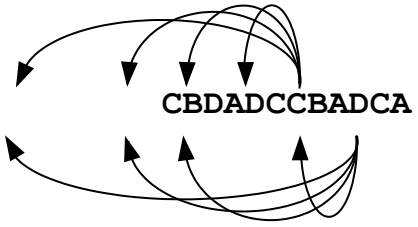


图 2

链式和顺序相结合的数据结构实现简单，效果显著，应用比较广泛。当然还有其它的结合方式，比如二叉堆和顺序结构的一一映射（单射），在解决某些问题时有非常好的效果。

小结

"网络式思维方式的核心是**联系**"。在做题目时，我们应深挖题目所给条件、各种数据结构以及算法之间的联系，这样才能更好地完成题目，并达到提高自己的目的。

这篇论文仅仅是从一类很常见的问题——约瑟夫环（也称 Josephus 排列）问题出发，并由此引申出数据结构的综合运用。对于形式多样的信息学问题来说，数据结构的综合运用只是解题策略中的一个小方面，但是如果我们对待每个问题、算法、数据结构等，都能深入发掘它与其它事物的联系，那么我们就可以自然而然地建立起知识网络，在必要的时候综合运用。而这对于我们的学习、研究将大有帮助。

特别需要强调的是：本文提到"数据结构的综合运用"，这里的综合并不单单是指形式上的，更重要的是指思想（即内涵）的综合。只要在思想上体现出两种或多种数据结构的优点，在操作时发挥出它们的优点，就已经从根本上达到了综合的目的。

从圆桌问题谈数据结构的综合运用

例．圆桌问题*(99 年安徽省赛题)*

题目：圆桌上围坐着 $2n$ 个人。其中 n 个人是好人，另外 n 个人是坏人。如果从第一个人开始数数，数到第 m 个人，则立即处死该人；然后从被处死的人之后开始数数，再将数到的第 m 个人处死……依此方法不断处死围坐在圆桌上的人。试问预先应如何安排这些好人与坏人的座位，能使得在处死 n 个人之后，圆桌上围坐的剩余的 n 个人全是好人。

输入：文件中的每一行都有两个数，依次为 n 和 m ，表示一个问题的描述信息，$n \le 32767$ ，$m \le 32767$ 。

输出：依次输出每一个问题的解。每一个问题的解可以用连续的若干行字符来表示，每行的字符数量不超过 50。但是在一个问题的解中不允许出现空白字符和空行，相邻的两个问题的解之间用空行隔开。用大写字母 G 表示好人，大写字母 B 表示坏人。

解法：

思想：模拟实际过程，寻找前 n 个被"处死"的人的位置（注：此处插入图示 1）

1． 普通解法——线性表"查找"法

1 用顺序存储结构实现

用数组记录当前所有未被处死的人在原来的位置，初始值为 1..2n。可根据前一个被处死的人在数组中的位置（即下标）直接定位，找到下一个应该被处死的人在数组中的位置，然后删去，并将它后面的元素全部前移一次。*（注：此处插入图示 2，并分析优缺点）*

如果我们将找下一个该被处死的人的操作简称为"找点"，将删除一个人后要进行的操作称为"去点"，可以看出：顺序存储结构的优点是"找点"时，可以由现在被处死的人的位置直接计算并在数组中精确定位；而缺点也很明显，就是"去点"时，都需要把它后面所有的元素整体移动一次，时间复杂度为 O(n)。所以应用顺序存储结构，程序的整体时间复杂度是 $O(n^2)$。

2 用链式存储结构实现

用链表记录当前所有未被处死的人在原来的位置，初始值为 1..2n。每处死一个人后，只要将这个结点直接从链表中删去即可，然后指针后移 (m-1) 次，找到下一个应该被处死的人。*（注：此处插入图示 3，并分析优缺点）*

链式存储结构的优点是"去点"时只要修改应该被删除结点的父结点指针指向就可以了；缺点是"找点"时，需要移动 (m-1) 次定位指针，

所以应用链式存储结构，程序的整体时间复杂度是O(nm)。

从哲学角度分析，“找点”和“去点”是存在于程序和数据结构中的一对矛盾。应用顺序存储结构时，“找点”效率高而“去点”效率低；应用链式存储结构时，“去点”效率高而“找点”效率低，这都是由数据结构本身决定的，不会随人的主观意志存在或消失。这就表明“找点”和“去点”的时间复杂度不会同时降为O(1)。我们希望有这样一种数据结构，在实现“找点”和“去点”时，使复杂度降到尽量低，在综合考虑顺序存储结构和链式存储结构的特点之后，我们设想出这样一种数据结构模型（*注：插入图示4“思想模型”*），总体思想就是在较好地实现“直接定位”的基础上，尽量避免大量元素移动。因为小规模的数据移动和指针移动，时间都可以接受，所以从总体上来说，这种数据结构的时间复杂度不会太高。实现时，我们将上面的数据结构模型做了一些小小的变动，并提出改进解法，即“优化直接定位”法。

2. 改进解法——“优化直接定位”法

设计出的存储结构如图所示：其中group表示将原来的数据分为几段存储；每一段的开头记下的amount值表示此段中现有元素的个数。随程序的运行，amount值是不断减小的。（*注：先显示图示4“实际模型”；然后手工删除，伴随讲解*）

“优化直接定位”法较好的体现出“直接定位”的思想，而且由于将所有的结点分为若干段之后，每次删除一个结点后，需要移动的结点数相对而言不是很多，这样就使程序效率大大提高，且m越大，这种效果越明显。

这种分段式数组可以看作是**链式**存储结构和**顺序**存储结构的结合产物，它兼具这两种存储结构的优点。

请注意，我们这里提到“结合产物”借用生物学中的部分思想——子代因为遗传作用而具有亲代的某些特征，同时又因为变异作用而与亲代存在差别（当然，我们希望这种变异总是向着好的方向的）。我们设计出的综合的数据结构应该继承了其“亲代”（即本来的未经变化数据结构）的优点，而摒弃它们的缺点。

运用了这种存储结构后，程序效率显著提高，可参见改进前后程序效率比较的表格。（*注：插入表格*）

引申

（*注：插入“引申”*）

- 横向延伸——约瑟夫环类的问题

  如：《翻牌游戏》、《猴子选大王》

- 纵向延伸——数据结构的综合运用

  在解决一些数据规模较大的题目时有很好的应用。如《隐藏的码字》*（IOI'99）*。在解决这道题目时，如果能建立起链式和顺序相结合的数据结构，程序效率就比较高。

  链式和顺序相结合的数据结构实现简单，效果显著，应用比较广泛。当然还有其它的结合方式，比如二叉堆和顺序结构的一一映射（单射），在解决某些问题时有非常好的效果。

小结

在计算机竞赛中成绩斐然的徐宙同学曾在他的论文《谈网络式思维方式及其应用》中写道"网络式思维方式的核心是**联系**"。在做题目时，我们也应该深挖题目所给条件、各种数据结构以及算法之间的联系，这样才能更好地完成题目，并达到提高自己的目的。

本篇论文仅仅是从一类很常见的问题——约瑟夫环（也称 Josephus 排列）问题出发，并由此引申出数据结构的综合运用。对于形式多样的信息学问题来说，数据结构的综合运用只是解题策略中的一个小方面，但是如果我们对待每个问题、算法、数据结构等，都能深入发掘它与其它事物的联系，那么我们就可以自然而然地建立起知识网络，在必要的时候综合运用。而这对于我们的学习、研究将大有帮助。

最后特别需要强调的是：本文提到"数据结构的综合运用"，这里的综合并不单单是指形式上的，更重要的是指思想（即内涵）的综合。只要在思想上体现出两种或多种数据结构的优点，在操作时发挥出它们的长处，就已经从根本上达到了综合的目的。

中等硬度解题报告

[摘要]中等硬度是 IOI2000 第一试的最后一道题目。这道题主要考察选手的创造性，自创算法正确高效的解决问的能力。本文主要讲述我做这到题目的过程和方法。

[关键字]二分法 随机数

[问题描述]见附件

[问题分析]

算法 1-1：由于每次比较可以得出最大的数和最小的数（虽然不知道那个数最大的），所以可以先求出 1，2，3 号中的最大与最小者，再用它们与 4 号比较得出 1~4 号中的最大最小者，再用这两个数与 5 号比.....。以此类推，可求出 1~n 中的最大与最小者，显然它们不是中等硬度物体，所以将它们去掉，再用上述方法求出剩下 n-2 中的最大与最小者，再去掉，.....，最后剩下的一个数就是中等硬度的物体编号。

这个算法比较的复杂度为 $O(n^2)$,不满足 1449 个物体用 7777 次比较出来的限制。究其原因是因为每次比较利用的信息不够。一次比较三个数，可以知道这三个数的顺序关系（虽然不知道是递增还是递减），**若已知两个数的顺序关系，再与第三个数比较，则可知道第三个数在前两个数的顺序关系下的位置。**算法 1-1 中正是没有利用这个信息，导致复杂度高。所以利用这个排序的观点，得出算法 2-1。

算法 2-1：已知前 m 个物体硬度的顺序关系。将下一个物体用**二分法**插入到适当的位置，得出前 m+1 个物体硬度的顺序关系。以此类推得出 n 个物体的顺序关系，从而知道中等硬度者。参看实例：

Label	1	2	3	4	5
Strength	2	5	4	3	1

插入物体编号	已知顺序	比较	返回结果	得出顺序
		1 2 3	3	1 3 2
4	1 3 2	1 3 4	4	1 ( ) 3 2
5	1 4 3 2	4 3 5	4	(1) 43 2
5	1 4 3 2	1 4 5	1	( ) 1 4 3 2
	5 1 4 3 2			

注：( )表示改物体可能插入的区域。第二行得出结论 4 在 1 和 3 中间。第三行得出结论 5 的位置在 4 的左侧，所以还要比较一次以确定 5 与 1 的位置关系。

这个算法采用二分插入法。二分插入法复杂度为 $O(\log_2 m)$。要插入 n-3 个数。所以总的复杂度比 $n \cdot \log_2 n$ 要小一些。实践证明，当 n=1449 时，平均用 11000 多次比较。（$1449 \times \log_2 1449 \approx 15216$）究其原因是因为这个算法将所有物体的硬度都排了序，其中有些是一定不只中等硬度的，对于它们的排序浪费了比较次数。所以在算法 2-1 的基础上加上剪枝，得出算法 2-2。

算法 2-2：设 2m+1=n。则可知对于一个已全部排好顺序的序列，中等硬度物体左边有 n 个物体，右边也有 n 个物体。也就是说若已知一个物体所在序列中位置的左边(右边)有 n 个以上物体时，则它一定不是所求。所以不用排出它的具体位置，将其插在最右边（左边）

即可。利用这一原理，用先将前 m+1 个物体排序，因为这 m+1 个物体都有可能是所求。当第 m+2 号物体插入后，最两端的物体肯定不是所求。同理当现在已经有 m+3 个物体排好序，则这个序列两端的两个物体一定不是所求。也就是说第 k 个物体只有插在位置（k-m-1）到位置（m+1）这个子序列中，才有可能成为中等硬度者。所以在二分法插入中，若当前插入区域已不在上述区域中，则它的具体位置对结果没有影响，所以直接插入到最左端或最右端。举个简单的实例，当前已将 n-1 个物体排好序，在这 n-1 个物体中只有最中间的两个物体才有可能为中等硬度。我们称其为 A,B（A 在 B 左侧）。按照上述方法，第 n 个物体 C 只需与 A、B 比较一次。若 A 在中间，则 C 的插入区域在 A 的左边，已不再可能区域中，于是将 C 插入到最左端，最终结果是 A。若 B 在中间，同理将 C 插入到最右端，结果为 C。若 C 在中间，则插入到 AB 之间，结果为 C。而不需要象算法 2-1 那样，比较若干次，将 C 插入到正确的位置。第二个剪枝在第一个剪枝的基础上，通过实践发现有相当一部分物体是通过上述剪枝，插入到两端。而当判断出它不属于可能区域时，已经做了 2~5 次判断。所以直接先用 1~2 次比较，判断出插入物体是否在可能区域。若是，则在用二分法插入；若否，则直接插入到两端。这样可以提高一些效率。

这个改进可使 1449 个物体的比较次数平均为 8200，只距 7777 的上限一步之遥。但改进幅度很小，原因是这种排序的方法对前 725 个物体的排序无法剪枝，要用去 5000 次左右的比较。所以要在规定次数内解出此题，只能改变算法，进入 3-X 算法系列。

算法 3-1：虽然前两系列算法没有成功，但我总结了一下，有以下 2 点经验值得借鉴 。1.要有位置概念。虽然不知大小，但要有顺序。2.要有区域概念。先判断出可能区域，再逐步细化。（算法 2-2 就是细化得太早）所以算法是：用 a[1]和 a[2]（a[i]表示可能区域中第 i 个物体的号码是 a[i]。不妨设 a[1]在 a[2]左边）依次与 a[3]~a[n]比较，将所有物体分为三类：一类在 a[1]左边（比 a[1]小或大），一类在 a[1]和 a[2]之间（硬度介于两者之间），一类在 a[2]右边（比 a[2]大或小）。统计个数可知中等硬度物体是 a[1]或 a[2]或三类之一。若是三类之一，再对这一类物体采用相似的方法分解，直到求出结果。之所以称相似的方法，是因为有两点不同：1.不能假设 a[1]在 a[2]左边（这时 a[1],a[2]的值已经改变，a[1]~a[u]分别记录这一类物体的编号）。a[1]与 a[2]的位置关系，要引入上一层分类中的一个基数（上一层的 a[1]或 a[2]），做一次比较，使得 a[1]a[2]的顺序与以前保持一致。2.中等硬度者位置已不是 m+1。具体位置，要根据上一层的中等硬度物体位置及分类结果得出。这个位置用一个参量在递归中传递。最后做一点改进，为防止特殊输入数据。每次所选的基数不再是 a[1]和 a[2]，而是 a[x]、a[y]，x、y 为随机数。

这个算法可以完全解决此问题。

[总结]三个算法的效率比较表，结果为运行 10 次随即产生的输入数据的平均值。：

	2-1	2-2	3-1
N=1449 所用比较次数	11176	8189	3442

通过这道题，我认识到改进一个程序要充分了解到该程序的不足之处，抓住主要矛盾 。并且当看到算法没有太大的改进地步时，也要善于总结经验与其成功之处，为新算法打基础。

[附录]

中等硬度结题报告.doc ----------本文

median1.pas ----------算法 2-1 程序

median2.pas ----------算法 2-2 程序

median3.pas ----------算法 3-1 程序

medmake.pas ----------输入数据生成程序

device.pas ----------交互单元

从一道题目的解法试谈网络流的构造与算法

福建师大附中 江鹏

1. 引论

A. 对网络流算法的认识

网络流算法是一种高效实用的算法，相对于其它图论算法来说，模型更加复杂，编程复杂度也更高，但是它综合了图论中的其它一些算法（如最短路径），因而适用范围也更广，经常能够很好地解决一些搜索与动态规划无法解决的，看似 NP 的问题。

B. 具体问题的应用

网络流在具体问题中的应用，最具挑战性的部分是模型的构造。这没用现成的模式可以套用，需要对各种网络流的性质了如指掌（比如点有容量、容量有上下限、多重边等等），并且归纳总结一些经验，发挥我们的创造性。

2. 例题分析

【问题 1】项目发展规划（Develop）

Macrosoft® 公司准备制定一份未来的发展规划。公司各部门提出的发展项目汇总成了一张规划表，该表包含了许多项目。对于每个项目，规划表中都给出了它所需的投资或预计的盈利。由于某些项目的实施必须依赖于其它项目的开发成果，所以如果要实施这个项目的话，它所依赖的项目也是必不可少的。现在请你担任 Macrosoft® 公司的总裁，从这些项目中挑选出一部分，使你的公司获得最大的净利润。

- 输入

  输入文件包括项目的数量 N，每个项目的预算 Ci 和它所依赖的项目集合 Pi。格式如下：

  第 1 行是 N；

  接下来的第 i 行每行表示第 i 个项目的信息。每行的第一个数是 Ci，正数表示盈利，负数表示投资。剩下的数是项目 i 所依赖的项目的编号。

  每行相邻的两个数之间用一个或多个空格隔开。

- 输出

  第 1 行是公司的最大净利润。接着是获得最大净利润的项目选择方案。若有多个方案，则输出挑选项目最少的一个方案。每行一个数，表示选择的项目的编号，所有项目按从小到大的顺序输出。

- 数据限制

  $0 \le N \le 1000$

  $-1000000 \le Ci \le 1000000$

- 输入输出范例

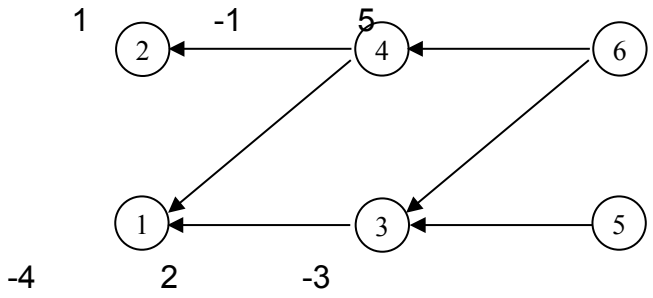
Sample Input	Sample Output
6 -4 1 2 2 -1 1 2 -3 3 5 3 4	3 1 2 3 4 6

【分析解答】

1. 抽象原题（图论模型）

给定包含 N 个顶点的有向图 G =（V, E），每个顶点代表一个项目，顶点有一权值 Ci 表示项目的预算。用有向边来表示项目间的依赖关系，从 u 指向 v 的有向边表示项目 u 依赖于项目 v。

问题：求顶点集的一个子集 V'，满足对任意有向边〈u,v〉∈E，若 u∈V'，则 v∈V'，使得 V'中所有顶点的权值之和最大。



2. 搜索

枚举 V 的所有符合条件的子集，时间复杂度 $O(2^n)$，指数级。无论如何剪枝优化，也摆脱不了非多项式。

3. 动态规划

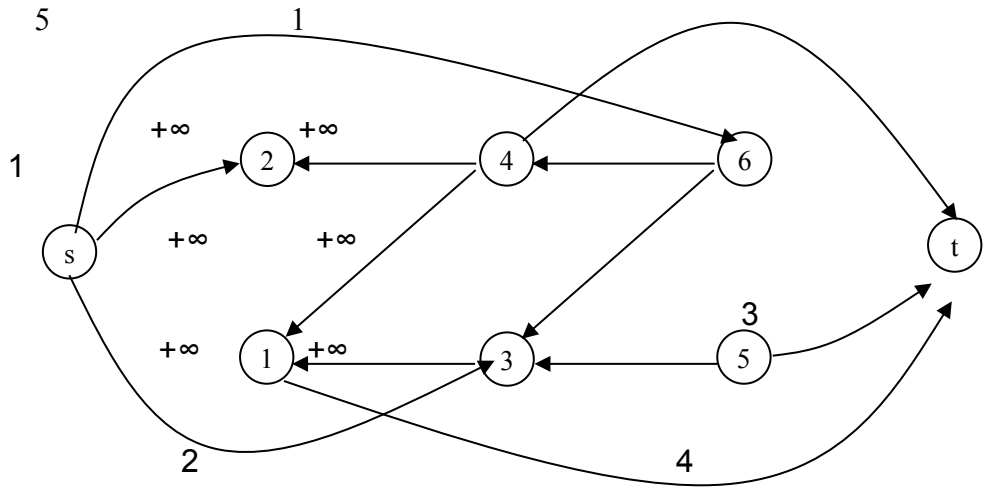
本题的结构是有向无环图，而非树形结构，不适合动态规划。如果一定要做，实质类似于搜索，由于状态数量众多，仍是指数级的时间复杂度。

4. 网络流

流网络的构造方法：

建立 N 顶点代表 N 个项目，另外增加源 s 与汇 t。若项目 i 必须依赖于项目 j，则从顶点 i 向顶点 j 引一条容量为无穷大的弧。对于每个项目 i，若它的预算 C 为正（盈利），则从源 s 向顶点 i 引一条容量为 C 的边；若它的预算 C 为负（投资），则从顶点 i 向汇 t 引一条容量为-C 的边。

求这个网络的最小割（S, T），设其容量 C(S,T) = F。设 R 为所有盈利项目的预算之和（净利润上界），那么 R-F 就是最大净利润；S 中的顶点就表示最优方案所选择的项目。



最小割：S = {s,1,2,3,4,6}；T = {5,t}

C(S,T) = 5　　净利润 R- C(S,T) = 8－5 = 3

证明算法的正确性：

- 建立项目选择方案与流网络的割（S,T）的一一对应关系：

  任意一个项目选择方案都可以对应网络中的一个割（S,T），S={s}+{所有选择的项目}，T=V-S。

对于任意一个不满足依赖关系的项目选择方案，其对应的割有以下特点：

存在一条容量为+∞弧〈u,v〉，u 属于 S 而 v 属于 T。这时割的容量是无穷大，显然不可能是网络的最小割。

- 对于任意一个割（S,T），如果其对应一个符合条件的方案，它的净利润是 R-C(S,T)。导致实际净利润小于上届 R 的原因有：

  1．未选取盈利项目 i，即顶点 i 包含在 T 中，那么存在一条从源 s 至顶点 i 的容量为 Ci 的弧

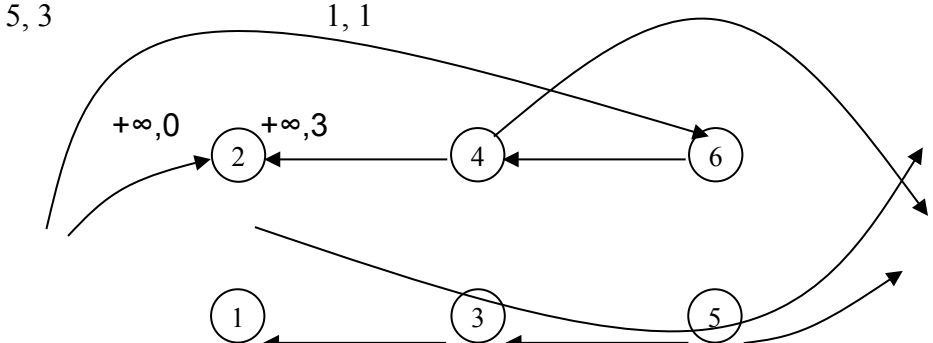
  2．选取投资项目 i，即顶点 i 包含在 S 中，那么存在一条从顶点 i 至汇的容量为－Ci 的弧

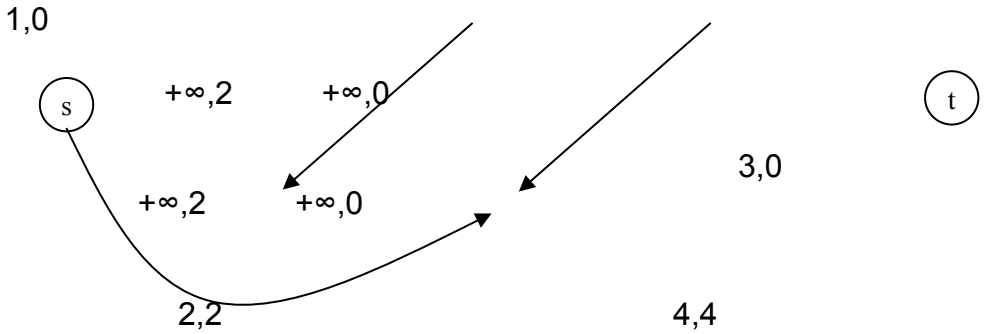
  C(S,T)就是上述两种弧的容量之和。

  综上所述，割的容量越小，方案的净利润就越大。

- 最小割的求法：

  根据最大流最小割定理，网络的最小割可以通过最大流的方法求得。





本题解题的关键在于流网络数学模型的建立。本题建模的独到之处在于：以前的网络流问题通常使用流量表示解答方案，而本题使用割表示解答方案，并充分利用了割的性质，流只是求得最小割的手段。这为我们开辟了一条构造网络流解决问题的新思路。

初看这个问题，要把它和网络流联系起来，有相当的难度。必须熟练地掌握流网络的各种性质，经过反复的类比尝试，才能发现它们之间的共性。

【联想思考】

作为本题的一个衍生，给每个项目估计一个完成时间，并假设公司同时只能进行一个项目。现在的问题是：如何选择一些能在给定时间内完成的项目，使得公司得到最大收益。这个问题我至今还没有找到有效算法，希望有兴趣的同学来共同研究。

3. 编程技巧

◆ 数据结构：邻接表

◆ 直接表示原问题

优点：节省空间

缺点：编程复杂度大，不具有通用性

发言稿

计算几何学是研究几何问题的算法，在现代工程学与数学，诸如计算机图形学、计算机辅助设计、机器人学都要应用计算几何学，在信息学竞赛中几何题也开始出现了，但是在实际的竞赛中，几何题得分率往往是最低的，所以我对几何题的算法进行了一下探索。

任何复杂的算法都是由许多简单的算法组合而成的，计算几何题也同样如此，先来看最基本的算法：

1、求直线的斜率

2、求 2 条直线的交点

3、判断 2 条线段是否相交

4、求叉积等等。

这些都是最基本的算法，是解几何题的基础，任何对基本算法的不熟悉，都可能导致解题的失败，所以熟悉几何题中的基本算法是非常重要的。

但是有了基本算法是远远不够的，因为光靠竞赛时的临时思考，组合算法从时间上来说是来不及的，这就需要熟悉一些经典算法，在竞赛中直接使用，比如：

1、求凸包

2、求最近点对

3、判断点是否在多边形内等等

基本算法和经典算法都是比较简单的，最后我们再来说一下几何题的题型及解几何题的一些技巧，

几何题的几种类型

1、 纯粹的计算求解题

解这一类题除了需要有扎实的解析几何的基础，还要全面地看待问题，仔细地分析题目中的特殊情况，比如求直线的斜率时，直线的斜率为无穷大，求 2 条直线的交点时，2 直线平行，等等。这些都是要靠平时学习时的积累。

2、 存在性问题

这一类问题可以用计算的方法来直接求解，如果求得了可行解，则说明是存在的，否则就是不存在的，但是模型的效率同模型的抽象化程度有关，模型的抽象化程度越高，它的效率也就越高，几何模型的的抽象化程度是非常低的，而且存在性问题一般在一个测试点上有好几组测试数据，几何模型的效率显然是远远不能满足要求的，这就需要对几何模型进行一定的变换，转换成高效率的模型，下面就通过一个例子来对这种方法进行一下阐述。

3、 求几何中的最佳值问题

这类问题是几何题中比较难的问题，一般没有什么非常有效的算法能够求得最佳解，最常用的是用近似算法去逼近最佳解，近似算法的优劣也完全取决于得出的解与最优解的近似程度。

[例 1]游戏者 B 在一张 100*100 纸上确定了一个目标点，游戏者 A 一开始在点(0,0)上，每次游戏者 A 从一个点到另一个点，如果新的点离目标点近了，那么游戏者 B 说“Hotter”，如果新的点离目标远了，那么游戏者 B 说“Colder”，如果距离不变，那么游戏者 B 说“Same”。

输入文件包括很多行，每行包含游戏者 A 这一步到达的点(x,y)和游戏者 B 说的话，对每次游戏者 B 说话，判断目标点可能的位置的面积，精确到小数点后 2 位。

这是一道纯粹的计算求解题，首先证明可能的位置的图形一定是个凸多边形。

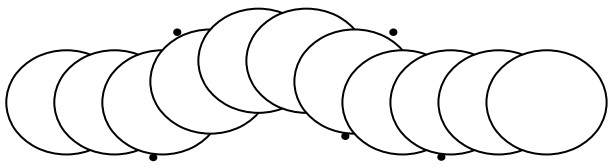
因为每次对游戏者 B 的回答，就可以确定可能的位置在出发点和到达点中垂线的哪一边或就是中垂线，每次的可能图形都是凸多边形。所以这个图形是许多个凸多边形的交集，所以这个图形是凸多边形。

接下来就是解题了，先令多边形为一个四边形，(0,0),(0,100),(100,100),(100,0)，然后对每次游戏者 B 的回答，用这条中垂线将多边形分成 2 部分，取可能的那部分，即可。

不过这样并不是完全正确的，必须考虑到特殊情况，比如游戏者 A 到达的点和这步前的点完全相同，这时就不存在中垂线了，这些都是解题中要注意的重点。

在这道题中就用到了很多解析集合的知识，包括求线段之间的交点，判断点是否在线段的两边，证明最终图形是凸多边形等等。

[例 2]在一个无限长的条形路上，有 n(n<=200)个柱子，体积不计，有一个人想从左边走到右边，人近似看成一个半径为 R 的圆（如右图），问能否实现。



拿到这道题最基本的做法是对从最左边的柱子到最右边的柱子中，每一个竖列进行扫描，计算可走到的范围，如果到最右边的柱子所在的列都有可走到的范围，则有解，否则无解。可是如果最左和最右的 2 个柱子相距非常远，那么这样计算的时间复杂度无疑是非常高的，所以我们应该对这个几何模型进行转化。

首先在这个图形中，不动的是柱子（近似看成点），动的是人（近似看成一个圆），这样处理比较麻烦，所以我们应该先把动的转换成点，圆转换成圆心是最容易想到的，对圆心来说，和柱子的距离不能<R，所以可以把每个柱子转换为以其为圆心，半径为 R 的圆，人转换成他的圆心，这样就使得计算可走到范围容易多了。

不过转换成这个模型后，问题还没有得到根本的解决，必须进一步的转换。

前 2 个算法有一个公共的特点，就是计算的都是圆外的部分，而计算圆内的部分的连通性显然比计算圆外部分来得简单，所以我们现在的目标就是把圆外部分换成圆内部分。

因为左右方向上是无穷长的，所以如果左右部分在圆外相通的话，那么上下两条直线在圆内部分就是不相通的，反之如果左右部分在圆外不相通的话，那么上下两条直线在圆内部分就是相通的，

所以我们可以将对每一个竖列的扫描转换成对每一个横行的扫描，而且又是在圆内操作，效率大大提高了。

但是前面的转换，对模型的抽象化程度却一点也没有改进，如何在这方面进行改进无疑是最关键的。

分析圆的特性，任意 2 个属于同一个圆的点必定是相通的，这就启发我们利用圆的特性，把难以处理的区域转换成几个具有代表性的点，使得能够完全表示出区域连通的特性来，到了这里，应该很容易就可以看出，取每个圆的圆心是最好不过了，因为每个圆的大小完全相同，不存在包含，相切（如果内切，就是重合了，如果外切，就是中间不连通的）等等复杂的关系，只有相交和相离的关系，而且如果 2 个圆之间相交的话，那么这 2 个圆就是相通的，可以在这 2 个圆的圆心之间连一条边，增加一个源点，与上边有交点的圆和源点连一条边，增加一个汇点，与下边有交点的圆和汇点连一条边，这样就把一道几何题完全转换成了一道图论题，只要判断源点和汇点之间是否有路就可以了，这是一道非常经典的图论题，解法就不说了。

从上面这道题的解题过程中，可以得出这样的结论：解这一类的几何题必须充分了解几何变换，挖掘题目中隐含的线索，对题目的本质进行充分的探索，才能做好几何题。

[例 3]一个农夫在一个 x*y 的矩形田地上放牧 n 头奶牛（n<=25），它们互相之间都非常仇视，所以都希望能够离其它奶牛尽量的远，它们有它们自己的标准，就是离其它奶牛的距离的倒数和越小越好，因为农夫想让它们尽量高兴，所以他必须找到一种使所有的奶牛之间的距离的倒数和尽量小，不过学历不高的农夫觉得自己很难做到，请你来为他找到这种方案，他将按照方案的解和最优解的差距来决定付给你的酬劳的多少（他知道最优解还叫你求什么呢？；））

输入 x,y,n，输出你的解的 n 头奶牛的位置。

显然这是一道求几何最值的问题，而且显然是应该用近似算法来做，那么决定解决问题质量的就是近似算法的优劣。

最简单的想法就是：既然 n 最多也只不过是 25，就来个“没有功劳，也有苦劳”吧，对在正方形上手算得到的比较好的解，按照比例放到矩形里去，比如 n=4 时，正方形上的解是四个角，而一般（x,y 之间相差不大）的矩形，这也就是最优解了。

但是这种算法在 x,y 之间差距比较大的时候，就充分显示出它的不足了，和最优解的差距非常大，几乎 1 分也得不到了，所以这种算法并不是一种非常好的方法，只能够混混而已。

在求最优解的题目中贪心法是很难有用武之地，但是贪心法是一种非常常用的近似算法，所以解这道题目，可以用贪心法一试。

第一个点取(0,0)，以后每个点都取可以使当前的倒数和最大的点，当然这里要用到逐步求精法，这样可以得到一个比较接近最优解的方案，经过分析，也很难找到可以使贪心法得到离最优解较远的数据，所以这道题中贪心法完全是一种有效的算法。

不过贪心法终究和最优解有一定距离，并不能得到所有的分数，所以需要找到另一种更加有效的算法，由于这道题没有固定的最佳解法，所以，对任何固定算法应该都有使其得不到满分的数据，所以我们又想到了非固定算法：随机化算法。

这个算法在这里我就不多介绍了，留给大家自己去思考：）

附录：程序题解

第一题：

```
program fat;
  const inputname='fat.in';
        outputname='fat.out';
  var f1,f2:text;
      x,y:array[1..100]of real;
      b:array[0..101,0..101]of boolean;
      i,j,k,n:integer;
      l,r:real;

  function dis(x1,y1,x2,y2:real):real;{求 2 点间距离的平方}

    begin
      dis:=sqr(x1-x2)+sqr(y1-y2);
    end;
  begin
    assign(f1,inputname);
    reset(f1);
    assign(f2,outputname);
    rewrite(f2);

    readln(f1,l,r);{读入输入数据}

    readln(f1,n);
    for i:=1 to n do
      readln(f1,x[i],y[i]);

    fillchar(b,sizeof(b),false);{转换成图论模型}

    for i:=1 to n do
      begin
        if y[i]<=r then
          begin
            b[0,i]:=true;
            b[i,0]:=true;
          end;
        if y[i]>=l-r then
          begin
            b[i,n+1]:=true;
            b[n+1,i]:=true;
          end;
      end;
    r:=4*r*r;
    for i:=1 to n do
      for j:=i+1 to n do
        if dis(x[i],y[i],x[j],y[j])<=r then
```

```
        begin
          b[i,j]:=true;
          b[j,i]:=true;
        end;
    for k:=0 to n+1 do
      for i:=0 to n+1 do
        for j:=0 to n+1 do
          if b[i,k] and b[k,j] then b[i,j]:=true;{求传递闭包}
    if b[0,n+1] then writeln(f2,'Not possible')
      else writeln(f2,'Possible');
    close(f1);
    close(f2);
  end.
```

第二题：

```
program game;
  const inputname='game.in';
        outputname='game.out';
  var f1,f2:text;
      i,j,t,f:integer;
      a:array[1..100,1..2]of real;
      d:array[1..100]of integer;
      xp,yp,x,y,xi1,xi2,xi3,zhi,x3,y3,x4,y4,sum:real;
      s:string;
  function sq(x1,y1,x2,y2,x3,y3:real):real;{求以(x1,y1),(x2,y2),(x3,y3)为
顶点的三角形的面积}
    begin
      sq:=abs(x1*y2+x2*y3+x3*y1-x1*y3-x2*y1-x3*y2)/2;
    end;
  begin
    assign(f1,inputname);
    reset(f1);
    assign(f2,outputname);
    rewrite(f2);
    xp:=0;yp:=0;t:=4;{初始化多边形}
    a[1,1]:=0;a[1,2]:=0;
    a[2,1]:=0;a[2,2]:=100;
    a[3,1]:=100;a[3,2]:=100;
    a[4,1]:=100;a[4,2]:=0;
    repeat
```

```
read(f1,x,y);
readln(f1,s);
while s[1]=' ' do delete(s,1,1);
if s='Same' then
  begin
    t:=0;
    continue;
  end;
 if (x=xp)and(y=yp) then continue;{考虑特殊情况：游戏者 A 没有移动}
i:=0;
xi1:=2*(x-xp);
xi2:=2*(y-yp);
xi3:=xp*xp+yp*yp-x*x-y*y;{求出中垂线的方程}
if xi1*x+xi2*y+xi3>0 then f:=1
  else f:=-1;
if s='Colder' then f:=-f;
for i:=1 to t do{判断每个点在中垂线的哪一侧}
  begin
    zhi:=xi1*a[i,1]+xi2*a[i,2]+xi3;
    if zhi>0 then d[i]:=1
      else if zhi<0 then d[i]:=-1
        else d[i]:=0;
  end;
a[t+1]:=a[1];d[t+1]:=d[1];
i:=1;
while i<=t do{增加多边形和中垂线的交点}
  begin
    if d[i]*d[i+1]<0 then
      begin
        for j:=t+1 downto i+1 do
          begin
            a[j+1]:=a[j];
            d[j+1]:=d[j];
          end;
        x3:=a[i,1];y3:=a[i,2];
        x4:=a[i+2,1];y4:=a[i+2,2];
        if x3=x4 then
          begin
```

```
              a[i+1,1]:=x3;
              a[i+1,2]:=-(xi3+xi1*a[i+1,1])/xi2;
            end
            else begin
              a[i+1,1]:=(x3*xi2*(y4-y3)-y3*xi2*(x4-x3)-xi3*(x4-x3))/(xi1*
(x4-x3)+xi2*(y4-y3));
              a[i+1,2]:=(y4-y3)/(x4-x3)*(a[i+1,1]-x3)+y3;
            end;
          d[i+1]:=0;
          inc(t);
        end;
      inc(i);
    end;
  i:=1;
  while i<=t do{删除不符合要求的点}
    begin
      if d[i]=-f then
        begin
          for j:=i+1 to t do
            begin
              a[j-1]:=a[j];
              d[j-1]:=d[j];
            end;
          dec(t);
        end
        else inc(i);
    end;
  sum:=0;
  for i:=2 to t-1 do{求出多边形的面积}
    begin
      sum:=sum+sq(a[1,1],a[1,2],a[i,1],a[i,2],a[i+1,1],a[i+1,2]);
    end;
  writeln(f2,sum:0:2);
  xp:=x;yp:=y;
 until eof(f1);
 close(f1);
 close(f2);
end.
```

第三题:贪心法

```
program enemy;
 const inputname='enemy.in';
```

```
    outputname='enemy.out';
var f1,f2:text;
   d:array[1..25,1..2]of real;
   i,j,k,l,x,y,n,bj,bk:integer;
   x1,y1,r,x2,y2,bs,sum,xz,yz:real;
   b1:boolean;
function dis(x1,y1,x2,y2:real):real;{求(x1,y1),(x2,y2)间的距离}
 begin
  dis:=sqrt(sqr(x1-x2)+sqr(y1-y2));
 end;
begin
 assign(f1,inputname);
 reset(f1);
 assign(f2,outputname);
 rewrite(f2);
 readln(f1,x,y,n);
 d[1,1]:=0;
 d[1,2]:=0;
 if n>=2 then
  begin d[2,1]:=x;d[2,2]:=y;end;
 for i:=3 to n do{依次贪心 n 头奶牛的位置,使当前倒数和最小}
  begin
   x1:=x/2;y1:=y/2;r:=100;
   xz:=x/4;yz:=y/4;
   while (xz>0.002)or(yz>=0.002) do{对整个田地进行分割求解，找
寻当前最佳位置}
    begin
     bs:=1000000;
     bj:=0;bk:=0;
     for j:=-2 to 2 do
      begin
       x2:=x1+j*xz;
       if (x2>x)or(x2<0) then continue;
       for k:=-2 to 2 do
        begin
         y2:=y1+k*yz;
         if (y2>y)or(y2<0) then continue;
         b1:=true;
         for l:=1 to i-1 do
          if (x2=d[l,1])and(y2=d[l,2]) then
```

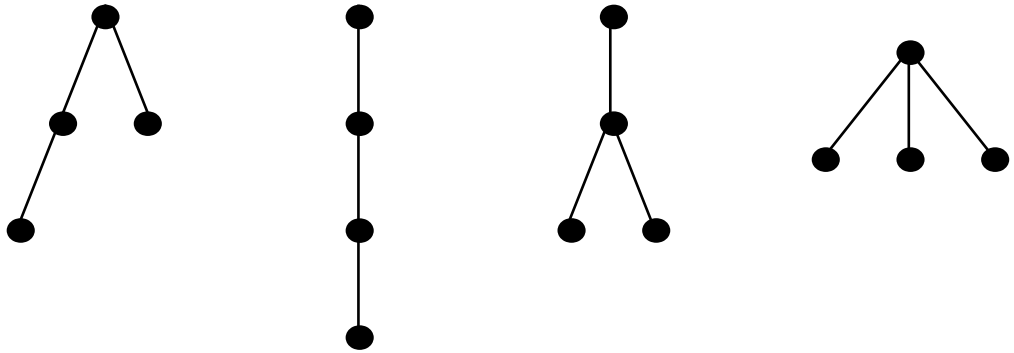
```
                begin
                  b1:=false;
                  break;
                end;
              if not b1 then continue;
              sum:=0;
              for l:=1 to i-1 do
                sum:=sum+1/dis(d[l,1],d[l,2],x2,y2);
              if sum<bs then
                begin
                  bs:=sum;
                  bj:=j;bk:=k;
                end;
            end;
          end;
        x1:=x1+bj*xz;
        y1:=y1+bk*yz;
        xz:=xz/4;yz:=yz/4;
      end;
    d[i,1]:=x1;d[i,2]:=y1;
  end;
 for i:=1 to n do
  writeln(f2,d[i,1]:0:2,' ',d[i,2]:0:2);
 close(f1);
 close(f2);
end.
```

树的枚举

江苏省常州高级中学　李源

引子

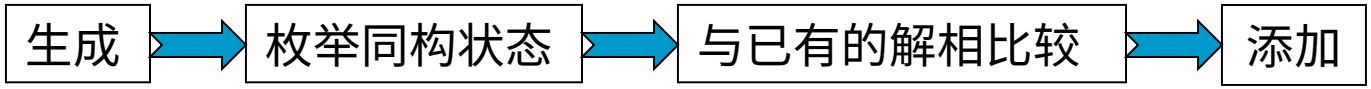
- 树，在计算机算法中是非常重要的非线形结构。即使撇开树的其他广泛应用不说，单单对树本身的形态进行思考与研究，也是一个十分有趣，且具有挑战性的过程。



4 个结点的树（有向树）

枚举算法

- 常规的搜索加判重的做法：



- 下面我们就来看一种**不重复地**生成**所有**确定结点数和深度的有向树的构造性算法。
  - **不重复性：树的大小定义**
  - **不遗漏性：树的变换算法**

树的大小定义

- 我们对树的“大小”作一个规定，使不同树结构之间的关系有序化。

- 假设当前要比较树 A 与树 B 的大小（树 A 与树 B 的子树也都要按照下述的大小关系递归地从大到小排序）。

  比较过程为：

  若 **A 的深度大于 B，**则 **A>B；**若 **A 的深度小于 B，**则 **A<B**；

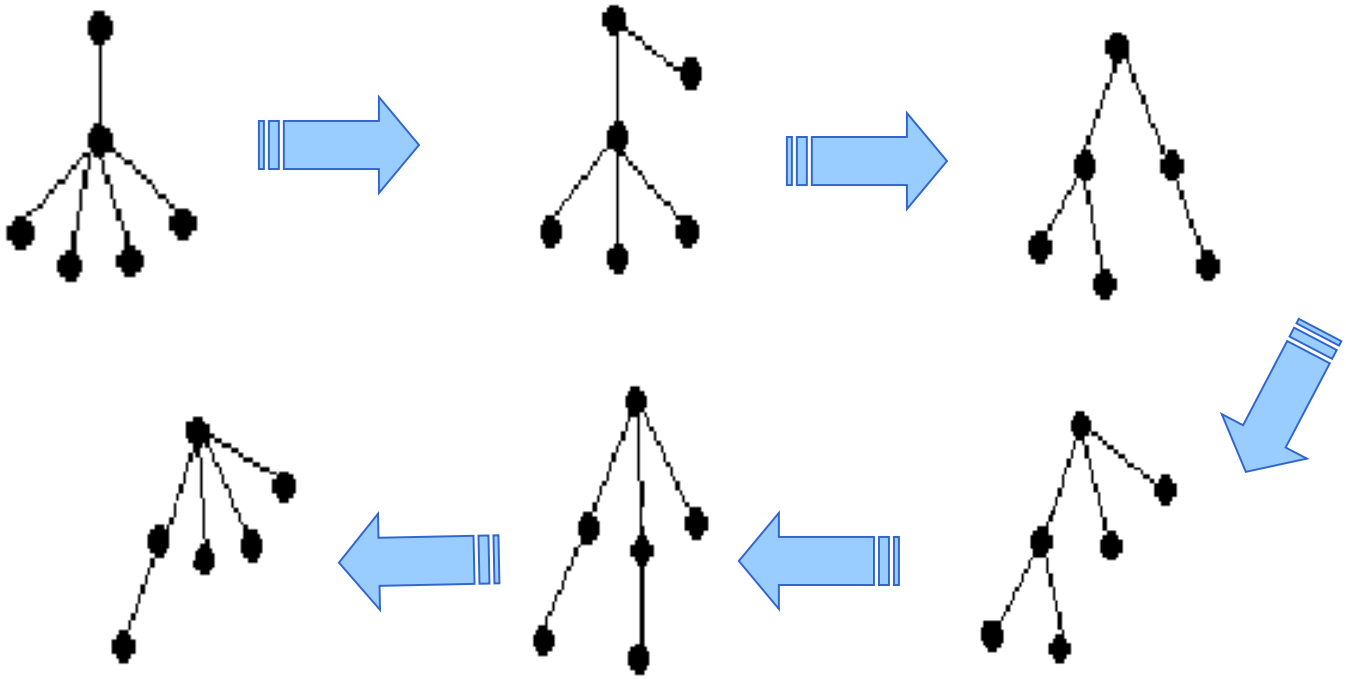
  若 **A** 与 **B 深度相等，视 A** 与 **B 的结点数：**

  若 **A 的结点数大于 B，**则 **A>B；**若 **A 的结点数小于 B，**则 **A<B**；

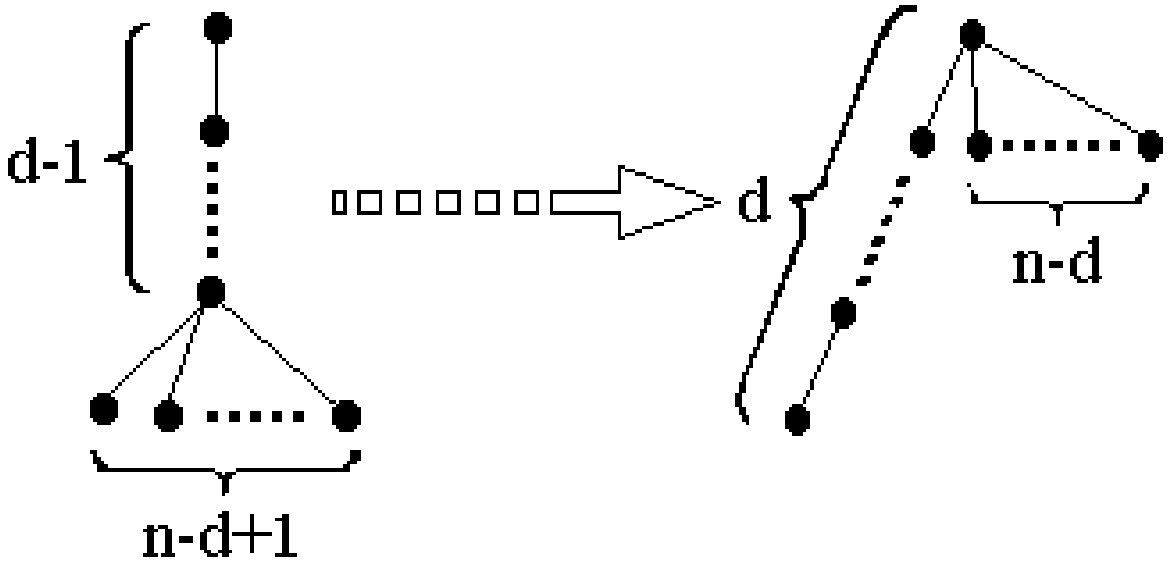
  若 **A** 与 **B 的结点数相等，依次讨论 A** 与 **B 的子树：**

  拥有较大子树的树较大。若当前讨论的子树相等，则讨论 A 与 **B 的下一棵子树。**

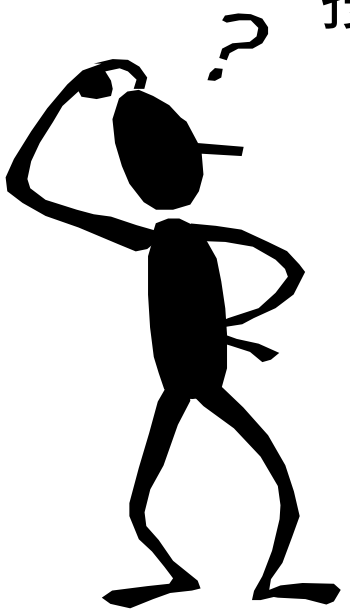
- 现在回到枚举有向树的问题上来：对于深度为 d ，结点数为 n 的所有形态的有向树，根据上面的比较规则，可以把它们从大到小排成一个序列。举 d=3， n=6 为例，这个序列是：



- 进一步地，对于任意的 n 和 d ，在相应序列中的第一棵树和最后一棵树的形态是容易确定的，如下：

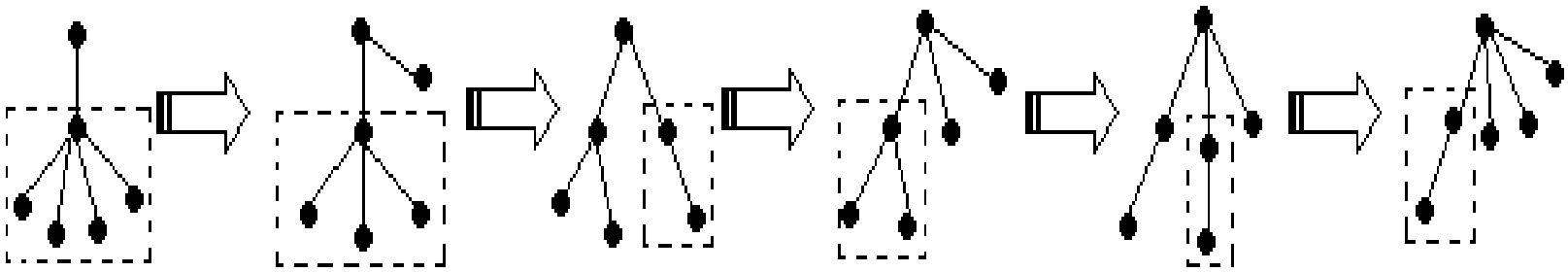


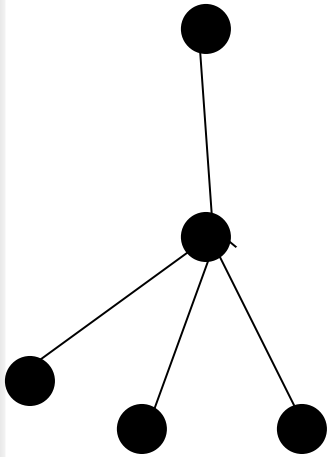
- 现在问题就转化为，根据上述的大小定义，能否找到一种变换的规则，使根据这种规则，可以从最小的一棵树开始，连续地、不遗漏不重复地生成接下来的所有不同形态的树？



变换过程

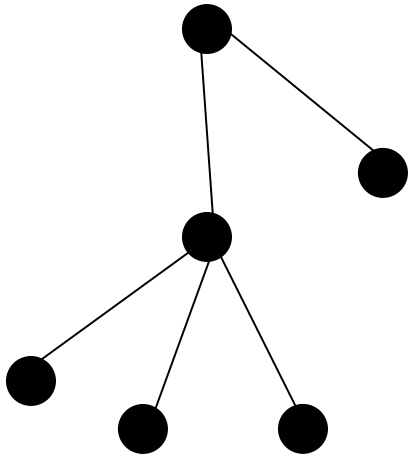
- 首先，从树的序列中的一个形态变换到另一个形态，是一个变小的过程。
- 在这个变换过程中，至少有一棵子树被变小了，变小的过程是通过夺走它的一个子结点实现的（我们用一个虚线框来表示被变小的子树）。





- 结点被夺去；

- 排在它后面的某些子树将会得到这个被夺走的结点，或被重新组合。



- 它们会适当地变大，然而由于它们在有向树大小比较的过程中的地位比先前被变小的那棵子树要低，于是，总的说来，整个有向树就被变小了。

- 过程 I　寻找被删去结点
- 过程 II　将被删的结点重组到后面的子树中

过程Ⅰ　寻找被删去结点

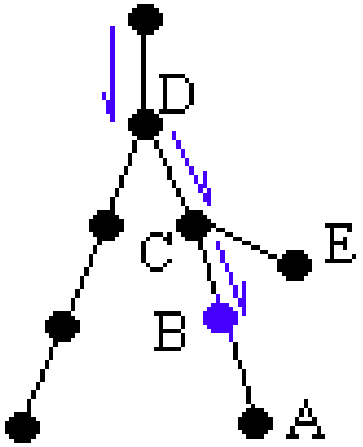
- 在选择被删去的结点时，其所在子树的变小对于整棵树的影响必须尽量小。使它经过变换后恰好可以成为**序列中紧邻它的一棵树**而不是跳跃性的变换。所以：

- 首先，这个被夺取的结点必然为叶结点。

- 第二，对于并列的若干子树，应当从后向前查找。

过程 I 算法

- 从根出发，在子结点中从后向前找到第一个非叶结点的结点，继续搜索直到到达一个子结点均为叶子的结点为止。

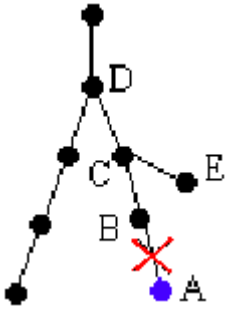


是否可认为该结点的最后一个子结点为待删除结点呢？事实上仍需进一步处理：

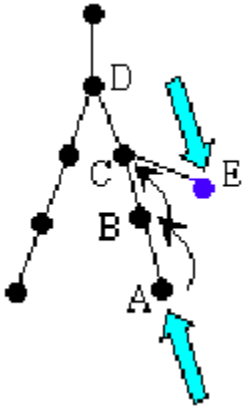
假如，找到的待删除结点 A 为其父亲 B 唯一的子结点，也就意味着如果将 A 从它的父结点 B 删除，那么以 B 为根的那棵子树的深度将会减少 1……

过程 I 算法

- 在对树的大小定义中，深度比结点数更优先。所以，在选择待删除结点时，应该**尽量选择删除后不影响子树深度的结点**优先处理。
- 因此，在找到 A 结点后，必须沿其父亲结点进行回溯，直到当前结点以下的子树的深度不因删除 A 而改变为止（在上图中直到 D 结点），在回溯的过程中，如果经过某一结点，它还有另一个子结点（比如上图中的 C，它还有另一个子结点 E），那么就舍弃原来找到的结点 A，把 E 定为待删除的结点。



如果将 A 从 B 断开，则以 B、C 为根的子树的深度都会受到影响。



C 有子结点 E，E→Target

回溯

找到 A，A→Target

过程 II　将被删的结点重组到后面的子树中

- 在找到该结点并删除之后，又需要进行一系列的处理以形成一棵新的树。在建立新树的时候要强调的一点就是，“要使整棵树变小，但变小的幅度必须达到最小”。

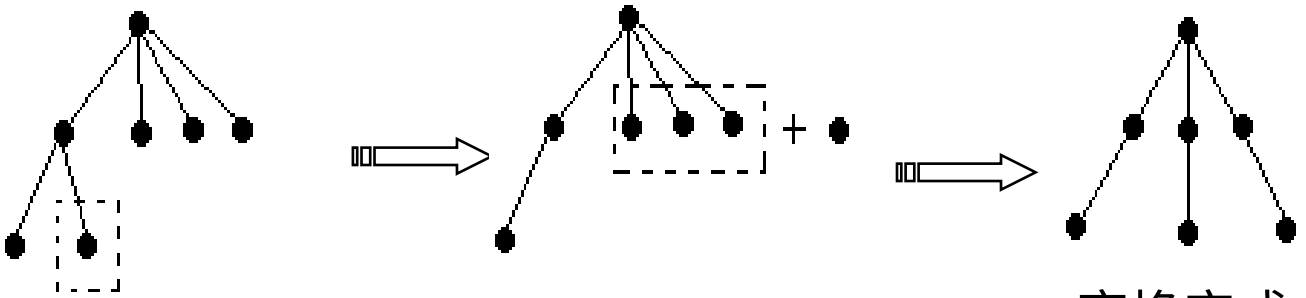
删除一个结点之后，会出现两种情况：

- **当前子树深度不变，结点数变小；**
- **或者子树深度变小。**

下面就对这两种情况分别进行讨论。

过程 II 算法　删除结点后子树深度不变

- 接下来，对排列在被改动的这棵子树以后的其它子树进行重新组合，使它们在满足不大于当前这棵子树的情况下变得最大（同时将被删除的那个结点加入）。



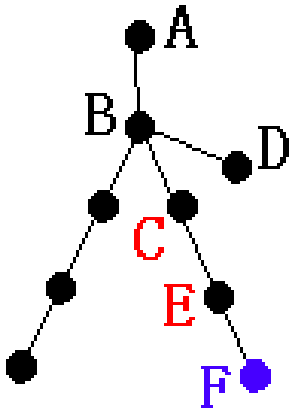
删除结点，使子树变小

将后面的子树重组（最大化）

变换完成

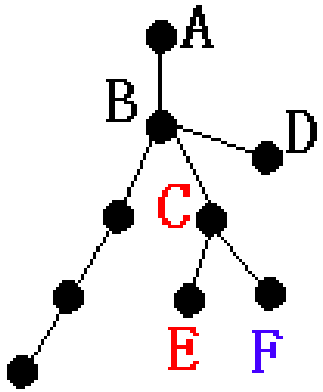
过程 II 算法　删除结点后子树深度变小

- 对于删除结点后子树深度改变的，则要进行如下的处理（举例来说）：

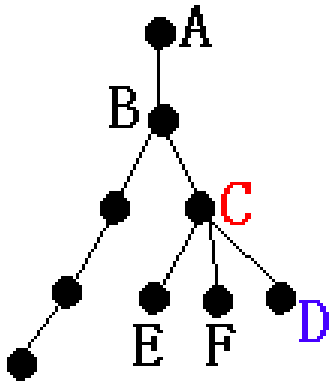


F：要从父亲处删除的结点。

CE：深度将会变小。要进一步处理。



将 F 删去，处理 E 极其兄弟（此处为空），连同删去的 F，以 C 为根进行重组。



处理 C 极其兄弟（D），连同 C 以下的所有子树，以 C 的父亲 B 为根进行重组。得到新的树。

- 完整的有根树枚举算法大致可以如下构成：

1 读入 d,n

2 找第一棵树（最大的）

3 while 未全部生成 do

{ 这步判断可以用计数算法得到的总数来判断，也可以先求得最小的一棵树用来判断 }

4 找到待删除的结点 target;

5 删除 target;

6 对树进行变换；{ 包括上面的两种情况： 子树深度改变， 以及深度未改变 }

7 end of while

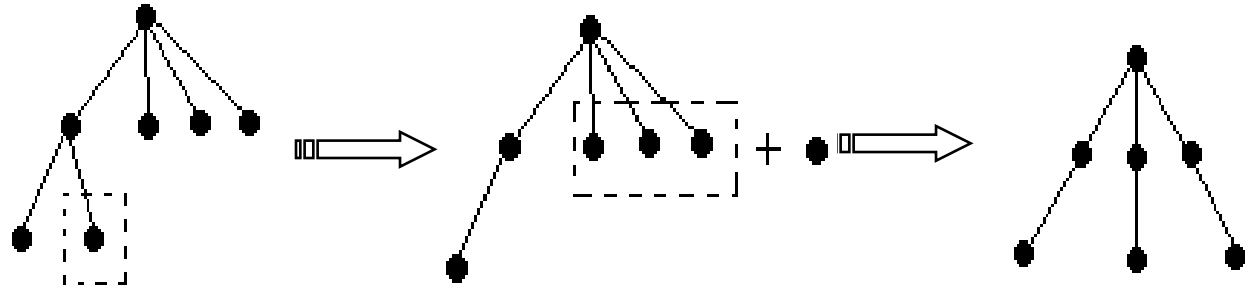
小结

- 虽然上面变换过程看似十分复杂，实质上它是以一种简洁和严谨的规律为基础的。

- 在仔细的研究中，大家可以体会到变换过程的和谐：它和自然数的递减与退位还颇有相似之处。

  比如说：为了使 2000 成为比它小，但又与它相差最少的自然数：

- 类似地，再看一个有向树变换的例子：



删除结点，使子树变小　　将后面的子树重组（最大化）　　变换完成

- 我们先从后向前找到一个待删除结点，删除它之后，然后对其它子树进行了一定的变换（类似于上面把 0 变成为 9 的过程），确保了整棵树变小的程度最小，从而得到了序列中的下一棵有向树。

- 以上介绍的算法可以实现给定深度 d ，结点数 n ，按照从大到小的顺序变换生成所有形态的有向树。算法中涉及的树的复制，以及遍历等，复杂度均为 O （ n ），并且在树的生成过程中完全没有重复生成，所以整个算法的耗时主要与不同形态树的总数有关。

- 该算法最直接的应用是“无根树”问题。下表可以作为算法时间复杂度的直观参考。

我们可以用按照枚举算法编写的生成程序与搜索算法作一个比较（测试环境： PIII 500MHz，192Mb RAM，Borland Pascal 7.0，用时单位： s）：

N	11	12	13	14	15	16	17	18	19	20
构造用时	0.05	0.05	0.11	0.16	0.49	1.21	3.19	8.52	23.08	62.91
搜索用时	0.27	0.71	2.09	6.04	17.69	51.92	152.20	-	-	-

搬运工问题的启示

重庆外语学校 刘汝佳

前言

“搬运工”是一个十分流行的单人智力游戏，玩家的任务是在一个仓库中操纵一个搬运工人将 N 个相同的箱子推到 N 个相同的目的地。不清楚规则不要紧，玩一玩附件里的 SokoMind 就知道了。我在里面加上了标准的测试关卡 90 关，幼儿关卡 61 关和文曲星 170 关，在以后的介绍中，我们就用这些关来测试我们的程序，因此我建议你自己先试一试，看看你能过多少关：）

因为本文内容不算少，在这里我先给大家提供一个小小的阅读建议。初学的朋友或者不知道 IDA*算法的朋友一定要从第一章开始阅读并弄懂，因为它是全文的基础。第二章很好懂，大家只需要做一个了解，知道搬运工问题的难点在哪里，为什么我们选择了 IDA*算法。急于看程序的朋友可以先看看第三章，我们的第一个版本 S4-Baby 诞生了，接着是一系列的改进措施，喜欢人工智能的朋友不妨认真看看，也许会找到灵感哦！最后是总结，和第一章一样的重要，不要错过了。

我假定本文的读者已经对相关知识有一定了解，所以一般不给出很严格的定义或者论证，只是粗略的提一下，语言尽量做到通俗易懂。但由于我的水平实在有限，错误之处一定不少，恳请大家批评指正：）

欢迎大家和我联系。Email:liurujia@163.net , OICQ:2575127。

搬运工问题的启示

重庆外语学校 刘汝佳

【关键字】搬运工问题，人工智能搜索，IDA*

【摘要】本文讨论了一个有趣又富有挑战性的问题：搬运工问题。文章从状态空间搜索的基本知识开始讨论，根据搬运工问题的特点选择了 IDA*算法，并做了初步改进。本文的主要部分讨论了让程序智能化的几个方法 – 下界估计的改进，死锁判断，合适的任务分解与合并，模式搜索已经随机化实验，最后粗略的介绍了一些前面没有提到的想法，并做了总结。

目录

搬运工问题的启示

重庆外语学校 刘汝佳

一 状态空间搜索基本知识

1.状态空间(state space)

对于一个实际的问题，我们可以把它进行一定的抽象。通俗的说，状态(state)是对问题在某一时刻的进展情况的数学描述，状态转移(state-transition)就是问题从一种状态转移到另一种（或几种）状态的操作。如果只有一个智能体(Agent)可以实施这种状态转移，则我们的目的是单一的，也就是从确定的起始状态(start state)经过一系列状态转移而到达一个（或多个）目标状态(goal state)。

如果不止一个智能体可以操纵状态转移(例如下棋)，那么它们可能会朝不同的，甚至是对立的目标进行状态转移。这样的题目不在本文讨论范围之内。

我们知道，搜索的过程实际是在遍历一个隐式图，它的结点是所有的状态，有向边对应于状态转移。一个可行解就是一条从起始结点出发到目标状态集中任意一个结点的路径。这个图称为状态空间(state space)，这样的搜索就是状态空间搜索(Single-Agent Search)

2.盲目搜索(Uninformed Search)

盲目搜索主要包括以下几种：

纯随机搜索(Random Generation and Random Walk)

听起来比较“傻”，但是当深度很大，可行解比较多，解的深度又不重要的时候还是有用的，而且改进后的随机搜索可以对付解分布比较有规律（相对密集或平均，或按黄金分割比例分布等）的题目。一个典型的例子是：你在慌乱中找东西的时候，往往都是进行随机搜索。

广度优先搜索(BFS)和深度优先搜索(DFS)

大家都很熟悉它们的时间效率，空间效率和特点了吧。广度优先搜索的例子是你的眼镜掉在地上以后，你趴在地板上找：）- 你总是先摸最接近你的地方，如果没有，在摸远一点的地方...深度优先搜索的典型例子是走迷宫。它们还有逆向和双向的搜索方式，但是不再本文讨论范围之内。

重复式搜索

这些搜索通过对搜索树扩展式做一些限制，用逐步放宽条件的方式进行重复搜索。这些方法包括：

重复式深度优先(Iterative Deepening)

限制搜索树的最大深度 Dmax,然后进行搜索。如果没有解就加大 Dmax 再搜索。虽然这样进行了很多重复工作，但是因为搜索的工作量与深度成指数关系，因此上一次（重复的）工作量比起当前的搜索量来是比较小的。这种方法适合搜索树总的来说又宽又深，但是可行解却不是很深的题目（一般的深度优先可能陷入很深的又没有解的地方，广度优先的话空间又不够）

重复式广度优先(Iterative Broadening)

它限制的是从一个结点扩展出来的子节点的最大值 Bmax,但是因为优点不是很明显，应用并不多，研究得也比较少。

柱型搜索(Beam Search)

它限制的是每层搜索树节点总数的最大值 Wmax。显然这样搜索树大小与深度成正比，但是可能错过很接近起点的解，而增加 Wmax 的时候保留哪些节点，Wmax 增加多少是当前正在研究的问题。

3.启发式搜索(Informed Search)

我们觉得一些问题很有“想头”，主要是因为启发信息比较多，思考起来容易入手，但是却不容易找到解。我们不愿意手工一个一个盲目的试验，同样也不愿意我们的程序机械的搜索。也就是说，我们希望尽可能的挖掘题目自身的特点让搜索智能化。下面介绍的启发式搜索就是这样的一种智能化搜索方法。

在刚才的那些算法中，我们没有利用状态本身的信息，只是利用了状态转移来进行搜索。事实上，我们自己在解决问题的时候常常会估计状态离目标到底有多接近，进而对多种方案进行选择。把这种方法用到搜索中来，我们可以用一个状态的估价函数来估计它到目标状态的距离。这个估价函数是和问题息息相关的，体现了一定的智能。为了以后叙述方便，我们先介绍一些记号：

S	问题的任何一种状态
H*(s)	s 到目标的实际（最短）距离－可惜事先不知道：）
H(s)	s 的启发函数－s 到目标距离的下界，也就是 h(s)<=h*(s),如果 h 函数对任意状态 s1 和 s2,还满足 h(s1)<=h(s2)+c(s1,s2)（其中 c(s1,s2)代表状态 s1 转移到 s2 的代价）,也就是状态转移时，下界 h 的减少值最多等于状态转移的实际代价，我们说 h 函数是相容(consistent)的。(其实就是要求 h 不能减少得太快)
G(s)	到达 s 状态之前的代价，一般就采用 s 在搜索树中的深度。
F(s)	s 的估价函数，也就是到达目标的总代价的估计。直观上，应该 有 f(s)=g(s)+h(s)，即已经付出的和将要付出的代价之和。如果 g 是相容的，对于 s1 和它的后辈节点，有 h(s1)<=h(s2)+c(s1,s2) 两边同时加上 g(s1),有 h(s1)+g(s1)<=h(s2)+g(s1)+c(s1,s2),也就

	是 f(s1)<=f(s2)。因此 f 函数单调递增。

表 1　启发式搜索用到的符号

贪心搜索(Best-First Search)

象广度优先搜索一样用一个队列储存待扩展，但是按照 h 函数值从小到大排序(其实就是优先队列)。显然由于 h 估计的不精确性，贪心搜索不能保证得到的第一个解最优，而且可能很久都找不到一个解。

A*算法

和贪心搜索很类似，不过是按照 f 函数值进行排序。但是这样会多出一个问题：新生成的状态可能已经遇到过了的。为什么会这样呢？由于贪心搜索是按照 h 函数值排序，而 h 只与状态有关，因此不会出现重复，而 f 值不仅状态有关，还与状态转移到 s 的方式有关，因此可能出现同一个状态有不同的 f 值。解决方式也很简单，如果新状态 s1 与已经遇到的状态 s2 相同，保留 f 值比较小的一个就可以了。(如果 s2 是待扩展结点，是有可能出现 f(s2)>f(s1)的情况的，只有已扩展结点才保证 f 值递增)。A*算法保证得到最优解，但是所用的空间是很大的，难以适应我们的搬运工问题。

IDA*算法

既然 A*算法存在空间问题，那么我们能不能借用深度优先搜索的空间优势，用重复式搜索的方式来缓解危机呢？经过研究，Korf 于 1985 年提出了一个 Iternative Deepening A*(IDA*)算法，比较好的解决了这一问题。一开始，我们把深度最大值 Dmax 设为起始结点的 h 值，开始进行深度优先搜索，忽略所有 f 值大于 Dmax 的结点，减少了很多搜索量。如果没有解，再加大 Dmax 的值，直到找到一个解。容易证明这个解一定是最优的。由于改成了深度优先的方式，与 A*比较起来，IDA*更加实用：

1. 不需要判重，不需要排序，只用栈就可以了。操作简单。
2. 空间需求大大减少，与搜索树大小成对数关系。

其他的启发式搜索

这些方法包括深度优先+最优剪枝式的 A*，双向 A*,但是由于很不成熟或者用处并不大，这里就不介绍了。A*算法有一个加权的形式，由于在搬运工问题中效果不明显，这里从略。

搬运工问题的启示

重庆外语学校 刘汝佳

二 搬运工问题及其特点

在对状态空间搜索算法有一定了解之后，我们来看看我们的搬运工问题。究竟用什么方法比较好呢？让我们先来看看该问题的特点。

1. 搬运工问题

我们在前面已经介绍过搬运工问题，这里我只是想提一些和解题有关的注意事项。首先，我们考虑的搬运工问题的地图规模最大是 20*20，这已经可以满足大部分关卡了。为了以后讨论方便，我们把地图加以编号。从左往右各列称为 A，B，C…，而从上往下各行叫 a,b,c…。而由于不推箱子时的走路并不重要，我们
在记录解的时候忽略了人的位置和移动，只记录箱子的移动。人的动作很容易根据箱子的动作推出来。下面是包含解答的标准关卡第一关。

He-Ge, Hd-Hc-Hd, Fe-Ff, Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh-Rh-Rg,
Ff-Fg-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh-Qi-Ri,
Fc-Fd-Fe-Ff-Fg-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh-Qg,
Ge-Fe-Ff-Fg-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh-Rh,
Hd-He-Ge-Fe-Ff-Fg-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Pi-Qi,
Ch-Dh-Eh-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh

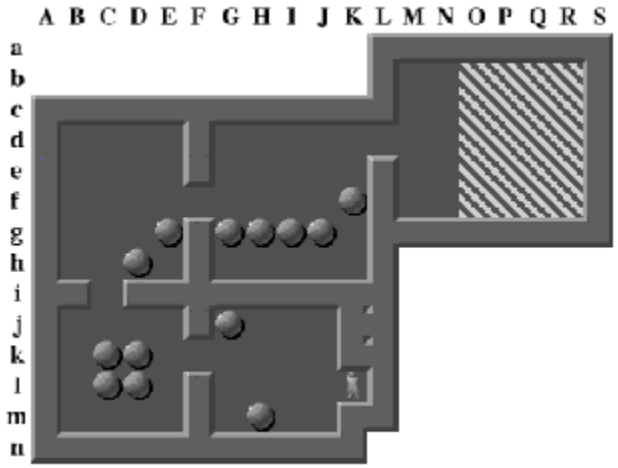
呵呵，怎么样，第一关都要那么多步啊...以后的各关，可是越来越难。

2. 搬运工问题的特点

我在前言里吹了这么半天，我想你即使以前没有玩，现在也已经玩过了吧：）。
有什么感觉呢？是不是变化太多了，不好把握？不仅人不好把握，连编程序也变得困难了很多。我们不妨拿它与经典的 8 数码问题作一个比较。

1.死锁！

初学者很快就会学到什么是死锁－一旦他（她）把一个箱子推到角上。显然这样的布局再继续玩下去是没戏了，不管以后怎么推都不可能把这个箱子推离那个角。不少玩家都总结了不少死锁的经验，但是要比较系统的解决这个问题并不是一件容易的事。我们将用整整一章（其实也不长啦）的篇幅来分析这个问题



典型的死锁。想一想，为什么：）我们再看一下 8 数码问题。它没有死锁，因为每一步都是可逆的。在这一点上，搬运工问题要令人头疼得多了。
容易看出，这样的状态空间不是无向图，而是有向图。

2.状态空间。

8 数码问题每次最多有 4 中移动方法，最多的步数也只有几十步。而搬运工问题呢？困难一点的关卡可以是一步有 100 多种选择，整个解答包括 600 多次推箱子动作。分支因子和解答树深度都这么大，状态空间自然就非同小可了。

3.下界估计

在启发式搜索中，我们需要计算 h 值，也就是需要对下界进行估计。8 数码问题有很多不错的下界函数（如“离家”距离和），但是搬运工问题又怎么样呢？我们不能直接计算“离家”距离，因为谁的家是哪儿都不清楚。很自然，我们可以做一个二分图的最佳匹配，但是这个下界怎么样呢？

a.准确性

对于 A*及其变种来说，下界与实际代价越接近，一般来说算法效率就越高。我们这个最佳匹配只是“理想情况”，但是事实上，在很多情况下箱子相互制约，不得已离开目标路线来为其他箱子腾位置的事情是非常普遍的。例如我们的标准关卡第 50 关，有的箱子需要从目标格子穿过并离开它来为其它箱子让路。我们的下界函数返回值是 100，但是目前的最好结果是 370。多么大的差别！

b.效率

由于下界函数是一个调用非常频繁的函数，其效率不容忽视。最佳匹配的时间渐进复杂度大约是 O(N^3),比 8 数码的下界函数不知大了多少...我们将会在后面给出一些改进方法，但是其本质不会改变。

3. 如何解决搬运工问题

已经有人证明了搬运工问题是 NP-Hard，看来我们还是考虑搜索吧。回想一下上一节提到过的状态空间搜索，用哪一种比较好呢？

既然是智力游戏，可用的启发式信息是非常丰富了，我们不仅是要用，而且要用得尽量充分,所以应该用启发式搜索。而前面已经提到了，搬运工问题的状态空间是非常大的，A*是没有办法了，因此我们选择了 IDA*算法：实现简单，空间需求也少。

既然搬运工问题这么难，为什么有那么多人都解决了相当数量的关卡呢（标准的 90N 年以前就被人们模透了）。因为人聪明嘛。他们会预测，会安排，会学习，有直觉的帮助，还有一定的冒险精神。他们（也包括我啦，呵呵）常用的是一些“高层次”的解题策略，既有效，又灵活。(Srbga:想学吗? Readers:当然想!!)可惜这些策略不是那么简单易学，也不是很有规律的。在后面的章节中，我将尽力模仿人的思维方式给我们的程序加入尽量多的智能。

搬运工问题的启示

重庆外语学校 刘汝佳

三 用 IDA*算法解搬运工问题 实现与改进

在上一节中，我们知道了 IDA*算法是我们解决搬运工问题的核心算法。在这一节里，我们将用 IDA*算法来做一个解决搬运工问题的程序 – 虽然是我们的最初版本(我们称做 S4-Baby)，但是不要小看它哦！

1 IDA*算法框架

由前所述，IDA*算法是基于重复式深度优先的 A*算法，忽略所有 f 值大于深度限制的结点。那么，我们不难写出 IDA*算法框架的伪代码

伪代码 1 - IDA*算法框架

```
procedure IDA_STAR(StartState)
begin
  PathLimit := H( StartState ) – 1;
  Success := False;
    repeat
       inc(PathLimit);
       StartState.g:= 0;
       Push(OpenStack,StartState);
       repeat
         CurrentState:=Pop(OpenStack);
         If Solution(CurrentState) then
           Success = True
         Else if PathLimit >= CurrentState.g + H(CurrentState) then
           Foreach Child(CurrentState) do
             Push(OpenStack, Child(CurrentState));
       until Success or empty(OpenStack);
    until Success or ResourceLimitsReached;
end;
```

这只是一个很粗略的框架，什么事情都不能做。不过我想大家可能比较急于试验一下 IDA*的威力，因此我们不妨就做一个最最基本的程序。

2. 第一个程序

要从框架做一个程序需要填充一些东西。在这里我们就展开一些讨论。

输入输出文件格式

输入文件是一个文本文件，它由 N 行构成，每行是一些字符。各种字符的含义是：

SPACE	空地
.	目标格子
$	箱子
*	目标格子中的箱子
@	搬运工
+	目标格子中的搬运工
#	墙

表 2 输入文件格式

这种格式和 Xsokoban, SokoMind 和 Rolling Stone 的格式是一致的，因此会比较方便一些。

输出文件第一行是推箱子的次数 M，以下 M 行，每行的格式是：x y direction,代表把第 x 行第 y 列的箱子往 direction 的方向推一步。Direction 可以是 left,right,up,down 之中的一个,1<=x,y<=20

数据结构

由于是最初的版本，我们不必考虑这么多：只需要可行，编程方便就可以了，暂时不管它的效率和其他东西。优化是以后的事。

我们定义新的数据类型 BitString,MazeType,MoveType,StateType 和 IDAType。请大家看附录中的程序，不难猜出它们的含义和用途。唯一需要说明的 BitString 类型。记录状态时，我们把地图看成一个大数，一个格子是一个 bit。那么所有箱子构成一个 BitString,检查某一个是否有箱子（或者目标，墙）时只需要检测对应位置上的 bit 是否为 1。这样虽然会浪费一些空间，但是判断会比较快，操作也比较简单。

我们把 x,y 坐标合并成一个"position"变量。其中 Position=(x-1)*width+(y-1)。我们用常量数组 DeltaPos:array[0..3]表示上，下，左，右的 Position 增量。

算法

为了简单起见，我们连最佳匹配也不做了，用所有箱子离最近目标的距离和作为下界函数。不过，这里的"距离"是指推的次数，计算的时候（MinPush 函数），只要忽略其它所有箱子，然后用一次 BFS 就可以了。

效果

嘿嘿，这个效果嘛，不说你也知道的，就是标准关一个也过不了啦。不过为了说明我的程序是正确的，你可以试验一下幼儿关卡(共 61 关)嘛！

什么！第一关都就没有动静了...55555，生成了 18 万个结点???不过很多关都很快就过了的。我们用 1,000,000 个结点为上限(在我的 Celeron 300A 上要运行十多分钟),得到以下的测试结果：

No.	步数	结点数	No.	步数	结点数	No.	步数	结点数
1	15	186476	21	8	102	41	11	145
2	6	24	22	7	110	42	10	118
3	5	14	23	10	192	43	12	223
4	6	24	24	10	432	44	8	63
5	9	31	25	4	23	45	12	138
6	5	8	26	11	846	46	14	178
7	6	35	27	3	18	47	8	296
8	11	39	28	9	38	48	8	156
9	4	12	29	10	142	49	5	60
10	5	14	30	8	641	50	11	14451
11	5	13	31	7	192	51	N/A	>1M
12	4	19	32	3	12	52	N/A	>1M
13	4	14	33	11	51	53	8	470
14	6	20	34	11	332	54	16	24270
15	6	57	35	16	11118	55	N/A	>1M
16	12	3947	36	10	242	56	14	3318
17	6	63	37	9	1171	57	N/A	>1M
18	11	5108	38	11	556	58	N/A	>1M
19	10	467	39	10	72	59	11	328
20	10	1681	40	9	203	60	N/A	>1M
						61	N/A	>1M

没有解决的几关是：51,52,55,57,,58,60,61

比较困难的几关是 1,16,18,20,26,30,35,37,38,50,53,54,56

下面，我们来看看“困难关卡”的下界估计的情况，看看“偷懒”付出的代价。

关卡	最优步数	初始深度	结点总数	顶层结点数
1	15	11	186476	7416
16	12	7	3947	844
18	11	10	5108	49
20	10	6	1681	42
26	11	5	846	394
30	8	6	641	200
35	16	3	11118	3464
37	9	4	1171	493
38	11	5	556	250

50	11	6	14451	51
53	8	5	470	48
54	16	9	24270	2562
56	14	4	3318	460

由此可见，下界估计对于搜索树的大小是很有关系的。看看第 18,20,35,50,54,56 关吧。顶层结点多么少！如果一开始就从这一层搜索不就...看来我们真的需要用最佳匹配算法了。

3.试验最佳匹配算法的威力

好，下面我们来使用最佳匹配算法。最佳匹配算法可以用网络流来实现，但是这里我们采用修改顶标算法，我是抄的书上的程序（偷个懒嘛，呵呵）。现在程序改叫 Baby2 了^_^

下面是刚才的"难题"的测试情况。

关卡	实际步数	初始深度	Baby-1 结点总数	Baby-2 结点总数
1	15	15	186476	60
16	12	10	3947	304
18	11	11	5108	46
20	10	8	1681	76
26	11	5	846	552
30	8	8	641	153
35	16	4	11118	6504
37	9	5	1171	438
38	11	5	556	546
50	11	7	14451	98
53	8	8	470	37
54	16	12	24270	273
56	14	4	3318	2225

哇！有的比刚才的顶层结点还要少！当然了，下界估计好了，当前层的深度剪枝也更准确了嘛。

另外，现在我们来看看文曲星的前 12 关，第 1,2,4,6,8,9,11 关已经可以在 50000 个结点之内出解。

关卡	实际步数	初始深度	结点总数	顶层结点数
1	31	31	75	75
2	11	11	142	142
4	26	18	33923	159
6	16	16	47	47
8	27	21	239	213
9	12	6	4806	2778
11	14	14	73	73

那么下一步干什么呢？打印出每个状态来分析，我们不难发现大量重复结点。所以下一个问题自然就是：怎么避免重复呢？

4.试验 HASH 表的威力

判重嘛，当然就需要 HASH 表了。不过一个很棘手的问题是：如何表示状态？在结点扩展中，我们用比特流的方式定义了箱子的状态，但是在这里我们需要的是合适的数组的下标。这种表示法不爽吧。所以在构造 HASH 表的时候我们就用箱子的坐标来表示状态，也就是 N 元组(p[1],p[2],p[3]..p[n])。至于散列函数嘛，我们根据 HASH 表的项数来考虑。这里，如果箱子最多 100 个，我们就用 10000 项试试看。一种方案是把所有的坐标加起来，但是这样做冲突很多！因为一个箱子 A 右移一格，另一个箱子 B 左移一格，散列函数值不变。考虑到必须使冲突变少，函数又不宜太复杂，我们采用坐标的加权和来作为散列函数值，也就是 Sum{k*Position[k]},当然，最后要对 10000 取余数，其实这个函数也不好，不过我比较懒了，以后再改进吧。至于冲突处理吗，为了简单起见，我们用开链法 设立链表来保存所有元素。值得注意的是，箱子坐标相同而人的坐标不能互通的状态是不同的，应该一起保存。下面是刚才那些关的测试结果：

关卡	实际步数	初始深度	Baby2 结点数	Baby3 结点数
Kid 1	15	15	60	52
Kid 16	12	10	304	189
Kid 18	11	11	46	41
Kid 20	10	8	76	67
Kid 26	11	5	552	192
Kid 30	8	8	153	145
Kid 35	16	4	6504	704
Kid 37	9	5	438	136
Kid 38	11	5	546	152
Kid 50	11	7	98	96
Kid 53	8	8	37	24
Kid 54	16	12	273	258
Kid 56	14	4	2225	1518
Wqx 1	31	31	75	75
Wqx 2	11	11	142	107
Wqx 4	26	18	33923	33916
Wqx 6	16	16	47	46
Wqx 8	27	21	239	226
Wqx 9	12	6	4806	968
Wqx 11	14	14	73	67

新完成的关卡有：

关卡	实际步数	初始深度	结点总数	顶层结点数
Kid 51	13	13	629	629
Kid 52	18	18	39841	39841
Ki d55	14	4	4886	1919
Kid 60	15	9	6916	916

需要注意的是，在保护模式下运行 kid57 的时候出现的 heap overflow，说明完全保存结点没有必要(费空间，费时间)，也不大可能。那么，我们应该怎样做呢？我们知道，评价一个 HASH 表的优劣，一般是从两个方面：查表成功的频率和查表成功以后节省的工作。因此，我们可以设置两个 Hash 表，一个保存最近的结点（查到的可能性比较大）和深度大的结点（一旦找到，节省很多工作）。这样做不会增加多少结点数，但是是程序效率有所提高，求解能力（空间承受能力）也有较大改善。但是为了方便，我们的程序暂时只使用第一个表。

5.结点扩展顺序的优化

在这一节中，我们的最后一个改进是优化结点扩展的顺序，不是想修剪搜索树，而是希望早一点得到解。具体的改进方法是这样的：

1.优先推刚刚推过的箱子
2.然后试所有的能够减少下界的方案，减少得越多越先试。如果减少得一样多，就先推离目标最近的。
3.最后试其他的，也象 2 一样按顺序考虑。

可以预料，这样处理以后，“比较容易”先找到解，但是因为下界估计不准所花费的代价是无法减小的（也就是说只能减少顶层结点数）。不过作为 IDA* 的标准改进方法之一，我们有必要把它加入我们的程序中试试。
(需要注意的是，我们使用的是栈，应该把比较差的方案先压栈)
实际测试结果，1 的效果比较好，2 和 3 的效果不佳，甚至产生了更多的结点。可能主要是我们的下界估计不准确，而 2 和 3 用到了下界函数的缘故。这一个版本 Baby-4 中，我们屏蔽了第 2，3 项措施。
好了，写了四个 Baby 版程序，想不想比较一下呢？不过我只对几个困难一点的数据感兴趣。

关卡	实际步数	Baby-1	Baby-2	Baby-3	Baby-4
Kid 1	11	186476	60	52	38
Kid 16	7	3947	304	189	149
Kid 18	10	5108	46	41	31
Kid 35	16	11118	6504	704	462
Kid 50	11	14451	98	96	152
Kid 51	13	Too many	Too many	629	54
Kid 52	18	Too many	Too many	39841	97
Kid 54	16	24270	273	258	140
Kid 55	14	Too many	Too many	4886	3390
Kid 56	14	3318	2225	1518	1069
Kid 60	15	Too many	Too many	6916	5022
Wqx 4	26	97855	33923	33916	24251
Wqx 9	12	116927	4806	968	350

从上表可以看出，我们的优化总的来说是有效的，而且直观的看，那些改进不明显的很多是因为下界估计比较差，这一点我们以后会继续讨论。不管怎样这 61 关“幼儿关”过了 58 关倒是挺不错的，至少可以说明我们程序的 Baby 版已经具有普通儿童的“智力”了^_^。不过这只是个开头，好戏还在后头！

6.Baby-4 源程序

程序 S4BABY4.PAS 在附件中，这里只是加了少量的注释。大家可以试试它的效果，但是没有必要看得太仔细，因为在以后的章节中，我会改动很多东西，甚至连 IDA* 主程序框架都会变得不一样。

常量定义：

```
const
  {Version}
  VerStr='S4 - SRbGa Super Sokoban Solver (Baby Version 4)';
  Author='Written by Liu Rujia(SrbGa), 2001.2, Chongqing, China';

  {Files}
  InFile='soko.in';
  OutFile='soko.out';

  {Charactors}
  Char_Soko='@';
  Char_SokoInTarget='+';
  Char_Box='$';
  Char_BoxInTarget='*';
  Char_Target='.';
  Char_Wall='#';
  Char_Empty=' ';

  {Dimentions}
  Maxx=21;
  Maxy=21;
  MaxBox=50;

  {Directions}
  Up=0;
  Down=1;
  Left=2;
  Right=3;
  DirectionWords:array[0..3] of string=('UP','DOWN','LEFT','RIGHT');

  {Movement}
  MaxPosition:integer=Maxx*Maxy;
  Opposite:array[0..3] of integer=(1,0,3,2);
  DeltaPos:array[0..3] of integer=(-Maxy,Maxy,-1,1);
```

我们把 x,y 坐标合成一个值 position，其中 position=(x-1)*maxy+(y-1)。这里用类型常量是因为以后会根据地图的尺寸改变 MaxPosition 的值。Opposite 就是相反方向例如

Opposite[UP]:=DOWN;DeltaPos 也是会重新设定的。我们在进行移动的时候只需要用：NewPos:=OldPos+DeltaPos[Direction]就可以了，很方便。

```
 {IDA Related}
 MaxNode=1000000;
 MaxDepth=100;
 MaxStack=150;
 DispNode=1000;
```

每生成多少个结点报告一次。

```
 {HashTable}
 MaxHashEntry=10000;
 HashMask=10000;
 MaxSubEntry=100;

 {BitString}
 BitMask:array[0..7] of byte=(1,2,4,8,16,32,64,128);

 Infinite=Maxint;
```

类型定义：

```
type
 PositionType=integer;

 BitString=array[0..Maxx*Maxy div 8-1] of byte;
```

整个地图就是一个 BitString。第 position 位为 1 当且仅当 position 位置有东西（如箱子，目标，墙）。

```
 MapType=array[1..Maxx] of string[Maxy];
 BiGraph=array[1..MaxBox,1..MaxBox] of integer;

 MazeType=
 record
   X,Y:integer;
   Map:MapType;
   GoalPosition:array[1..MaxBox] of integer;
   BoxCount:integer;
   Goals:BitString;
   Walls:BitString;
 end;
```

尺寸，原始数据（用来显示状态的），目标的 BitString，箱子总数，目标位置（BitString 和位置数组都用是为了加快速度）和 Walls 的 BitString。

```
 MoveType=
 record
```

```
    Position:integer;
    Direction:0..3;
  end;
```
Direction 是箱子被推向的方向。

```
  StateType=
  record
    Boxes:BitString;
    ManPosition:PositionType;
    MoveCount:integer;
    Move:array[1..MaxDepth] of MoveType;
    g,h:integer;
  end;

  IDAType=
  record
    TopLevelNodeCount:longint;
    NodeCount:longint;
    StartState:StateType;
    PathLimit:integer;
    Top:integer;
    Stack:array[1..MaxStack] of StateType;
  end;
```
Top 是栈顶指针。

```
  PHashTableEntry=^HashTableEntry;
  HashTableEntry=
  record
    Next:PHashTableEntry;
    State:StateType;
  end;

  PHashTableType=^HashTableType;
  HashTableType=
  record
    FirstEntry:array[0..MaxHashEntry] of PHashTableEntry;
    Count:array[0..MaxHashEntry] of byte;
  end;
```
这些是 Hash 表相关类型。我们采用的是拉链法，这样可以利用指针申请到堆空间，结合保护模式使用，效果更好。

```
var
  HashTable:PHashTableType;
  SokoMaze:MazeType;
```

```
  IDA:IDAType;

procedure SetBit(var BS:BitString; p:integer);
begin
  BS[p div 8]:=BS[p div 8] or BitMask[p mod 8];
end;

procedure ClearBit(var BS:BitString; p:integer);
begin
  BS[p div 8]:=BS[p div 8] xor BitMask[p mod 8];
end;

function GetBit(var BS:BitString; p:integer):byte;
begin
  if BS[p div 8] and BitMask[p mod 8]>0 then GetBit:=1 else GetBit:=0;
end;
```

这些是位操作，设置，清除和得到一个 BitString 的某一项。

```
procedure Init;
var
  Lines:MapType;

  procedure ReadInputFile;
  var
    f:text;
    s:string;
  begin
    SokoMaze.X:=0;
    SokoMaze.Y:=0;
    SokoMaze.BoxCount:=0;
    assign(f,infile);
    reset(f);
    while not eof(f) do
    begin
      readln(f,s);
      if length(s)>SokoMaze.Y then
        SokoMaze.Y:=length(s);
      inc(SokoMaze.X);
      Lines[SokoMaze.X]:=s;
    end;
    close(f);
  end;

  procedure AdjustData;
```

```
 var
  i,j:integer;
 begin
  for i:=1 to SokoMaze.X do
   while length(Lines[i])<SokoMaze.Y do
    Lines[i]:=Lines[i]+' ';

  SokoMaze.Map:=Lines;
  for i:=1 to SokoMaze.X do
   for j:=1 to SokoMaze.Y do
    if SokoMaze.Map[i,j] in [Char_BoxInTarget,Char_SokoInTarget,Char_Target] then
     SokoMaze.Map[i,j]:=Char_Target
    else if SokoMaze.Map[i,j]<>Char_Wall then
     SokoMaze.Map[i,j]:=Char_Empty;
```
调整 Map 数组，把箱子和搬运工去掉。
```
  for i:=1 to SokoMaze.X do
   for j:=1 to SokoMaze.Y do
    if Lines[i,j] in [Char_Target,Char_BoxInTarget,Char_SokoInTarget] then
    begin
     inc(SokoMaze.BoxCount);
     SokoMaze.GoalPosition[SokoMaze.BoxCount]:=(i-1)*SokoMaze.Y+j-1;
    end;
```
统计 Goal 的个数和 GoalPosition。
```
  DeltaPos[Up]:=-SokoMaze.Y;
  DeltaPos[Down]:=SokoMaze.Y;
  MaxPosition:=SokoMaze.X*SokoMaze.Y;
```
根据地图尺寸调整 DeltaPos 和 MaxPosition
```
 end;

 procedure ConstructMaze;
 var
  i,j:integer;
 begin
  fillchar(SokoMaze.Goals,sizeof(SokoMaze.Goals),0);
  fillchar(SokoMaze.Walls,sizeof(SokoMaze.Walls),0);

  for i:=1 to SokoMaze.X do
   for j:=1 to SokoMaze.Y do
    case Lines[i,j] of
     Char_SokoInTarget, Char_BoxInTarget, Char_Target:
      SetBit(SokoMaze.Goals,(i-1)*SokoMaze.Y+j-1);
     Char_Wall:
      SetBit(SokoMaze.Walls,(i-1)*SokoMaze.Y+j-1);
```

```
      end;
  end;

  procedure InitIDA;
  var
   i,j:integer;
   StartState:StateType;
  begin
   IDA.NodeCount:=0;
   IDA.TopLevelNodeCount:=0;
   fillchar(StartState,sizeof(StartState),0);

   for i:=1 to SokoMaze.X do
    for j:=1 to SokoMaze.Y do
     case Lines[i,j] of
      Char_Soko, Char_SokoInTarget:
       StartState.ManPosition:=(i-1)*SokoMaze.Y+j-1;
      Char_Box, Char_BoxInTarget:
       SetBit(StartState.Boxes,(i-1)*SokoMaze.Y+j-1);
     end;

   StartState.g:=0;
   IDA.StartState:=StartState;
   new(HashTable);
   for i:=1 to MaxHashEntry do
   begin
    HashTable^.FirstEntry[i]:=nil;
    HashTable^.Count[i]:=0;
   end;
  end;

begin
  ReadInputFile;
  AdjustData;
  ConstructMaze;
  InitIDA;
end;

procedure PrintState(State:StateType);
var
  i,x,y:integer;
  Map:MapType;
begin
  Map:=SokoMaze.Map;
```

```
  x:=State.ManPosition div SokoMaze.Y+1;
  y:=State.ManPosition mod SokoMaze.Y+1;
  if Map[x,y]=Char_Target then
    Map[x,y]:=Char_SokoInTarget
  else
    Map[x,y]:=Char_Soko;

  for i:=1 to MaxPosition do
    if GetBit(State.Boxes,i)>0 then
    begin
      x:=i div SokoMaze.Y+1;
      y:=i mod SokoMaze.Y+1;
      if Map[x,y]=Char_Target then
        Map[x,y]:=Char_BoxInTarget
      else
        Map[x,y]:=Char_Box;
    end;
  for i:=1 to SokoMaze.X do
    Writeln(Map[i]);
end;

function Solution(State:StateType):boolean;
var
  i:integer;
begin
  Solution:=false;
  for i:=1 to MaxPosition do
    if (GetBit(State.Boxes,i)>0) and (GetBit(SokoMaze.Goals,i)=0) then
      exit;
  Solution:=true;
end;

function CanReach(State:StateType; Position:integer):boolean;
用 BFS 判断在状态 State 中，搬运工是否可以到达 Position
var
  Direction:integer;
  Pos,NewPos:integer;
  Get,Put:integer;
  Queue:array[0..Maxx*Maxy] of integer;
  Reached:Array[0..Maxx*Maxy] of boolean;
begin
  fillchar(Reached,sizeof(Reached),0);

  Pos:=State.ManPosition;
```

```
  Get:=0; Put:=1;
  Queue[0]:=Pos;
  Reached[Pos]:=true;

  CanReach:=true;
  while Get<>Put do
  begin
    Pos:=Queue[Get];
    inc(Get);
    if Pos=Position then
      exit;
    for Direction:=0 to 3 do
    begin
      NewPos:=Pos+DeltaPos[Direction];
      if Reached[NewPos] then continue;
      if GetBit(State.Boxes,NewPos)>0 then continue;
      if GetBit(SokoMaze.Walls,NewPos)>0 then continue;
      Reached[NewPos]:=true;
      Queue[Put]:=NewPos;
      inc(Put);
    end;
  end;
  CanReach:=false;
end;

function MinPush(BoxPosition,GoalPosition:integer):integer;
在没有其他箱子的情况下，从 BoxPosition 推到 GoalPosition 至少要多少步。
var
  i:integer;
  Direction:integer;
  Pos,NewPos,ManPos:integer;
  Get,Put:integer;
  Queue:array[0..Maxx*Maxy] of integer;
  Distance:Array[0..Maxx*Maxy] of integer;
begin
  for i:=0 to Maxx*Maxy do
    Distance[i]:=Infinite;

  Pos:=BoxPosition;
  Get:=0; Put:=1;
  Queue[0]:=Pos;
  Distance[Pos]:=0;

  while Get<>Put do
```

```
  begin
    Pos:=Queue[Get];
    inc(Get);
    if Pos=GoalPosition then
    begin
      MinPush:=Distance[Pos];
      exit;
    end;
    for Direction:=0 to 3 do
    begin
      NewPos:=Pos+DeltaPos[Direction];
      ManPos:=Pos+DeltaPos[Opposite[Direction]];
      人应该站在后面
          if Distance[NewPos]<Infinite then continue;
      if GetBit(SokoMaze.Walls,NewPos)>0 then continue;
      推不动
      if GetBit(SokoMaze.Walls,ManPos)>0 then continue;
      人没有站的地方
      Distance[NewPos]:=Distance[Pos]+1;
      Queue[Put]:=NewPos;
      inc(Put);
    end;
  end;
  MinPush:=Infinite;
end;

procedure DoMove(State:StateType; Position,Direction:integer; var NewState:StateType);
var
  NewPos:integer;
begin
  NewState:=State;
  NewPos:=Position+DeltaPos[Direction];
  NewState.ManPosition:=Position;
  SetBit(NewState.Boxes,NewPos);
  ClearBit(NewState.Boxes,Position);
end;

function MinMatch(BoxCount:integer;Gr:BiGraph):integer;
这个是标准算法，抄的书上的程序，不用看了。
var
  VeryBig:integer;
  TempGr:BiGraph;
  L:array[1..MaxBox*2] of integer;
  SetX,SetY,MatchedX,MatchedY:Set of 1..MaxBox;
```

```
procedure MaxMatch(n,m:integer);
function Path(x:integer):boolean;
var
  i,j:integer;
begin
  Path:=false;
  for i:=1 to m do
    if not (i in SetY)and(Gr[x,i]<>0) then
    begin
      SetY:=SetY+[i];
      if not (i in MatchedY) then
      begin
        Gr[x,i]:=-Gr[x,i];
        MatchedY:=MatchedY+[i];
        Path:=true;
        exit;
      end;
      j:=1;
      while (j<=m)and not (j in SetX) and (Gr[j,i]>=0) do inc(j);
      if j<=m then
      begin
        SetX:=SetX+[j];
        if Path(j) then
        begin
          Gr[x,i]:=-Gr[x,i];
          Gr[j,i]:=-Gr[j,i];
          Path:=true;
          exit;
        end;
      end;
    end;
end;

var
  u,i,j,al:integer;

begin
  Fillchar(L,sizeof(L),0);
  TempGr:=Gr;
  for i:=1 to n do
    for j:=1 to m do
      if L[i]<Gr[i,j] then
        L[i]:=Gr[i,j];
```

```
  u:=1; MatchedX:=[]; MatchedY:=[];
  for i:=1 to n do
    for j:=1 to m do
      if L[i]+L[n+j]=TempGr[i,j] then
        Gr[i,j]:=1
      else
        Gr[i,j]:=0;
  while u<=n do
  begin
    SetX:=[u]; SetY:=[];
    if not (u in MatchedX) then
    begin
      if not Path(u) then
      begin
        al:=Infinite;
        for i:=1 to n do
          for j:=1 to m do
            if (i in SetX) and not (j in SetY) and (L[i]+L[n+j]-TempGr[i,j]<al) then
              al:=L[i]+L[n+j]-TempGr[i,j];
        for i:=1 to n do if i in SetX then L[i]:=L[i]-al;
        for i:=1 to m do if i in SetY then l[n+i]:=l[n+i]+al;
        for i:=1 to n do
          for j:=1 to m do
            if l[i]+l[n+j]=TempGr[i,j] then
              Gr[i,j]:=1
            else
              Gr[i,j]:=0;
        MatchedX:=[]; MatchedY:=[];
        for i:=1 to n+m do
          if l[i]<-1000 then
            exit;
      end
      else
        MatchedX:=MatchedX+[u];
      u:=0;
    end;
    inc(u);
  end;
end;

var
  i,j:integer;
  Tot:integer;
begin
```

```
  VeryBig:=0;
  for i:=1 to BoxCount do
    for j:=1 to BoxCount do
      if (Gr[i,j]<Infinite)and(Gr[i,j]>VeryBig) then
        VeryBig:=Gr[i,j];
  inc(VeryBig);

  for i:=1 to BoxCount do
    for j:=1 to BoxCount do
      if Gr[i,j]<Infinite then
        Gr[i,j]:=VeryBig-Gr[i,j]
      else
        Gr[i,j]:=0;
```
这些语句是进行补集转化。
```
  MaxMatch(BoxCount,BoxCount);
  Tot:=0;
  for i:=1 to BoxCount do
  begin
    for j:=1 to BoxCount do
      if Gr[i,j]<0 then
      begin
        Tot:=Tot+VeryBig-TempGr[i,j];
        break;
      end;
    if Gr[i,j]>=0 then
    begin
      MinMatch:=Infinite;
      exit;
    end;
  end;
  MinMatch:=Tot;
end;

function CalcHeuristicFunction(State:StateType):integer;
```
计算启发函数值
```
var
  H,Min:integer;
  i,j,p,Count,BoxCount,Cost:integer;
  BoxPos:array[1..MaxBox] of integer;
  Distance:BiGraph;
begin
  p:=0;
  for i:=1 to MaxPosition do
    if GetBit(State.Boxes,i)>0 then
```

```
    begin
      inc(p);
      BoxPos[p]:=i;
    end;
  for i:=1 to p do
    for j:=1 to p do
      Distance[i,j]:=MinPush(BoxPos[i],SokoMaze.GoalPosition[j]);
  BoxCount:=SokoMaze.BoxCount;

  H:=0;
  for i:=1 to BoxCount do
  begin
    Count:=0;
    for j:=1 to BoxCount do
      if Distance[i,j]<Infinite then
        inc(Count);
    if Count=0 then
    有一个箱子推不到任何目的地
    begin
      CalcHeuristicFunction:=Infinite;
      exit;
    end;
  end;

  H:=MinMatch(BoxCount, Distance);
  CalcHeuristicFunction:=H;
end;

function HashFunction(State:StateType):integer;
var
  i,h,p:integer;
begin
  h:=0;
  p:=0;
  for i:=1 to MaxPosition do
    if GetBit(State.Boxes,i)>0 then
    begin
      inc(p);
      h:=(h+p*i) mod HashMask;
      你可以自己换一个
    end;
  HashFunction:=h;
end;
```

```
function SameState(S1,S2:StateType):boolean;
var
  i:integer;
begin
  SameState:=false;
  for i:=1 to MaxPosition do
    if GetBit(S1.Boxes,i)<>GetBit(S2.Boxes,i) then
      exit;
  if not CanReach(S1,S2.ManPosition) then
  注意只要两个状态人的位置是相通的就应该算同一个状态
    exit;
  SameState:=true;
end;

function Prior(State:StateType;M1,M2:MoveType):boolean;
var
  NewPos:integer;
  Inertia1,Inertia2:boolean;
  S1,S2:StateType;
  H1,H2:integer;
begin
  Prior:=false;
  if State.MoveCount>0 then
  begin
    NewPos:=State.Move[State.MoveCount].Position+
          DeltaPos[State.Move[State.MoveCount].Direction];
    if NewPos=M1.Position then Inertia1:=true else Inertia1:=false;
    连续推同一个箱子的动作优先
    if NewPos=M2.Position then Inertia2:=true else Inertia2:=false;
    if Inertia1 and not Inertia2 then begin Prior:=true; exit; end;
    if Inertia2 and not Inertia1 then begin Prior:=false; exit; end;
  end;
end;

procedure IDA_Star;
var
  Sucess:boolean;
  CurrentState:StateType;
  H:integer;
  f:Text;

  procedure IDA_Push(State:StateType);
  begin
    if IDA.Top=MaxStack then
```

```
    Exit;
  inc(IDA.Top);
  IDA.Stack[IDA.Top]:=State;
 end;

 procedure IDA_Pop(var State:StateType);
 begin
  State:=IDA.Stack[IDA.Top];
  dec(IDA.Top);
 end;

 function IDA_Empty:boolean;
 begin
  IDA_Empty:=(IDA.Top=0);
 end;
```

上面的是栈操作

```
 procedure IDA_AddToHashTable(State:StateType);
 var
  h:integer;
  p:PHashTableEntry;
 begin
  h:=HashFunction(State);
  if HashTable^.Count[h]<MaxSubEntry then
  begin
    new(p);
    p^.State:=State;
    p^.Next:=HashTable^.FirstEntry[h];
    HashTable^.FirstEntry[h]:=p;
    inc(HashTable^.Count[h]);
  end
  else begin
    p:=HashTable^.FirstEntry[h];
    while p^.Next^.Next<>nil do
      p:=p^.Next;
    p^.Next^.State:=State;
    p^.Next^.Next:=HashTable^.FirstEntry[h];
    HashTable^.FirstEntry[h]:=p^.Next;
    p^.Next:=nil;
  end;
 end;

 function IDA_InHashTable(State:StateType):boolean;
 var
```

```
  h:integer;
  p:PHashTableEntry;
 begin
  h:=HashFunction(State);
  p:=HashTable^.FirstEntry[h];
  IDA_InHashTable:=true;
  while p<>nil do
  begin
    if SameState(p^.State,State) then
    begin
      if p^.State.g>State.g then
      begin
        p^.State.g:=State.g;
        IDA_InHashTable:=false;
```

如果找到的表项深度要大些，并不代表这一次深度小点的也无解。本来应该动态更新下界的，这里作为没有找到处理，后面的章节会改进这个地方的。

```
      end;
      exit;
    end;
    p:=p^.Next;
  end;
  IDA_InHashTable:=false;
 end;
```

这是 Hash 表的操作。

```
 procedure IDA_AddNode(State:StateType);
 begin
  IDA_Push(State);
  inc(IDA.NodeCount);
  if IDA.NodeCount mod DispNode=0 then
    Writeln('NodeCount=',IDA.NodeCount);
  inc(IDA.TopLevelNodeCount);
  IDA_AddToHashTable(State);
 end;

 procedure IDA_Expand(State:StateType);
 var
  MoveCount:integer;
  MoveList:array[1..Maxx*Maxy*4] of MoveType;
  t:MoveType;
  i,j,Direction:integer;
  NewBoxPos, NewManPos:integer;
  NewState:StateType;
 begin
```

```
  MoveCount:=0;
  for i:=1 to MaxPosition do
    if GetBit(State.Boxes,i)>0 then
      for Direction:=0 to 3 do
      begin
        NewBoxPos:=i+DeltaPos[Direction];
        NewManPos:=i+DeltaPos[Opposite[Direction]];
        if GetBit(State.Boxes,NewBoxPos)>0 then continue;
        if GetBit(SokoMaze.Walls,NewBoxPos)>0 then continue;
        if GetBit(State.Boxes,NewManPos)>0 then continue;
        if GetBit(SokoMaze.Walls,NewManPos)>0 then continue;
        if CanReach(State,NewManPos) then
        begin
          DoMove(State,i,Direction,NewState);
          if CalcHeuristicFunction(NewState)=Infinite then continue;
          if CalcHeuristicFunction(NewState)+State.g>=IDA.PathLimit then continue;
```

IDA*算法的核心：深度限制

```
          if IDA_InHashTable(NewState) then continue;
          inc(MoveCount);
          MoveList[MoveCount].Position:=i;
          MoveList[MoveCount].Direction:=Direction;
        end;
      end;

  for i:=1 to MoveCount do
    for j:=i+1 to MoveCount do
      if Prior(State,MoveList[i],MoveList[j]) then
      调整推法次序
      begin
        t:=MoveList[j];
        MoveList[j]:=MoveList[i];
        MoveList[i]:=t;
      end;

  for i:=1 to MoveCount do
  依次考虑所有移动方案
  begin
    DoMove(State,MoveList[i].Position,MoveList[i].Direction,NewState);
    inc(NewState.MoveCount);
    NewState.Move[NewState.MoveCount].Position:=MoveList[i].Position;
    NewState.Move[NewState.MoveCount].Direction:=MoveList[i].Direction;
    NewState.g:=State.g+1;
    IDA_AddNode(NewState);
  end;
```

```
  end;

  procedure IDA_Answer(State:StateType);
  var
    i:integer;
    x,y:integer;
  begin
    Writeln(f,'Solution Found in ', State.MoveCount,' Pushes');
    for i:=1 to State.Movecount do
    begin
      x:=State.Move[i].Position div SokoMaze.Y+1;
      y:=State.Move[i].Position mod SokoMaze.Y+1;
      Writeln(f, x,' ',y,' ',DirectionWords[State.Move[i].Direction]);
    end;
  end;

begin
  Writeln(VerStr);
  Writeln(Author);

  IDA.PathLimit:=CalcHeuristicFunction(IDA.StartState)-1;
  Sucess:=false;
  repeat
    inc(IDA.PathLimit);
    Writeln('Pathlimit=',IDA.PathLimit);
    IDA.TopLevelNodeCount:=0;
    IDA.Top:=0;
    IDA.StartState.g:=0;
    IDA_Push(IDA.StartState);
    repeat
      IDA_Pop(CurrentState);
      H:=CalcHeuristicFunction(CurrentState);
      if H=Infinite then continue;
      if Solution(CurrentState) then
        Sucess:=true
      else if IDA.PathLimit>=CurrentState.g+H then
        IDA_Expand(CurrentState);
    until Sucess or IDA_Empty or (IDA.NodeCount>MaxNode);
    Writeln('PathLimit ',IDA.PathLimit,' Finished. NodeCount=',IDA.NodeCount);
  until Sucess or (IDA.PathLimit>=MaxDepth) or (IDA.NodeCount>MaxNode);

  Assign(f,outfile);
  ReWrite(f);
  Writeln(f,VerStr);
```

```
  Writeln(f,Author);
  Writeln(f);

  if not Sucess then
    Writeln(f,'Cannot find a solution.')
  else
    IDA_Answer(CurrentState);

  Writeln('Node Count:',IDA.NodeCount);
  Writeln;
  close(f);
end;

begin
  Init;
  IDA_Star;
end.
```

由“汽车问题”浅谈深度搜索的一个方面

———搜索对象与策略的重要性

问 题 描 述

有一个人在某个公共汽车站上，从 12:00 到 12:59 观察公共汽车到达本站的情况，该站被多条公共汽车线路所公用，他依次记下公共汽车到达本站的时刻。

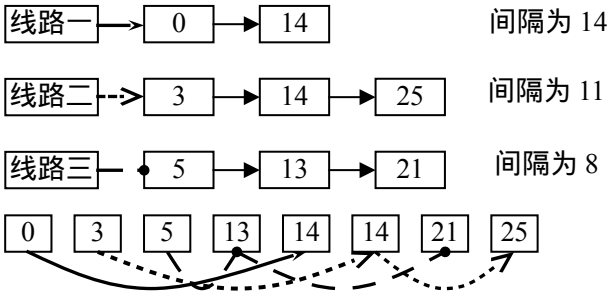
- 在 12:00－12:59 期间，同一条线路上的公共汽车以相同的时间间隔到站。
- 时间单位用“分”表示，从 0 到 59 。
- 每条公共汽车线路至少有两辆车到达本站。
- 公共汽车线路数 K 一定≤17，汽车数目 N 一定小于 300。
- 来自不同线路的公共汽车可能在同一时刻到达本站。
- 不同公共汽车线路的车首次到站时间和到站的时间间隔都有可能相同。

请为公共汽车线路编一个调度表，目标是：公共汽车线路数目最少的情况下，使公共汽车到达本站的时刻满足输入数据的要求。

例如：

汽车编号	1	2	3	4	5	6	7	8
到达时间	0	3	5	13	14	14	21	25

那就可能存在这样一个解，由以下 3 条汽车线路组成：



解　析

经过一系列的分析，我们决定用深度搜索（由于通篇讨论的是深度搜索，以下就统一简称搜索）解这道题目。

对这样一个问题，首先提取出三个关键要素：**时间、车、路线**。

☆车辆的特征是时间，

☆路线的特征是“首发车时间”和“间隔时间”，这等效于“第一辆车”和“第二辆车”。

面对这三个关键要素，下面就要从中确定搜索对象和搜索策略。可以看出，**题目要求的是车和线路的关系，而时间在其中起的是描述作用和条件制约作用**，因此，本题的搜索对象应该是车或线路这两个关键要素。

- **分析搜索对象及策略**

1．对象—→ 车

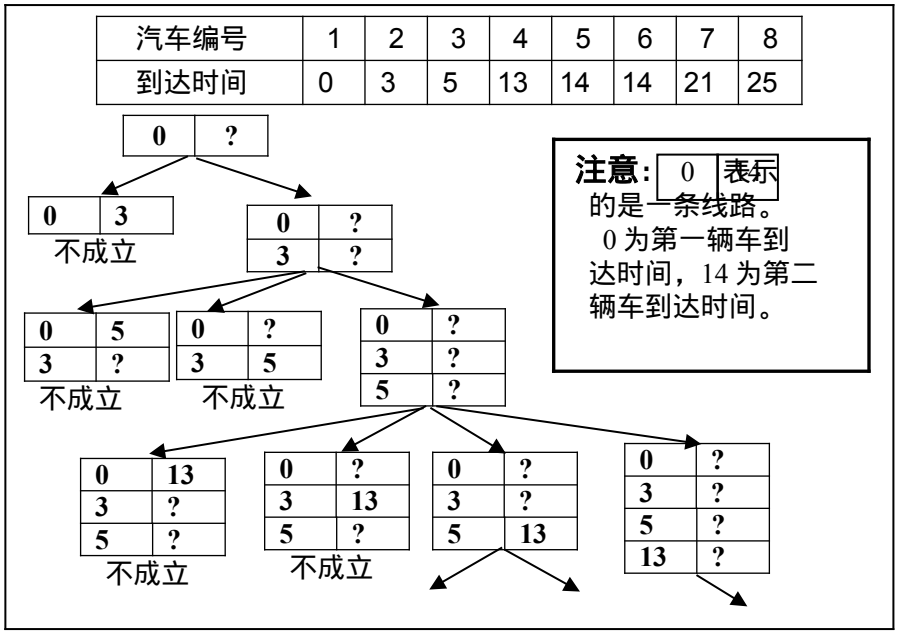
由此对象而产生的搜索策略是：按到站时间顺序，依次枚举每辆车属于哪条路线。

注意路线的特征，若一路线的**第一辆车**和**第二辆车**确定了，那该路线也就确定了。

大致搜索方案为：

按到达时间顺序，依次对于那些没有确定属于哪条线路的车进行枚举，该车属于某新线路的**第一辆车**或属于某已有线路的**第二辆车**，若为后一种选择，则可确定路线上其他所有的车辆。

还是看先前给的那个简单例子来构造搜索树大致如下：



观察该搜索树，发现：随着搜索树层数的递增，每层节点所扩展出的树叉数目逐渐增大。从直观上说：该搜索树从根开始，“分叉”越来越多，“枝叶”越来越茂盛。

这就是搜索对象为车的搜索树的典型特点。

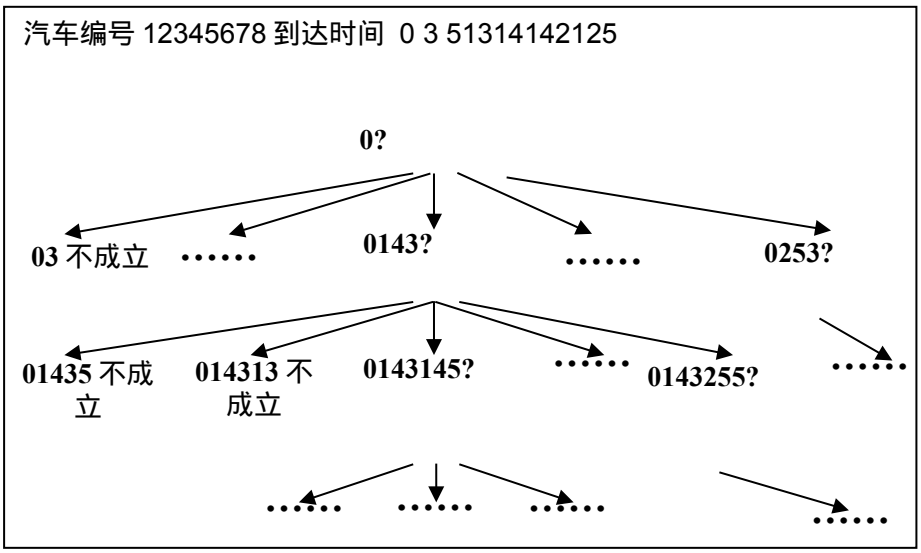
2．对象—→线路

由此对象而产生的搜索策略是：枚举每条路线包含哪些车，确定该路线。

实际上，对每条路线，我们只要枚举其特征：**第一辆车**和**第二辆车**。再根据“有序化”思想，固定所有路线是按照第一辆车的到达时间为关键字排序的。

大致搜索方案为：

搜索每层都要确定一条路线：将未确定归属路线的到达时间最小的车**固定**为新路线的



第一辆车，其后枚举这条路线的第二辆车，从而确定该路线。

同样，根据先前给的那个简单例子我们也构造出了方法二的搜索树（见前页）。

观察这个搜索树，和方法一的搜索树对比，我们发现，两者特性截然相反，该搜索树从根开始，“分叉”越来越少。

两种搜索对象及策略是完全不同的，搜索树特性又截然相反。从宏观上，搜索树上节点多少，两者相差无几。如何抉择呢？这时就要从微观上比较：

● 比较哪个搜索对象和策略更优

既然是比较，就要有比较的标准。这里，确定了两个标准：

谁易于优化剪枝 ； 谁的操作量小

★关于**谁易于优化剪枝**的比较：

本题的主要剪枝有三种，如下逐一分析。

◇ 可行性剪枝

当路线的**特征**确定了，就可以判断该路线是否成立。

方法一，搜索中，每层枚举当前车是哪条路线的第二辆车，都要用到该判断；方法二，每层是确定一条路线，也用到该判断。关键是，根据两者搜索树，方法一一旦剪枝，将剪去的是一大片“茂盛”的树枝，显然，相比之下，方法二剪枝的效果就差了许多。

故在此剪枝应用上，方法二比方法一逊色许多。

◇ 与已知最优解比较剪枝

这就要看谁能很快找到解了。显然，由于方法一可行性剪枝的优点，每次剪枝都能删去**很多**的不可行的节点，找到解的速度就不比方法二慢了。

此剪枝，方法二不比方法一要好。

◇ 排除重复剪枝

注意到题目中**时间**这个关键要素的范围为 0~59，而车辆数目可达 300，说明，在同一时间到达的车辆数目很多！前面那个简单例子中，就出现了两个 14，而到达时间为 14 的两辆车各属于那条路线是等效的，这就有重复。

方法一对于同时到达的且未确定归属的车，若编号小的车为某路线的第一辆车，则编号大的车为也必为一条路线的第一辆车。

方法二，对到达时间相同的且未确定归属的车，固定只选编号最小的车为第二辆车。

两者剪去的都是重复的枝，所以，效果是一样的，故此剪枝上，双方平分秋色。

结论： 由于方法一搜索树的良好特性，使得方法一在剪枝优化方面前景更广阔。

★关于**谁的操作量小**的比较：

操作量是“主递归程序操作量”的简称，由主递归程序的枚举循环和剪枝函数决定的。

◇主递归程序的枚举循环

方法一，每层利用循环来枚举一辆车是属于新路线的第一辆车还是已知路线的第二辆车，而且搜索树上这个循环枚举量是由未确定“第二辆车”的线路数目决定的，最大为 17。

方法二，每层枚举哪辆车是第二辆车。由于用了排除重复剪枝，这个循环量最大为 60。

直观上看两者的最大界限就已知道，方法二不比方法一好。

◇剪枝函数消耗时间

由于本题特殊性，主要剪枝函数基本上差异不大，消耗时间也差不多。

结论： 操作量大小方面，方法二不比方法一好。

最终结论：

通过以上微观理论分析，对两种方法进行了比较，得出最终结论是：方法一比方法二

好！选定了搜索对象和策略，刚才又分析了实现中的主要问题：如何剪枝，编写程序自然就得心应手了。

事实也证明了我们的结论。两种程序运行比较如下，可以发现两者优劣差异巨大。

	car.in1 K=3	car.in2 K=5	car.in3 K=8	car.in4 K=10	car.in5 K=15	car.in6 K=17
方法一用时	0.01s	0.01s	0.05s	0.11s	1.48s	1.76s
方法二用时	0.01s	0.01s	9.07s	>100s	>100s	>100s

总　结

就本题而言，从宏观上看，很难知道两种方法效率的差异，为什么方法一更好呢？关键原因在于：它适合程序上的**剪枝优化**，且操作量小。

那为什么选择这两个方面为标准呢？

我们知道：

深度搜索消耗时间 ≈ 每个节点操作系数 × 节点个数

从上面一个公式，我们很显然地能从微观上看出，要减少消耗时间，

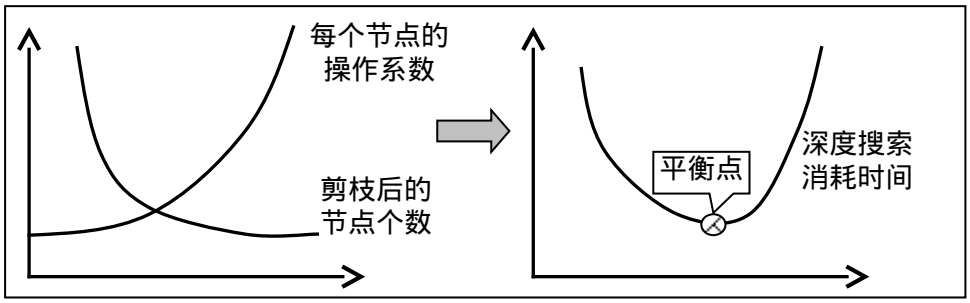
一是减少节点个数——这就是我们所说地**剪枝优化**；

二是减少每个节点的操作系数——即刚才分析的**程序操作量**。

为了提高搜索效率，根据这个公式，我们往往在以上两方面反复进行改进。殊不知，**从宏观前提上，如何才能使我们"可以、充分、有效"剪枝，如何才能使我们"可以、充分、有效"降低程序操作量也都是很重要的！**

于是，搜索对象和策略为剪枝，降低操作系数创造前提条件的好坏就成了我们的标准！

但是，在以这两方面为标准比较的时候，我们要注意到：这两个标准紧密关联，要剪枝多，就要有好的复杂的剪枝函数，但这就增大了每个节点操作系数。如图所示：



两者在目的上是统一的，效果上却是对立的。在以这两者为标准的时候，要把握好"如何协调，找准两者**平衡点**"。

总的来说，对深度搜索题目，一个好的搜索对象和策略是十分重要的。本文通过对"汽车问题"的分析，充分说明了这点，并且根据**深度搜索消耗时间公式**提出了比较搜索对象和策略的标准：优化剪枝与操作系数。

同时，对**深度搜索消耗时间公式**的分析也启发我们，**为了更好的解决问题达到目的，仅仅在微观上进行变动更新是不够的，还要首先为这个目的去创造良好的宏观条件！**

好的搜索对象和策略正是成功解搜索题的宏观条件，这也是本文的主旨。

动态规划算法的优化技巧

福州第三中学　毛子青

[关键词] 动态规划、 时间复杂度、优化、状态

[摘要]

动态规划是信息学竞赛中一种常用的程序设计方法，本文着重讨论了运用动态规划思想解题时时间效率的优化。全文分为四个部分，首先讨论了动态规划时间效率优化的可行性和必要性，接着给出了动态规划时间复杂度的决定因素，然后分别阐述了对各个决定因素的优化方法，最后总结全文。

[正文]

一、引言

动态规划是一种重要的程序设计方法，在信息学竞赛中具有广泛的应用。

使用动态规划方法解题，对于不少问题具有空间耗费大、时间效率高的特点，因此人们在研究动态规划解题时更多的注意空间复杂度的优化，运用各种技巧将空间需求控制在软硬件可以承受的范围之内。但是，也有一部分问题在使用动态规划思想解题时，时间效率并不能满足要求，而且算法仍然存在优化的余地，这时，就需要考虑时间效率的优化。

本文讨论的是在确定使用动态规划思想解题的情况下，对原有的动态规划解法的优化，以求降低算法的时间复杂度，使其能够适用于更大的规模。

二、动态规划时间复杂度的分析

使用动态规划方法解题，对于不少问题之所以具有较高的时间效率，关键在于它减少了“**冗余**”。所谓“冗余”，就是指不必要的计算或重复计算部分，算法的冗余程度是决定算法效率的关键。动态规划在将问题规模不断缩小的同时，记录已经求解过的子问题的解，充分利用求解结果，避免了反复求解同一子问题的现象，从而减少了冗余。

但是，动态规划求解问题时，仍然存在冗余。它主要包括：求解无用的子问题，对结果无意义的引用等等。

下面给出动态规划时间复杂度的决定因素：

时间复杂度=状态总数*每个状态转移的状态数*每次状态转移的时间[1]

下文就将分别讨论对这三个因素的优化。这里需要指出的是：这三者之间不是相互独立的，而是相互联系，矛盾而统一的。有时，实现了某个因素的优化，另外两个因素也随之得到了优化；有时，实现某个因素的优化却要以增大另一因素为代价。因此，这就要求我们在优化时，坚持“全局观”，实现三者的平衡。

三、动态规划时间效率的优化

3.1 减少状态总数

我们知道，动态规划的求解过程实际上就是计算所有状态值的过程，因此状态的规模直接影响到算法的时间效率。所以，减少状态总数是动态规划优化的重要部分，本节将讨论减少状态总数的一些方法。

1、改进状态表示

状态的规模与状态表示的方法密切相关，通过改进状态表示减小状态总数是应用较为普遍的一种方法。

例一、 Raucous Rockers 演唱组（USACO`96）

[问题描述]

现有 n 首由 Raucous Rockers 演唱组录制的珍贵的歌曲，计划从中选择一些歌曲来发行 m 张唱片，每张唱片至多包含 t 分钟的音乐，唱片中的歌曲不能重叠。按下面的标准进行选择：

（1） 这组唱片中的歌曲必须按照它们创作的顺序排序；

（2） 包含歌曲的总数尽可能多。

输入 n，m，t，和 n 首歌曲的长度，它们按照创作顺序排序，没有一首歌超出一张唱片的长度，而且不可能将所有歌曲的放在唱片中。输出所能包含的最多的歌曲数目。

（1≤n, m, t≤20）

[算法分析]

本题要求唱片中的歌曲必须按照它们创作顺序排序，这就满足了动态规划的无后效性要求，启发我们采用动态规划进行解题。

分析可知，该问题具有最优子结构性质，即：设最优录制方案中第 i 首歌录制的位置是从第 j 张唱片的第 k 分钟开始的，那么前 j-1 张唱片和第 j 张唱片的前 k-1 分钟是前 1..i-1 首歌的最优录制方案，也就是说，问题的最优解包含了子问题的最优解。

设 n 首歌曲按照写作顺序排序后的长度为 long[1..n]，则动态规划的状态表示描述为：

g[i, j, k]，0≤i≤n，0≤j≤m，0≤k<t，表示前 i 首歌曲，用 j 张唱片另加 k 分钟来录制，最多可以录制的歌曲数目，则问题的最优解为 g[n,m,0]。由于歌曲 i 有发行和不发行两种情况，而且还要分另加的 k 分钟是否能录制歌曲 i。这样我们可以得到如下的状态转移方程和边界条件：

当 k≥long[i]，i≥1 时：

g[i, j, k]=max{g[i-1,j,k-long[i]]，g[i-1,j,k]}

当 k<long[i]，i≥1 时：

g[i, j, k]=max{g[i-1,j-1,t-long[i]]，g[i-1,j,k]}

规划的边界条件为：

当 0≤k<t 时：g[0,0,k]=0;

我们来分析上述算法的时间复杂度，上述算法的状态总数为 O(n*m*t)，每个状态转移的状态数为 O(1)，每次状态转移的时间为 O(1)，所以总的时间复杂度为 O(n*m*t)。由于 n，m，t 均不超过 20，所以可以满足要求。

[算法优化]

当数据规模较大时，上述算法就无法满足要求，我们来考虑通过改进状态表示提高算法的时间效率。

本题的最优目标是用给定长度的若干张唱片录制尽可能多的歌曲，这实际上等价于在录制给定数量的歌曲时尽可能少地使用唱片。所谓“尽可能少地使用唱片”，就是指使用的完整的唱片数尽可能少，或是在使用的完整的唱片数相同的情况下，另加的分钟数尽可能少分析可知，在这样的最优目标之下，该问题同样具有最优子结构性质，即：设 D 在前 i 首歌中选取 j 首歌录制的最少唱片使用方案，那么若其中选取了第 i 首歌，则 D-{i}是在前 i-1 首歌中选取 j-1 首歌录制的最少唱片使用方案，否则 D 前 i-1 首歌中选取 j 首歌录制的最少唱片使用方案，同样，问题的最优解包含了子问题的最优解。

改进的状态表示描述为：

g[i, j]=(a, b)，0≤i≤n，0≤j≤i，0≤a≤m，0≤b≤t，表示在前 i 首歌曲中选取 j 首录制所需的最少唱片为：a 张唱片另加 b 分钟。由于第 i 首歌分为发行和不发行两种情况，这样我们可以得到如下的状态转移方程和边界条件：

g[i, j]=min{g[i-1,j]，g[i-1,j-1]+long[i]}

其中(a, b)+long[i]=(a′, b′)的计算方法为：

当 long[i]≤t-b 时： a′=a;　b′=b+long[i];

当 long[i]>t-b 时： a′=a+1;　b′=long[i];

规划的边界条件：

g[i,0]=(0,0)　0≤i≤n

这样题目所求的最大值是：ans=max{k| g[n, k]≤(m-1,t)}

改进后的算法，状态总数为 $O(n^2)$，每个状态转移的状态数为 O(1)，每次状态转移的时间为 O(1)，所以总的时间复杂度为 $O(n^2)$。值得注意的是，算法的空间复杂度也由改进前的 O(m*n*t)降至优化后的 $O(n^2)$。

（程序及优化前后的运行结果比较见附件）

通过对本题的优化，我们认识到：应用不同的状态表示方法设计出的动态规划算法的性能也迥然不同。改进状态表示可以减少状态总数，进而降低算法的时间复杂度。在降低算法的时间复杂度的同时，也降低了算法的空间复杂度。因此，减少状态总数在动态规划的优化中占有重要的地位。

2、选择适当的规划方向

动态规划方法的实现中，规划方向的选择主要有两种：顺推和逆推。在有些情况下，选取不同的规划方向，程序的时间效率也有所不同。一般地，若初始状态确定，目标状态不确定，则应考虑采用顺推，反之，若目标状态确定，而初始状态不确定，就应该考虑采用逆推。那么，若是初始状态和目标状态都已确定，一般情况下顺推和逆推都可以选用，但是，能否考虑选用双向规划呢？

双向搜索的方法已为大家所熟知，它的主要思想是：在状态空间十分庞大，而初始状态和目标状态又都已确定的情况下，由于扩展的状态量是指数级增长的，于是为了减少状态的规模，分别从初始状态和目标状态两个方向进行扩展，并在两者的交汇处得到问题的解。

上述优化思想能否也应用到动态规划之中呢？来看下面这个例子。

例二、 Divide (Merc`2000)

[问题描述]

有价值分别为 1..6 的大理石各 a[1..6]块，现要将它们分成两部分，使得两部分价值和相等，问是否可以实现。其中大理石的总数不超过 20000。(英文试题详见附件)

[算法分析]

令 S=∑(i*a[i])，若 S 为奇数，则不可能实现，否则令 Mid=S/2，则问题转化为能否从给定的大理石中选取部分大理石，使其价值和为 Mid。

这实际上是母函数问题，用动态规划求解也是等价的。

m[i, j]，0≤i≤6，0≤j≤Mid，表示能否从价值为 1..i 的大理石中选出部分大理石，使其价值和为 j，若能，则用 true 表示，否则用 false 表示。则状态转移方程为：

m[i, j]=m[i, j]　OR　m[i-1,j-i*k]　　(0≤k≤a[i])

规划的边界条件为：m[i,0]=true;　0≤i≤6

若 m[i, Mid]=true，0≤i≤6，则可以实现题目要求，否则不可能实现。

我们来分析上述算法的时间性能，上述算法中每个状态可能转移的状态数为 a[i]，每次状态转移的时间为 O(1)，而状态总数是所有值为 true 的状态的总数，实际上就是母函数中项的数目。

[算法优化]

实践发现：本题在 i 较小时，由于可选取的大理石的价值品种单一，数量也较少，因此值为 true 的状态也较少，但随着 i 的增大，大理石价值品种和数量的增多，值为 true 的状态也急剧增多，使得规划过程的速度减慢，影响了算法的时间效率。

另一方面，我们注意到我们关心的仅是能否得到价值和为 Mid 的值为 true 的状态，那么，我们能否从两个方向分别进行规划，分别求出从价值为 1..3 的大理石中选出部分大理石所能获得的所有价值和，和从价值为 4..6 的大理石中选出部分大理石所能获得的所有价值和。最后通过判断两者中是否存在和为 Mid 的价值和，由此，可以得出问题的解。

状态转移方程改进为：

当 i≤3 时：

m[i, j]=m[i, j] OR m[i-1,j-i*k]　　(1≤k≤a[i])

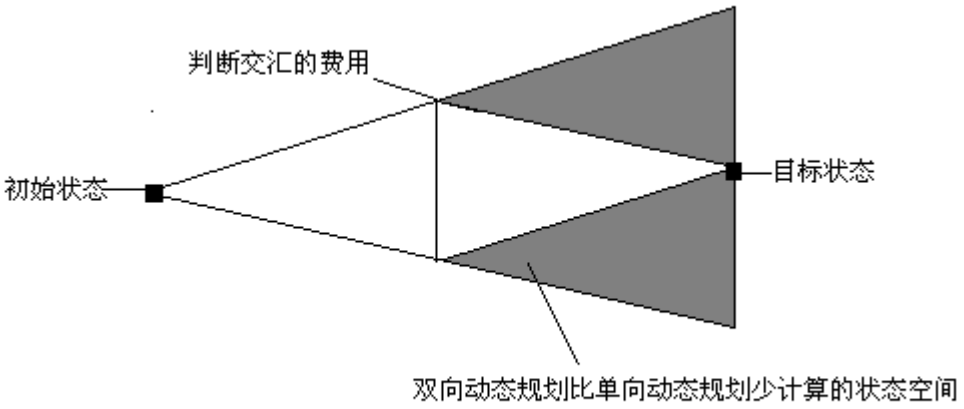
当 i>3 时：

m[i, j]=m[i, j] OR m[i+1,j-i*k]　　(1≤k≤a[i])

规划的边界条件为：m[i,0]=true； 0≤i≤7

这样，若存在 k，使得 m[3,k]=true, m[4,Mid-k]=true，则可以实现题目要求，否则无法实现。

（程序及优化前后的运行结果比较见附件）



从上图可以看出双向动态规划与单向动态规划在计算的状态总数上的差异。

回顾本题的优化过程可以发现：本题的实际背景与双向搜索的背景十分相似，同样有庞大的状态空间，有确定的初始状态和目标状态，状态量都迅速增长，而且可以实现交汇的判断。因此，由本题的优化过程，我们认识到，双向扩展以减少状态量的方法不仅适用于搜索，同样适用于动态规划。这种在不同解题方法中，寻找共通的属性，从而借用相同的优化思想，可以使我们不断创造出新的方法。

3.2 减少每个状态转移的状态数

在使用动态规划方法解题时，对当前状态的计算都是进行一些决策并引用相应的已

经计算过的状态，这个过程称为“状态转移”。因此，每个状态可能做出的决策数，也就是每个状态可能转移的状态数是决定动态规划算法时间复杂度的一个重要因素。本节将讨论减少每个状态可能转移的状态数的一些方法。

1、四边形不等式和决策的单调性

例三、石子合并问题(NOI`95)

[问题描述]

在一个操场上摆放着一排 n（n≤20）堆石子。现要将石子有次序地合并成一堆。规定每次只能选相邻的 2 堆石子合并成新的一堆，并将新的一堆石子数记为该次合并的得分。

试编程求出将 n 堆石子合并成一堆的最小得分和最大得分以及相应的合并方案。

[算法分析]

这道题是动态规划的经典应用。由于最大得分和最小得分是类似的，所以这里仅对最小得分进行讨论。设 n 堆石子依次编号为 1，2，.....，n。各堆石子数为 d[1..n]，则动态规划的状态表示为：

m[i,j]，1≤i≤j≤n，表示合并 d[i..j]所得到的最小得分，则状态转移方程和边界条件为：

m[i,j]=0　　i=j

$m[i,j]=\min_{i<k\le j}\{m[i,k-1]+m[k,j]+\sum_{l=i}^{j}d[l]\}$　　i < j

同时令 s[i,j]=k，表示合并的断开位置，便于在计算出最优值后构造出最优解。

上式中 $\sum_{l=i}^{j}d[l]$ 的计算，可在预处理时计算 $t[i]=\sum_{j=1}^{i}d[j]$，i=1..n；t[0]=0, 则：

$$\sum_{l=i}^{j}d[l]=t[j]-t[i-1]$$

上述算法的状态总数为 $O(n^2)$，每个状态转移的状态数为 O(n)，每次状态转移的时间为 O(1)，所以总的时间复杂度为 $O(n^3)$。

[算法优化]

当函数 w[i,j]满足 $w[i,j]+w[i',j']\le w[i',j]+w[i,j'],\ i\le i'\le j\le j'$ 时，称 w 满足四边形不等式[2]。

当函数 w[i,j]满足 w[i′,j]≤w[i,j′] $,\ i\le i'\le j\le j'$ 时称 w 关于区间包含关系单调。

在石子归并问题中，令 w[i,j]= $\sum_{l=i}^{j}d[l]$，则 w[i,j]满足四边形不等式，同时由 d[i]≥0,t[i]≥0 可知 w[i,j]满足单调性。

m[i,j]=0　　i=j

$m[i,j]=\min_{i<k\le j}\{m[i,k-1]+m[k,j]\}+w[i,j]$　　i<j　　……①

对于满足四边形不等式的单调函数 w，可推知由递推式①定义的函数 m[i,j]也满足四边形不等式，即 $m[i,j]+m[i',j']\le m[i',j]+m[i,j'],\ i\le i'\le j\le j'$。这一性质可用数学归纳法证明如下：

我们对四边形不等式中“长度”l=j′-i 进行归纳：

当 i=i′或 j=j′时，不等式显然成立。由此可知，当 l≤1 时，函数 m 满足四边形不等式。

下面分两种情形进行归纳证明：

情形 1：i<i′=j<j′

在这种情形下，四边形不等式简化为如下的反三角不等式：m[i,j]+m[j,j′]≤m[i,j′]，设 k=max{p | m[i,j′]=m[i,p-1]+m[p,j′]+w[i,j′] }，再分两种情形 k≤j 或 k>j。下面只讨论 k≤j，k>j 的情况是类似的。

情形 1.1：k≤j，此时：

$$\begin{aligned}&m[i,j]+m[j,j']\le w[i,j]+m[i,k-1]+m[k,j]+m[j,j']\\&\le w[i,j']+m[i,k-1]+m[k,j]+m[j,j']\\&\le w[i,j']+m[i,k-1]+m[k,j']\\&=m[i,j']\end{aligned}$$

情形 2：i<i′<j<j′

设 y=max{p | m[i′,j]=m[i′,p-1]+m[p,j]+w[i′,j] }

z=max{p | m[i,j′]=m[i,p-1]+m[p,j′]+w[i,j′] }

仍需再分两种情形讨论，即 z≤y 或 z>y。下面只讨论 z≤y，z>y 的情况是类似的。

由 i<z≤y≤j 有：

$$\begin{aligned}&m[i,j]+m[i',j']\le w[i,j]+m[i,z-1]+m[z,j]+w[i',j']+m[i',y-1]+m[y,j']\\&\le w[i,j']+w[i',j]+m[i',y-1]+m[i,z-1]+m[z,j]+m[y,j']\\&\le w[i,j']+w[i',j]+m[i',y-1]+m[i,z-1]+m[y,j]+m[z,j']\\&=m[i,j']+m[i',j]\end{aligned}$$

综上所述，m[i,j]满足四边形不等式。

令 s[i,j]=max{k | m[i,j]=m[i,k-1]+m[k,j]+w[i,j] }

由函数 m[i,j]满足四边形不等式可以推出函数 s[i,j]的单调性，即

$$s[i,j]\le s[i,j+1]\le s[i+1,j+1],\quad i\le j$$

当 i=j 时，单调性显然成立。因此下面只讨论 i<j 的情形。由于对称性，只要证明 s[i,j]≤s[i,j+1]。

令 $m_k[i,j]=m[i,k-1]+m[k,j]+w[i,j]$。要证明 s[i,j]≤s[i,j+1]，只要证明对于所有 i<k≤k′≤j 且 $m_{k'}[i,j]\le m_k[i,j]$，有：$m_{k'}[i,j+1]\le m_k[i,j+1]$。

事实上，我们可以证明一个更强的不等式

$$m_k[i,j]-m_{k'}[i,j]\le m_k[i,j+1]-m_{k'}[i,j+1]$$

也就是：$m_k[i,j]+m_{k'}[i,j+1]\le m_k[i,j+1]+m_{k'}[i,j]$

利用递推定义式将其展开整理可得：m[k,j]+m[k′,j+1]≤m[k′,j]+m[k,j+1]，这正是 k≤k′≤j<j+1 时的四边形不等式。

综上所述，当 w 满足四边形不等式时，函数 s[i,j]具有单调性。

于是，我们利用 s[i,j]的单调性，得到优化的状态转移方程为：

m[i,j]=0　　i=j

$$m[i,j]=\min_{s[i,j-1]\le k\le s[i+1,j]}\{m[i,k-1]+m[k,j]\}+w[i,j]\qquad i<j$$

用类似的方法可以证明，对于最大得分问题，也可采用同样的优化方法。

改进后的状态转移方程所需的计算时间为

$$O\left(\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}(1+s[i+1,j]-s[i,j-1])\right)=O\left(\sum_{i=1}^{n-1}(n-i+s[i+1,n]-s[1,n-i])\right)=O(n^2)$$

（程序及优化前后的运行结果比较见附件）

上述方法利用四边形不等式推出最优决策的单调性，从而减少每个状态转移的状态数，

降低算法的时间复杂度。

上述方法是具有普遍性的。对于状态转移方程与①式类似，且 w[i,j]满足四边形不等式的动态规划问题，都可以采用相同的优化方法，如最优二叉排序树（NOI`96）等。下面再举一例。

例四、邮局(IOI`2000)

[问题描述]

按照递增顺序给出一条直线上坐标互不相同的 n 个村庄，要求从中选择 p 个村庄建立邮局，每个村庄使用离它最近的那个邮局，使得所有村庄到各自所使用的邮局的距离总和最小。

试编程计算最小距离和，以及邮局建立方案。

[算法分析]

本题也是一道动态规划问题，详细分析请看文本附件（邮局解题报告）。将 n 个村庄按坐标递增依次编号为 1，2，……，n，各个邮局的坐标为 d[1..n]，状态表示描述为：m[i,j]表示在前 j 个村庄建立 i 个邮局的最小距离和。所以，m[p,n]即为问题的解，且状态转移方程和边界条件为：

m[1,j]=w[1,j]

$$m[i,j]=\min_{i-1\le k\le j-1}\{m[i-1,k]+w[k+1,j]\}\qquad i\le j$$

其中 w[i,j]表示在 d[i..j]之间建立一个邮局的最小距离和，可以证明，当仅建立一个邮局时，最优解出现在中位数，即设建立邮局的村庄为 k，则 $k=\lfloor (i+j)/2\rfloor$ 或 $k=\lceil (i+j)/2\rceil$，于是，我们有：

$$w[i,j]=\sum_{l=i}^{j}|d[l]-d[k]|\quad ，\quad k=\lfloor (i+j)/2\rfloor \text{或} k=\lceil (i+j)/2\rceil$$

同时，令 s[i,j]=k，记录使用前 i-1 个邮局的村庄数，便于在算出最小距离和之后构造最优建立方案。

上述算法中 w[i,j]可通过 O(n)时间的预处理，在 O(1)的时间内算出，所以，该算法的状态总数为 O(n*p)，每个状态转移的状态数为 O(n)，每次状态转移的时间为 O(1)，该算法总的时间复杂度为 $O(p*n^2)$。

[算法优化]

本题的状态转移方程与①式十分相似，因此我们猜想其决策是否也满足单调性，即

$$s[i-1,j]\le s[i,j]\le s[i,j+1]$$

首先，我们来证明函数 w 满足四边形不等式，即：

$$w[i,j]+w[i',j']\le w[i',j]+w[i,j']\quad ,i\le i'\le j\le j'$$

设 $y=\lfloor (i'+j)/2\rfloor$，$z=\lfloor (i+j')/2\rfloor$，下面分为两种情形，z≤y 或 z>y，下面仅讨论 z≤y，z>y 的情况是类似的。

由 i≤z≤y≤j 有：

$$w[i,j]+w[i',j'] \le \sum_{l=i}^{j} |d[l]-d[z]| + \sum_{l=i'}^{j'} |d[l]-d[y]|$$

$$\le \sum_{l=i}^{j} |d[l]-d[z]| + \sum_{l=i'}^{j'} |d[l]-d[y]| + \sum_{l=j+1}^{j'} |d[l]-d[z]| - \sum_{l=j+1}^{j'} |d[l]-d[y]|$$

$$= \sum_{l=i}^{j'} |d[l]-d[z]| + \sum_{l=i'}^{j} |d[l]-d[y]|$$

$$= w[i',j]+w[i,j']$$

接着，我们用数学归纳法证明函数 m 也满足四边形不等式。对四边形不等式中"长度"l=j'-i 进行归纳：

当 i=i'或 j=j'时，不等式显然成立。由此可知，当 l≤1 时，函数 m 满足四边形不等式。

下面分两种情形进行归纳证明：

情形 1：i<i'=j<j'，即 m[i,j]+m[j,j'] ≤m[i,j']，

设 k=max{p | m[i,j']=m[i,p-1]+m[p,j']+w[i,j'] }，再分两种情形 k≤j 或 k>j。下面只讨论 k≤j，k>j 的情况是类似的。

$$m[i,j]+m[j,j']$$
$$\le m[i-1,k]+w[k+1,j]+m[j-1,j-1]+w[j,j']$$
$$\le m[i-1,k]+w[k+1,j']$$
$$= m[i,j']$$

情形 2：i<i'<j<j'

设　y=max{p | m[i',j]=m[i'-1,p]+w[p+1,j] }

　　z=max{p | m[i,j']=m[i-1,p]+w[p+1,j'] }

仍需再分两种情形讨论，即 z≤y 或 z>y。

情形 2.1，当 z≤y<j<j'时：

$$m[i,j]+m[i',j']$$
$$\le m[i'-1,y]+w[y+1,j']+m[i-1,z]+w[z+1,j]$$
$$\le m[i'-1,y]+m[i-1,z]+w[y+1,j]+w[z+1,j']$$
$$= m[i',j]+m[i,j']$$

情形 2.2，当 i-1<i'-1≤y<z<j'时：

$$m[i,j]+m[i',j']$$
$$\le m[i-1,y]+w[y+1,j]+m[i'-1,z]+w[z+1,j']$$
$$\le m[i-1,z]+m[i'-1,y]+w[y+1,j]+w[z+1,j'];$$
$$= m[i,j']+m[i',j]$$

最后，我们证明决策 s[i,j]满足单调性。

为讨论方便，令 $m_k[i,j]=m[i-1,k]+w[k+1,j]$;

我们先来证明 s[i-1,j]≤s[i,j]，只要证明对于所有 i≤k<k'<j 且 $m_{k'}[i-1,j]\le m_k[i-1,j]$，有：$m_{k'}[i,j]\le m_k[i,j]$。

类似地，我们可以证明一个更强的不等式

$$m_k[i-1,j]-m_{k'}[i-1,j]\le m_k[i,j]-m_{k'}[i,j]$$

也就是：　　$m_k[i-1,j]+m_{k'}[i,j]\le m_k[i,j]+m_{k'}[i-1,j]$

利用递推定义式展开整理的：m[i-2,k]+m[i-1,k']≤m[i-1,k]+m[i-2,k']，这就是 i-2<i-1<k<k'时 m 的四边形不等式。

我们再来证明 s[i,j]≤s[i,j+1]，与上文类似，设 k<k'<j，则我们只需证明一个更强的不等式：　　$m_k[i,j]-m_{k'}[i,j]\le m_k[i,j+1]-m_{k'}[i,j+1]$

也就是：　　$m_k[i,j]+m_{k'}[i,j+1]\le m_k[i,j+1]+m_{k'}[i,j]$

利用递推定义式展开整理的：w[k+1,j]+w[k'+1,j+1]≤w[k+1,j+1]+w[k'+1,j]，这就是 k+1<k'+1<j<j+1 时 w 的四边形不等式。

综上所述，该问题的决策 s[i,j]具有单调性，于是优化后的状态转移方程为：

m[1,j]=w[1,j]

$$m[i,j]=\min_{s[i-1,j]\le k\le s[i,j+1]}\{m[i-1,k]+w[k+1,j]\}$$　　i≤j

s[i,j]=k

同上文所述，优化后的算法时间复杂度为 O(n*p)。

（程序及优化前后的运行结果比较见附件）

四边形不等式优化的实质是对结果的充分利用。它通过分析状态值之间的特殊关系，推出了最优决策的单调性，从而在计算当前状态时，利用已经计算过的状态所做出的最优决策，减少了当前的决策量。这就启发我们，在应用动态规划解题时，不仅可以实现状态值的充分利用，也可以实现最优决策的充分利用。这实际上是从另一个角度实现了"减少冗余"。

2、决策量的优化

通过分析问题最优解所具备的性质，从而缩小有可能产生最优解的决策集合，也是减少每个状态可能转移的状态数的一种方法。

大家所熟悉的 NOI`96 中的添加号问题，正是从"所得的和最小"这一原则出发，仅在等分点的附近添加号，从而大大减少了每个状态转移的状态数，降低了算法的时间复杂度让我们在再来看一例。

例五、石子归并的最大得分问题

[问题描述]

见例三，本例只考虑最大得分问题。

[算法分析]

设 n 堆石子依次编号为 1，2，.....，n。各堆石子数为 d[1..n]，则动态规划的状态表示为：

m[i,j]，1≤i≤j≤n，表示合并 d[i..j]所得到的最大得分，则状态转移方程和边界条件为：

m[i,j]=0　　i=j

$$m[i,j]=\max_{i<k\le j}\{m[i,k-1]+m[k,j]+\sum_{l=i}^{j}d[l]\}$$　　i < j

同时令 s[i,j]=k，表示合并的断开位置，便于在计算出最优值后构造出最优解。

该算法的时间复杂度为 $O(n^3)$。

[算法优化]

仔细分析问题，可以发现：s[i,j]要么等于 i+1，要么等于 j，即：

$$\max_{i<k\le j}\{m[i,k-1]+m[k,j]\}=\max\{m[i,j-1],m[i+1,j]\},i<j$$

证明可以采用反证法，设使 m[i,j]达到最大值的断开位置为 p，且 i+1<p<j，

$y=\sum_{l=i}^{p-1}a[l]$，$z=\sum_{l=p}^{j}a[l]$，下面分为 2 种情形讨论。

情形 1、y≥z

由 p<j，可设 s[p,j]=k，则相应的合并方式可以表示为：

（（a[i]…a[p-1]）（(a[p]...a[k-1]) (a[k]..a[j])））

相应的得分为：$T = m[i, p-1] + m[p, k-1] + m[k, j] + y + z + z$ .....①

下面考虑另一种合并方案 s′[i,j]=k，s′[i,k]=p，表示为：

(((a[i]…a[p-1]) (a[p]...a[k-1])) (a[k]..a[j])　)

相应的得分为：$T' = m[i, p-1] + m[p, k-1] + m[k, j] + y + y + z + \sum_{l=p}^{k-1} a[l]$ ……

②

由 y≥z 可得，T<T′，这与使 m[i,j]达到最大值的断开位置为 p 的假设矛盾。

情形 2、y<z　与情形 1 类似。

于是，状态转移方程优化为：

m[i,j]=0　　i=j

$m[i, j] = \max\{m[i, j-1] + m[i+1, j]\} + \sum_{l=i}^{j} d[l]$　　i < j

优化后每个状态转移的状态数减少为 O(1)，算法总的时间复杂度也降为 $O(n^2)$。

（程序及优化前后的运行结果比较见附件）

本题的优化过程是通过对问题最优解性质的分析，找出最优决策必须满足的必要条件，这与搜索中的最优性剪枝的思想十分类似，由此我们再次看到了相同的优化思想应用于不同的算法设计方法。同时，我们也认识到：动态规划的优化必须建立在全面细致分析问题的基础上，只有深入分析问题的属性，挖掘问题的实质，才能实现算法的优化。

3、合理组织状态

在动态规划求解的过程中，需要不断地引用已经计算过的状态。因此，合理地组织已经计算出的状态有利于提高动态规划的时间效率。

例六、求最长单调上升子序列

[问题描述]

给出一个由 n 个数组成的序列 x[1..n]，找出它的最长单调上升子序列。即求最大的 m 和 $a_1,a_2……,a_m$，使得 $a_1<a_2<……<a_m$ 且 $x[a_1]<x[a_2]<……<x[a_m]$。

[算法分析]

这也是一道动态规划的经典应用。动态规划的状态表示描述为：

m[i]，1≤i≤n，表示以 x[i]结尾的最长上升子序列的长度，则问题的解为 max{m[i],1≤i≤n}，状态转移方程和边界条件为：

m[i]=1+max{0, m[k]|x[k]<x[i] , 1≤k<i }

同时当 m[i]>1 时，令 p[i]=k，表示最优决策，以便在计算出最优值后构造最长单调上升子序列。

上述算法的状态总数为 O(n)，每个状态转移的状态数最多为 O(n)，每次状态转移的时间为 O(1)，所以算法总的时间复杂度为 $O(n^2)$。

[算法优化]

我们先来考虑以下两种情况：

1、若 x[i]<x[j]，m[i]=m[j]，则 m[j]这个状态不必保留。因为，可以由状态 m[j]转移得到的状态 m[k] (k>j，k>i)，必有 x[k]>x[j]>x[i]，则 m[k]也能由 m[i]转移得到；另一方面，可以由状态 m[i]转移得到的状态 m[k] (k>j，k>i)，当 x[j]>x[k]>x[i]时，m[k]就无法由 m[j]转移得到。

由此可见，在所有状态值相同的状态中，只需保留最后一个元素值最小的那个状态即可。

2、若 x[i]<x[j]，m[i]>m[j]，则 m[j]这个状态不必保留。因为，可以由状态 m[j]转移得到的状态 m[k] (k>j，k>i)，必有 x[k]>x[j]>x[i]，则 m[k]也能由 m[i]转移得到，而且 m[i]>m[j]，所以 m[k]≥m[i]+1>m[j]+1，则 m[j]的状态转移是没有意义的。

综合上述两点，我们得出了状态 m[k]需要保留的必要条件：不存在 i 使得：x[i]<x[k]且 m[i]≥m[k]。于是，我们保留的状态中不存在相同的状态值，且随着状态值的增加，最后一个元素的值也是单调递增的。

也就是说，设当前保留的状态集合为 S，则 S 具有以下性质 D：

对于任意 i∈S, j∈S, i≠j 有：m[i]≠m[j]，且若 m[i]<m[j]，则 x[i]<x[j]，否则 x[i]>x[j]。

下面我们来考虑状态转移：假设当前已求出 m[1..i-1]，当前保留的状态集合为 S，下面计算 m[i]。

1、若存在状态 k∈S，使得 x[k]=x[i]，则状态 m[i]必定不需保留，不必计算。因为，不妨设 m[i]=m[j]+1，则 x[j]<x[i]=x[k]，j∈S，j≠k，所以 m[j]<m[k]，则 m[i]=m[j]+1≤m[k]，所以状态 m[i]不需保留。

2、否则，m[i]=1+max{m[j]| x[j]<x[i], j∈S}。我们注意到满足条件的 j 也满足 x[j]=max{x[k]|x[k]<x[i], k∈S}。同时我们把状态 i 加入到 S 中。

3、若 2 成立，则我们往 S 中增加了一个状态，为了保持 S 的性质，我们要对 S 进行维护，若存在状态 k∈S，使得 m[i]=m[k]，则我们有 x[i]<x[k]，且 x[k]=min{x[j]| x[j]>x[i], j∈S}。于是状态 k 应从 S 中删去。

于是，我们得到了改进后的算法：

```
For i:=1 to n do
{
  找出集合 S 中的 x 值不超过 x[i]的最大元素 k;
  if  x[k]<x[i]  then
  {
    m[i]:=m[k]+1;
    将状态 i 插入集合 S;
    找出集合 S 中的 x 值大于 x[i]的最小元素 j;
    if  m[j]=m[i]  then   将状态 j 从 S 中删去;
  }
}
```

从性质 D 和算法描述可以发现，S 实际上是以 x 值为关键字（也是以 m 值为关键字）的有序集合。若使用平衡树实现有序集合 S，则该算法的时间复杂度为 $O(n*\log_2 n)$。本题优化后，每个状态转移的状态数仅为 O(1)，而每次状态转移的时间变为 $O(\log_2 n)$，这也体现了上文所提到的优化中不同因素之间的矛盾，但从总体上看，算法的时间复杂度是降低了。

（程序及优化前后的运行结果比较见附件）

回顾本题的优化过程，首先通过分析状态之间的分析，减少需要保留的状态数，同时发现需要保留状态的单调性，从而减少了每个状态可能转移的状态数，并通过高效数据结构平衡树组织当前保留的状态，实现算法的优化。

通过对本题的优化，我们认识到减少保留的状态数，合理组织已经计算出的状态可以实现减少每个状态可能转移的状态数，同时，选取恰当的数据结构也是算法优化的一个重

要原则，在下文的阐述中，还会看到借助数据结构实现算法优化。

4、细化状态转移

所谓“细化状态转移”，就是将原来的一次状态转移细化成若干次状态转移，其目的在于减少总的状态转移的次数。在优化前，问题的决策一般都是复合决策，也就是一些子决策的排列，因此决策的规模较大，每个状态可能转移的状态数也就较多，优化的方法就是将每个复合决策细化成若干个子决策，并在每个子决策后面增设一个状态，这样，后面的子决策只在前面的子决策达到最优解时才进行转移，因此在优化后，虽然，状态总数增加了，但是总的状态转移次数却减少了，算法总的复杂度也就降低了。

应该注意的是：上述优化应该满足一个条件，即原来每个复合决策的各个子决策之间也满足最优化原理和无后效性，也就是说：复合最优决策的子决策也是最优决策；前面的子决策不影响后面的子决策。

上述优化方法再一次体现了实现一个因素的优化要以增大另一个因素作为代价，但是，算法总的时间复杂度的降低才是我们的真正目的。

3.3 减少状态转移的时间

我们知道，状态转移是动态规划的基本操作，因此，减少每次状态转移所需的时间，对提高算法的时间效率具有重要的意义。

状态转移主要有两个部分构成：

1. 进行决策：通过当前状态和选取的决策计算出需要引用的状态。
2. 计算递推式：根据递推式计算当前状态值。其中主要操作是常数项的计算。

本节将分别讨论提高这两部分时间效率的一些方法。

1、减少决策时间

例七、LOSTCITY (NOI`2000)

[问题描述]

现给出一张单词表、特定的语法规则和一篇文章：

文章和单词表中只含 26 个小写英文字母 a…z。

单词表中的单词只有名词，动词和辅词这三种词性，且相同词性的单词互不相同。单词的长度均不超过 20。

语法规则可简述为：名词短语：任意个辅词前缀接上一个名词；动词短语：任意个辅词前缀接上一个动词；句子：以名词短语开头，名词短语与动词短语相间连接而成。

文章的长度不超过 1000。且已知文章是由有限个句子组成的，句子只包含有限个单词。

编程将这篇文章划分成最少的句子，在此前提之下，要求划分出的单词数最少。

[算法分析]

这是也是一道动态规划问题。我们分别用 v,u,a 表示动词，名词和副词，给出的文章用 L[1..M]表示，则状态表示描述为：

F(v,i)：表示 L 前 i 个字符划分为以动词结尾（当 i<>M 时，可带任意个辅词后缀）的最优分解方案下划分的句子数与单词数；

F(u,i)：表示 L 前 i 个字符划分为以名词结尾（当 i<>M 时，可带任意个辅词后缀）的最

优分解方案下划分的句子数与单词数。

过去的分解方案仅通过最后一个非辅词的词性影响以后的决策，所以这种状态表示满足无后效性，

状态转移方程为：

F(v,i)=min{ F(n,j)+(0,1)， L(j+1..i)为动词；

F(v,j)+(0,1)， L(j+1..i)为辅词，i<>M；}

F(n,i)=min{ F(n,j)+(1,1)，L(j+1..i)为名词；

F(v,j)+(0,1)，L(j+1..i)为名词；

F(n,j)+(0,1)，L(j+1..i)为辅词，i<>M；}

边界条件：F(v,0)=(1,0)； F(n,0)=(∞, ∞)；

问题的解为：min{ F(v,M), F(u,M) }；

上述算法中，状态总数为 O(M)，每个状态转移的状态数最多为 20，在进行状态转移时，需要查找 L[j+1..i]的词性，根据其词性做出相应的决策，并引用相应的状态。下面就通过不同的方法查找 L[j+1..i]的词性，比较它们的时间复杂度。

[算法实现]

设单词表的规模为 N，首先我们对单词表进行预处理，将单词按字典顺序排序并合并具有多重词性的单词。在查找词性时有以下几种方法：

方法 1、采用顺序查找法。最坏情况下需要遍历整个单词表，因此最坏情况下的时间复杂度为 O(20*N*M)，比较次数最多可达 $1000*5k*20=10^8$，当数据量较大时效率较低。

方法 2、采用二分查找法。最坏情况下的时间复杂度为 $O(20*M*\log_2 N)$，最多比较次数降为 $5k*20*\log_2 1000=10^6$，完全可以忍受。

集合查找最为有效的方法要属采用哈希表了。

方法 3、采用哈希表查找单词的词性。首先将字符串每四位折叠相加计算关键值 k，然后用双重哈希法计算哈希函数值 h(k)。采用这种方法，通过 O(N)时间的预处理构造哈希表，每次查找只需 O(1)的时间，因此，算法的时间复杂度为 O(20*M+N)=O(M)。

采用哈希表是进行集合查找的一般方法，对于以字符串为元素的集合还有更为高效的方法，即采用检索树[3]。

方法四、采用检索树查找单词的词性。由于每个状态在进行状态转移时需要查找的所有单词都是分布在同一条从树根到叶子的路径上的，因此，如果选取从树根走一条路径到叶子作为基本操作，则每个状态进行状态转移时的最多 20 次单词查找只需 O(1)的时间，另外，建立检索树需要 O(N)的时间，因此，算法总的时间复杂度虽然仍为 O(M)，但是由于时间复杂度的常数因子小于方法三，因此实际测试的速度也最快。

（程序及四种方法的运行结果的比较见附件）

从本题的优化过程可以看出：采用正确的数据结构是算法优化的重要原则，在动态规划算法的优化中也同样适用。方法 3 使用了哈希表这一高效的集合查找数据结构，方法四使用的针对性更强的检索树，使得算法的时间效率得到了提高。

2、减少计算递推式的时间

计算递推式的主要操作是对常数项的计算，因此减少计算递推式所需的时间主要是指减少计算常数项的时间。

例八、公路巡逻 (CTSC`2000)

[问题描述]

在一条没有分岔的公路上有 n（n≤50）个关口，相邻两个关口之间的距离都是

10km。所有车辆在这条公路上的最低速度为 60km/h，最高速度为 120km/h，且只能在关口出改变速度。

有 m(m≤300)辆巡逻车分别在时刻 T_i 从第 n_i 个关口出发，匀速行驶到达第 n_i+1 个关口，路上耗费时间为 t_i 秒。

两辆车相遇指他们之间发生超车现象或同时到达某个关口。

求一辆于 6 点整从第 1 个关口出发去第 n 个关口的车（称为目标车）最少会与多少辆巡逻车相遇，以及在此情况下到达第 n 个关口的最早时刻。

假设所有车辆到达关口的时刻都是整秒。

[算法分析]

本题也是用动态规划来解。问题的状态表示描述为：

F(i,T)表示在时刻 T 到达第 i 个关口的途中最少已与巡逻车相遇的次数。则状态转移方程和边界条件为：

F(i,T)=min{F(i-1,T-Tk)+w(i-1,T-Tk,T)，300≤Tk≤600}　2≤i≤n

边界条件：F(1,06:00:00)=0;

问题的解为：min{F(n,T)}

其中，函数 w(i-1,T-Tk,T)是计算目标车于时刻 T-Tk 从第 i-1 个关口出发，于时刻 T 到达第 i 个关口，途中与巡逻车相遇的次数。

下面来分析上述算法的时间复杂度，问题的阶段数为 n，第 i 个阶段的状态数为(i-1)*300，则状态总数为：$O(\sum_{i=1}^{n}(i-1)*300)=O(300*\frac{n*(n-1)}{2})=O(150n^2)$，每个状态转移的状态数为 300，每次状态转移所需的时间关键取决与函数 w 的计算。下面比较采用不同的计算方法时，时间复杂度的差异。

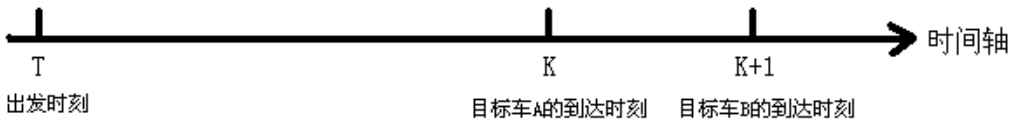
[函数 w 的计算]

方法 1、在每个决策中都进行一次计算，对所有从第 i 个关口出发的巡逻车进行判断，这样平均每次状态转移的时间为 O(1+m/n)，由 M 的最大值为 300，算法总的时间复杂度为：

$$O\left(150n^2*300*(1+\frac{m}{n})\right)=O\left(\frac{m^2n^2}{2}+\frac{m^3n}{2}\right)=O(m^3n)$$

方法 2、仔细观察状态转移方程可以发现，在对状态 F(i,T)进行转移时，所计算的函数 w 都是从第 i 个关口出发的，而且出发时刻都是 T，只是相应的到达时刻不同，我们考虑能否找出它们之间的联系，从而能够利用已经得出的结果，减少重复运算。

我们来考虑 w(i,T,k)与 w(i,T,k+1)之间的联系：



对于每辆从第 i 个关口出发的巡逻车，设其出发时刻和到达时刻分别为 Stime 和 Ttime，则：

若 Ttime<k 或 Ttime>k+1，则目标车 A、目标车 B 与该巡逻车的相遇情况相同；

若 Ttime=k，则目标车 A 与该巡逻车相遇，对目标车 B 的分析又分为：若 Stime≤T，则目标车 B 不与该巡逻车相遇，否则目标车 B 也与该巡逻车相遇；

若 Ttime=k+1,则目标车 B 与该巡逻车相遇，对目标车 A 的分析又分为：若 Stime≥T，则目标车 A 不与该巡逻车相遇，否则目标车 A 也与该巡逻车相遇；

我们令⊿[k]=w[i,T,k+1]-w[i,T,k]，由上述讨论得：

⊿[k]=G((Ttime=k+1) and (Stime≥T)) –G((Ttime=k) and (Stime≤T)).

其中函数 G(P)表示所有从 i 个关口出发，且满足条件 P 的巡逻车的数目。

这样我们就找到了函数 w 之间的联系。于是，我们在对状态 F(i,T)进行转移时，先对所有从第 i 个关口出发的巡逻车进行一次扫描，在求出 w[i,T,T+300]的同时求出⊿[T+301..T+600]，这一步的时间复杂度为 O(m/n)。在以后的状态转移中，由 w[i,T,k+1]=w[i,T,k]+⊿[k]，仅需 O(1)的时间就可以求出函数值 w，状态转移时间仅为 O(1)。则算法总的时间复杂度为：

$$O\left(150n^2*(\frac{m}{n}+300)\right)=O\left(\frac{m^2n^2}{2}+\frac{m^2n}{2}\right)=O(m^2n^2)$$

虽然，算法时间复杂度的阶并没有降低，但由于 M 的最大值为 300,N 的最大值为 50，所以实际测试中，优化的效果还是十分明显的。

（程序及两种方法的运行结果比较见附件）

本题对动态规划的优化实际上是应用了动态规划本身的思想，在计算递推式的常数项时，引进了函数⊿，利用了过去的计算结果，避免了重复计算，消除了“冗余”，从而提高算法的时间效率。上文邮局问题中函数 w 的计算也是通过预处理减少了重复计算，近来新出现的双重动态规划也是应用这个思想，利用动态规划计算递推式的常数项。可见，这种优化方法是很有普遍性的。

四、结语

本文主要从减少状态总数，减少每个状态转移的状态数和减少状态转移的时间这三个方面讨论了对动态规划时间效率的优化，同时也间接地讨论了对一般算法进行优化的方法。

在优化的过程，我认识到：对算法的优化一方面要深入分析问题的属性，挖掘问题的本质，另一方面要从原有算法的不足之处入手，不断优化、精益求精。

动态规划的算法设计具有很大的灵活性，需要具体模型具体分析。算法设计如此，算法优化也是如此，本文所述只是一些一般性的方法，许多优化技巧还需要选手们在平时的训练比赛中深入挖掘。

动态规划作为一种高效的算法，仍有许多优化的余地。不断提高算法的性能，使其适应于更大的规模，我想这是广大信息学选手共同的愿望，希望大家共同研究探讨动态规划算法的优化，这也是本文创作的初衷。

参考文献

[1] 吴文虎、赵鹏，1993-1996 美国计算机程序设计竞赛试题与解析，清华大学出版社，1999。

[2] 吴文虎、王建德，国际国内青少年信息学（计算机）竞赛试题解析，清华大学出版社，1997。

[3] 傅清祥、王晓东，算法与数据结构，电子工业出版社，1998。

[4] 全国青少年信息学（计算机）奥林匹克分区联赛组织委员会，信息学奥林匹克（季刊），1999.3，2000.2。

附录

[1] 这个式子只是直观描述了动态规划的时间复杂度的决定因素，并不能作为普遍的计算公式。
[2] 四边形不等式是 Donald E. Knuth 从最优二叉搜索树的数据结构中提出的，这里被运用于证明动态规划中决策的单调性。
[3] 采用检索树查找字符串只要从树根出发走到叶结点即可，需要的时间正比于字符串的长度。如果哈希函数确实是随机的，那么哈希函数的值与字符串中的每一个字母都有关系。所以，计算哈希函数值的时间与检索树执行一次运算的时间大致相当。但计算出哈希函数值后还要处理冲突。因此，一般情况下，在进行字符串查找时，检索树比哈希表省时间。

动态规划算法时间效率的优化

福州第三中学　　毛子青

动态规划算法的时间复杂度＝

状态总数＊每个状态转移的状态数＊每次状态转移的时间

一、减少状态总数

1、改进状态表示；（例一）

2、其他方法：选取恰当的规划方向等；

二、减少每个状态转移的状态数

1、根据最优解的性质减少决策量；（例二）

2、其他方法：利用四边形不等式证明决策的单调性等；

三、减少状态转移的时间

1、减少决策时间 （例三）

方法：采用恰当的数据结构；

2、减少计算递推式的时间

方法：进行预处理，利用计算结果等；

例一、 Raucous Rockers 演唱组（ USACO`96 ）

[问题描述]

现有 n 首由 Raucous Rockers 演唱组录制的歌曲，计划从中选择一些歌曲来发行 m 张唱片，每张唱片至多包含 t 分钟的音乐，唱片中的歌曲不能重叠。按下面的标准进行选择：

(1) 这组唱片中的歌曲必须按照它们创作的顺序排序；

(2) 包含歌曲的总数尽可能多。

输入 n， m， t，和 n 首歌曲的长度，它们按照创作顺序排序，没有一首歌超出一张唱片的长度，而且不可能将所有歌曲的放在唱片中。输出所能包含的最多的歌曲数目。

设 n 首歌曲按照创作顺序排序后的长度为 long[1..n]，则动态规划的状态表示描述为：

g[i, j, k]，(0≤i≤n， 0≤j≤m， 0≤k<t)，表示前 i 首歌曲，用 j 张唱片另加 k 分钟来录制，最多可以录制的歌曲数目。

状态转移方程为：

当 k≥long[i]， i≥1 时：

g[i, j, k]=max{g[i-1,j,k-long[i]]+1， g[i-1,j,k]}

当 k<long[i]， i≥1 时：

g[i, j, k]=max{g[i-1,j-1,t-long[i]]+1， g[i-1,j,k]}

规划的边界条件为：

当 0≤j≤m， 0≤k<t 时： g[0,j,k]=0;

问题的最优解为： g[n,m,0] 。

算法的时间复杂度为： O(n*m*t) 。

改进的状态表示描述为：

g[i,j]=(a, b)， $0 \le i \le n$， $0 \le j \le i$， $0 \le a \le m$， $0 \le b \le t$，表示在前 i 首歌曲中选取 j 首录制所需的最少唱片为： a 张唱片另加 b 分钟。

状态转移方程为：

g[i, j]=min{g[i-1,j]， g[i-1,j-1]+long[i]}

其中 (a, b)+long[i]=(a', b') 的计算方法为：

当 b+long[i] ≤t 时： a'=a; b'=b+long[i];

当 b+long[i] > t 时： a'=a+1; b'=long[i];

规划的边界条件：

当 $0 \le i \le n$ 时， g[i,0]=(0,0)

题目所求的最大值是： answer=max{k| g[n, k]≤(m-1,t)}

算法的时间复杂度为： $O(n^2)$ 。

Back

例三、石子合并问题 (NOI`95)

[问题描述]

在一个操场上摆放着一圈 n 堆石子。现要将石子有次序地合并成一堆。规定每次只能选相邻的 2 堆石子合并成新的一堆，并将新的一堆的石子数记为该次合并的得分。

试编程求出将 n 堆石子合并成一堆的最小得分和最大得分以及相应的合并方案。

本例只考虑最大得分。

设各堆的石子数依次为 d[1..n]，则动态规划的状态表示为：

m[i,j]， $1 \le i, j \le n$，表示合并 d[i..j] 所得到的最大得分：

令 $t[i,j]=\sum_{k=i}^{j} d[k]$，则状态转移方程为：

$$m[i,j]=\max_{i \le k \le j-1}\{m[i,k]+m[k+1,j]+t[i,j]\} \qquad i<j$$

规划的边界条件为： m[i,i]=0

令 s[i,j]=k，表示合并的最优断开位置。

算法的时间复杂度为 $O(n^3)$ 。

合并第 i 堆到第 j 堆石子的最优断开位置 s[i,j] 要么等于 i，要么等于 j-1，也就是说最优合并方案只可能是：

{ (i) (i+1… j) }　　或　　{ (i… j-1) (j) }

证明：设合并第 i 堆到第 j 堆石子的最优断开位置 s[i,j]=p，且 i<p<j-1 。

情况 1 、 t[i, p]≤t[p+1,j]

由于 i<p，所以可以设 q=s[i,p] 。于是最优合并方案为：

{ [(i…q) (q+1...p)] (p+1…j) }

它的得分 **F_1=m[i, q]+m[q+1,p]+m[p+1,j]+t[i, j]+t[i, p]**

我们可以构造如下的合并方案：

{ (i…q) [(q+1...p) (p+1…j)] }

它的得分 **F_2=m[i, q]+m[q+1,p]+m[p+1,j]+t[i, j]+t[q+1,j]**

由于 q<p，所以 t[i, p]≤t[p+1,j]<t[q+1,j]

所以 $F_1<F_2$，这与合并第 i 堆到第 j 堆石子的最优断开位置 s[i, j]=p 矛盾

状态转移方程优化为：

$$m[i,j]=\max\{m[i+1,j],\ m[i,j-1]\}+t[i,j] \qquad i<j$$

规划的边界条件是：　$m[i,i]=0$

算法的时间复杂度 $O(n^2)$ 。

例三、 LOSTCITY (NOI`2000)

[问题描述]

现给出一张单词表、特定的语法规则和一篇文章：

文章和单词表中只含 26 个小写英文字母 a…z 。

单词表中的单词只有名词，动词和辅词这三种词性，且相同词性的单词互不相同。单词的个数不超过 1000 ，单词的长度均不超过 20 。

语法规则可简述为：名词短语：任意个辅词前缀接上一个名词；动词短语：任意个辅词前缀接上一个动词；句子：以名词短语开头，名词短语与动词短语相间连接而成。

文章的长度不超过 5k 。且已知文章是由有限个句子组成的，句子只包含有限个单词。

编程将这篇文章划分成最少的句子，在此前提之下，要求划分出的单词数最少。

我们分别用 v,u,a 表示动词，名词和辅词，给出的文章用 L[1..M] 表示，则状态表示描述为：

F(v,i)：表示将 L 的前 i 个字符划分为以动词结尾（当 i<>M 时，可带任意个辅词后缀）的最优分解方案下划分的句子数与单词数；

F(u,i)：表示将 L 的前 i 个字符划分为以名词结尾（当 i<>M 时，可带任意个辅词后缀）的最优分解方案下划分的句子数与单词数。

状态转移方程为：

F(v,i)=min{ F(u,j)+(0,1)，　L(j+1..i) 为动词；

　　F(v,j)+(0,1)，　L(j+1..i) 为辅词，　i<>M；　}

F(u,i)=min{ F(u,j)+(1,1)，L(j+1..i) 为名词；

　　F(v,j)+(0,1)，L(j+1..i) 为名词；

　　F(u,j)+(0,1)，L(j+1..i) 为辅词，　i<>M；　}

边界条件：　F(v,0)=(1,0)；　　F(u,0)=(∞, ∞)；

问题的解为：　min{ F(v,M), F(u,M) }；

采用不同的方法查找字符串的比较：

设单词表的规模为 N （ N 的最大值为 1000 ）

设文章的长度为 M （ M 的最大值为 5000 ）

	顺序查找	二分查找	哈希表	检索树
算法的时间复杂度	O(20*M*N)	O(20*M*$\log_2$N)	O(20*M)	O(M)
最坏情况下的比较次数	10^8	10^6	10^5	$5*10^3$
Input.009	超时	1.27s	0.32s	0.05s
Input.010	超时	1.33s	0.33s	0.05s

（测试环境： Pentium 200 / 32MB ）

Back

基本动态规划问题的扩展

应用动态规划可以有效的解决许多问题，其中有许多问题的数学模型，尤其对一些自从 57 年就开始研究的基本问题所应用的数学模型，都十分精巧。有关这些问题的解法，我们甚至可以视为标准——也就是最优的解法。不过随着问题规模的扩大化，有些模型显出了自身的不足和缺陷。这样，我们就需要进一步优化和改造这些模型。

一. 程序上的优化：

程序上的优化主要依赖问题的特殊性。我们以 $f(X^T)= opt\{f(u^T)\}+ A(X^T), u^T \in Pred\_Set(X^T)$ 这样的递推方程式为例（其中 $A(X^T)$ 为一个关于 X^T 的确定函数，$Pred\_Set(X^T)$ 表示 X^T 的前趋集）。我们设状态变量 X^T 的维数为 t，每个 X^T 与前趋中有 e 维改变，则我们可以通过方程简单的得到一个时间复杂度为 $O(n^{t+e})$ 的算法。

当然，这样的算法并不是最好的算法。为了简化问题，得到一个更好的算法。我们设每个 X^T 所对应的 $g(X^T)=opt\{f(u^T)\}$，则 $f(X^T)=g(X^T)+A(X^T)$，问题就变为求 $g(X^T)$ 的值。下面分两个方面讨论这个问题：

1. $Pred\_Set(X^T)$ 为连续集：

在这样的情况下，我们可以用 $g(X^T)= opt\{g(Pred(X^T)), f(Pred(X^T))\}$ 这样一个方程式来求出 $g(X^T)$ 的值，并再用 $g(X^T)$ 的值求出 $f(X^T)$ 的值。这样，虽然我们相当于对 $g(X^T)$ 和 $f(X^T)$ 分别作了一次动态规划，但由于两个规划是同时进行的，时间复杂度却降为了 $O(n^t)$。由于我们在实际使用中的前趋即通常都是连续的，故这个方法有很多应用。例如 IOI'99 的《小花店》一题就可以用该方法把表面上的时间复杂度 $O(FV^2)$ 降为 $O(FV)$。

2. $Pred\_Set(X^T)$ 为与 X^T 有关的集合：

这样的问题比较复杂，我们以最长不下降子序列问题为例。规划方程为：$f(i)=max\{f(j)\}+1, d[i] \geq d[j]; i>j$。通常认为，这个问题的最低可行时间复杂度为 $O(n^2)$。不过，这个问题只多了一个 $d[i] \geq d[j]$ 的限制，是不是也可以优化呢？我们注意到 $max\{f(j)\}$ 的部分，它的时间复杂度为 $O(n)$。但对于这样的式子，我们通常都可以用一个优先队列来使这个 max 运算的时间复杂度降至 $O(\log n)$。对于该问题，我们也可以用这样的方法。在计算 $d[i]$ 时，我们要先有一个平衡排序二叉树（例如红黑树）对 $d[1]\sim d[i-1]$ 进行排序。并且我们在树的每一个节点新增一个 MAX 域记录它的左子树中的函数 f 的最大值。这样，我们在计算 $f(i)$ 时，只需用 $O(\log n)$ 时间找出不比 $d[i]$ 大的最大数所对应的节点，并用 $O(1)$ 的时间访问它的 MAX 域就可以得出 $f(i)$ 的值。并且，插入操作和更新 MAX 域的操作也都只用 $O(\log n)$ 的时间（我们不需要删除操作），故总时间复杂度为 $O(n \log n)$。实际运行时这样的程序也是十分快的，$n=10000$ 时用不到 1 秒就可以得出结果，而原来的程序需要 30 秒。

从以上的讨论可以看出，再从程序设计上对问题优化时，要尽量减少问题的约束，尽可能的化为情况 1。若不可以变为情况 1，那么就要仔细考虑数据上的联系，设计好的数据结构来解决问题。

二. 方程上的优化：

对于方程上的优化，其主要的方法就是通过某些数学结论对方程进行优化，避免不必要的运算。对于某一些特殊的问题，我们可以使用数学分析的方法对写出的方程求最值，这样甚至不用状态之间的递推计算就可以解决问题。不过用该方法解决的问题数量是在有限，并且这个方法也十分复杂。不过，却的确有相当数量的比较一般的问题，在应用某些数学结论后，可以提高程序的效率。

一个比较典型的例子是最优排序二叉树问题（CTSC96）。它的规划方程如下：

$$C[i,j]=w(i,j)+\min_{i<k\le j}\{C[i,k-1]+C[k,j]\}\mid 1\le i<j\le n$$

我们可以从这个规划方程上简单的得到一个时间复杂度为 $O(n^3)$的算法。但是否会有更有效的算法呢？我们考虑一下 $w(i,j)$的性质。它表示的是结点 i 到结点 j 的频率之和。很明显，若有$[i,j]\subseteq[i',j']$，则有 $w[i,j]\le w[i',j']$，这样可知 $C[i,j]$具有凸性[1]。为了表示方便，我们记 $C_k(i,j)=w(i,j)+C[i,k-1]+C[k,j]$，并用 $K_{i,j}$表示取到最优值 $C[i,j]$时的 $C_k(i,j)$的 k 值。我们令 $k=K_{i,j-1}$，并取 $i<k'<k$。由于 $k'<k\le j-1<j$，故有：

$$C[k',j-1]+C[k,j]\le C[k',j]+C[k,j-1]$$

在等式两侧同时加上 $w(i,j-1)+w(i,j)+C[i,.\,k-1]+C[i,k'-1]$，可得：

$$C_{k'}(i,j-1)+C_k(i,j)\le C_{k'}(i,j)+C_k(i,j-1)$$

由 k 的定义可知 $C_k(i,j-1)\le C_{k'}(i,j-1)$，故 $C_k(i,j)\le C_{k'}(i,j)$，所以 $k'\ne K_{i,j}$，故 $K_{i,j}\ge K_{i,j-1}$。同理，我们可得 $K_{i,j}\le K_{i+1,j}$，即 $K_{i,j-1}\le K_{i,j}\le K_{i+1,j}$。这样，我们就可以按对角线来划分阶段（就是按照 $j-i$ 划分阶段）来求 $K_{i,j}$。求 $K_{i,j}$的时间复杂度为 $O(K_{i+1,j}-K_{i,j-1}+1)$，故第 d 阶段（即计算 $K_{1,1+d}$~$K_{n-d,n}$）共需时 $O(K_{n-d+1,n}-K_{1,d}+n-d)\le O(n)$。有共有 n 个阶段，故总时间复杂度为 $O(n^2)$。

虽然这道题由于空间上的限制给这个算法的实际应用造成了困难，不过这种方法却给我们以启示。

我们在考虑 IOI2000 的 POST 问题。这一题的数学模型不是讨论的重点，我们先不加讨论，直接给出规划方程 $D_{i,j}=\min_{i-1\le k<j}\{D_{i-1,k}+w(k,j)\}$。从规划方程直接得出的算法的时间复杂度为 $O(n^3)$。从这个规划方程可以看出，它的每一阶段都只与上一阶段有关。故我们可以把方程变得简单些，变为对如下的方程执行 n 次：

$$E[j]=\min_{j<i}\{D[i]+w(i,j)\}$$

在递推时，阶段之间时没有优化的余地的，故优化的重点就在于这个方程的优化上。我们用 $B[i,j]$表示 $D[i]+w(i,j)$，而原算法就是求出 B 并对每一列求最小值。

事实上，这一题的 w 有其特殊的性质：

对于 $a\le b\le c\le d$，我们有 $w(a,c)+w(b,d)\le w(a,d)+w(b,c)$。

这一性质对解题是应该有所帮助的。仿照上例，在两侧加上 $D[a]+D[b]$，可得 $B[a,c]+B[b,d]\le B[a,d]+B[b,c]$。

也就是说，若 $B[a,c]\ge B[b,c]$，则有 $B[a,d]\ge B[b,d]$。于是我们在确定了 $B[a,c]$与 $B[b,c]$的大小关系之后，就可以决定是不是需要比较 $B[a,d]$与 $B[b,d]$的大小。

更进一步的，我们只要找出满足 $B[a,h]\ge B[b,h]$的最小的 h，就可以免去 h 之后对第 a 列的计算。而这样的 h，我们可以用二分查找法在 $O(\log n)$时间内找到（若 w 更特殊一些，例如说是确定的函数，我们甚至可以在 $O(1)$的时间找到）。并且对于每一行来说，都只需要执行一次二分查找。在求出所有的 h 之后，只需用 $O(n)$的时间对每列的第 h 行求值就可以了。这样得出的总时间为 $O(n)+O(n\log n)=O(n\log n)$。至于程序设计上的问题，虽然并不复杂，但不是 15 分钟所可以解决的，也不是重点，略过不谈。[2]不过由于该题目可以用滚动数组的技巧解决空间的问题，故在大数据量时该算法有优异的表现。

从上面的叙述可以看出，对于方程的优化主要取决于权函数 w 的性质。其中应用最多的就是 $w(a,c)+w(b,d)\le w(a,d)+w(b,c)$这个不等式。实际上，这个式子被称作函数的凸性判定不等式。在实际问题中，权函数通常都会满足这个不等式或这是它的逆不等式。故这样的优化应用是比较广泛的。还有许多特殊的不等式，若可以在程序中应用，都可以提高程序的效率。

三. 从低维向高维的转化：

在问题扩大规模时，有一种方式就是扩大问题的维数。这时，规划时决策变量的维数也要增加。这样，存储的空间也要随着成指数级增加，导致无法存储下所有的状态，这就是动态规划的维数灾难问题。如果我们还要在这种情况下使用动态规划，那么就要使用极其复杂的数学分析方法。对于我们来说，使用这种方法显然是不现实的。这时，我们就需要改造动态规划的模型。通常我们都可以把这时的动态规划模型变为网络流模型。

对于模型的转化方法，我们有一些一般的规律。若状态转移方程只与另一个状态有关，我们可以肯定得到一个最小费用最大流的模型[3]。这个模型必然有其规律的地方，甚至用对偶算法在对网络流的求解时也还要用到动态规划的方法。不过这不是重点，我们关心的只是动态规划问题如何转化。例如说 IOI'97《火星探测器》一题。这一题的一维模型是可以用动态规划来解决的（这里的维数概念是指探测器的数目）。在维数增加时，我们就可以用该方法来用网络流的方法解决问题。

除此之外，还有许多问题可以用该方法解决。例如最长区间覆盖问题，在维数增加时也同样可以用该方法解决。更进一步来说，甚至图论的最短路问题也可以做同样的转化来求出特殊的最短路。不过一般来说转化后流量最大为 1，有许多特殊的性质也没有得到应用，并且些复杂的动态规划问题还无法转化为网络流问题（例如说最优二叉树问题），故标准的网络流算法显然有些浪费，它的解决还需要进一步的研究。

参考文献：

[EGG88] David Eppstein, Zvi Galil and Raffaele Giancarlo, Speeding up Dynamic Programming

[GP90] Zvi Galil and Kunsoo Park, Dynamic Programming with Convexity, Concavity, and Sparsity

[附录]

[1] *C[i, j]*的凸性是指对于任意的 $a \le b \le c \le d$，都有 $C[a, c]+ C[b, d] \le C[a, d]+ C[b, c]$。它的证明如下：

我们设 $k=K_{b,c}$，则有 $C[a, c]+ C[b, d] \le C[a, k-1]+ C[k, c]+ C[b, k-1]+ C[k, d]+ w(a, c) + w(b, d)= C[a, k-1]+ C[k, d]+ w(a, d)+ C[b, k-1]+ C[k, c]+ w(b, c)= C[a, d]+ C[b, c]$。得证。

[2] 有关这个问题的伪代码如下：

```
begin
 E[1]←D[1];
 Queue.Add(K:1, H:n);
 for j←2 to n do
 begin
  if B(j-1, j)≤B(Queue.K[head], j) then
  begin
   E[j]←B(j-1, j);
   Queue.Empty;
   Queue.Add(K:j-1, H:n+1);
  end else
  begin
   E[j]←B(Queue.K[head], j);
   while B(j-1, Queue.H[tail-1])≤B(Queue.K[tail], Queue.H[tail-1]) do Queue.Delete(tail);
   Queue.H[tail]←h(Queue.K[tail], j-1);
   if h_OK then Queue.Add(K:j-1, H:n+1);
  end;
  if Queue.H[head]=j+1 then Queue.SkipHead;
 end;
end;
```

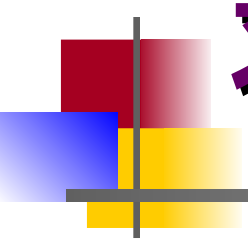
其中的队列 *Queue* 可以称作备选队列，其中的队列头为第 *j* 行的最小值，并假设 *Queue.H[0]=j*。其中的 *h(a, b)*函数是用二分查找法查找 $B(a, h) \ge B(b, h)$的最小的 *h* 值，*h_OK* 为查找成功与否的标志。在备选队列 *Queue* 中的数据$(K:i_r, H:h_r)$的意义是：当行数在区间$[h_{r-1}, h_r-1]$的范围内时，第 i_r 列为最小值。

[3] 我们知道，动态规划实际是求无向图的最短路的一种方法，故我们可以把动态规划中的每一个状态看成一个点，并将状态的转移过程变为一个图。在转化为最小费用最大流时，我们把这一个点拆成两个点，一个出点和一个入点，所有指向原来这个点的边都与入点相连，且所有由原来这个点发出的边现都以出点为起点。原来的边的容量设为正无穷，边权值一般不变。新增的入点与出点之间连一条边，它的权值为点权值，容量为每一点可以经过的次数（一般为一）。并且建立一个超级源和一个超级汇，并与可能的入点和出点连边。若有必要，超级源（或汇）也要拆成两个点，并且两个点之间的边的容量为最大的可能容量，边权值为 0。这样，用最小费用流的方法得出的解就是该问题多维情况下的接。

[4] 源程序：

最长不下降子序列的一般程序	最长不下降子序列的改进程序	最长不下降子序列的数据生成程序	最优排序二叉数问题的一般程序

 Lennor.pas	 Lentree.pas	 Lengern.pas	 Btreenor.pas
最优排序二叉树问题的改进程序	最优排序二叉数问题的数据生成程序	邮局问题的一般程序	邮局问题的大数据程序
 Btreespe.pas	 Tgern.pas	 post.pas	 Pbig.pas
邮局问题的改进程序	邮局问题的测试数据生成程序	邮局问题的大数据生成程序	
 Pspe.pas	 pgern.pas	 Pbgern.pas	



求 N! 的高精度算法

本文中的算法主要针对 Pascal 语言

这篇文章的内容

- 你了解高精度吗？
- 你曾经使用过哪些数据结构？
- 你仔细思考过如何优化算法吗？

在这里，你将看到怎样成倍提速求 N! 的高精度算法

关于高精度

- Pascal中的标准整数类型
- 高精度算法的基本思想

Pascal 中的标准整数类型

数据类型	值域
Shortint	-128 ~ 127
Byte	0 ~ 255
Integer	-32768 ~ 32767
Word	0 ~ 65535
Longint	-2147483648 ~ 2147483647
Comp	-9.2e18 ~ 9.2e18

Comp 虽然属于实型，实际上是一个 64 位的整数

高精度算法的基本思想

- Pascal 中的标准整数类型最多只能处理在 -2^{63} ~ 2^{63} 之间的整数。如果要支持更大的整数运算，就需要使用高精度
- 高精度算法的基本思想，就是将无法直接处理的大整数，分割成若干可以直接处理的***小整数段***，把对大整数的处理转化为对这些***小整数段***的处理

数据结构的选择

- 每个小整数段保留尽量多的位
- 使用Comp类型
- 采用二进制表示法

每个小整数段保留尽量多的位

- 一个例子：计算两个 15 位数的和
  - 方法一
    - 分为 15 个小整数段，每段都是 1 位数，需要 15 次 1 位数加法
  - 方法二
    - 分为 5 个小整数段，每段都是 3 位数，需要 5 次 3 位数加法
  - 方法三
    - Comp 类型可以直接处理 15 位的整数，故 1 次加法就可以了
  - 比较
    - 用 Integer 计算 1 位数的加法和 3 位数的加法是一样快的
    - 故方法二比方法一效率高
    - 虽然对 Comp 的操作要比 Integer 慢，但加法次数却大大减少
    - 实践证明，方法三比方法二更快

使用 Comp 类型

- 高精度运算中，每个小整数段可以用 Comp 类型表示
- Comp 有效数位为 19 ~ 20 位
- 求两个高精度数的和，每个整数段可以保留 17 位
- 求高精度数与不超过 m 位整数的积，每个整数段可以保留 18–m 位
- 求两个高精度数的积，每个整数段可以保留 9 位
- 如果每个小整数段保留 k 位十进制数，实际上可以认为其只保存了 1 位 10^k 进制数，简称为***高进制数***，称 1 位高进制数为***单精度数***

采用二进制表示法

- 采用二进制表示，运算过程中时空效率都会有所提高，但题目一般需要以十进制输出结果，所以还要一个很耗时的进制转换过程。因此这种方法竞赛中一般不采用，也不在本文讨论之列

算法的优化

- 高精度乘法的复杂度分析
- 连乘的复杂度分析
- 设置缓存
- 分解质因数求阶乘
- 二分法求乘幂
- 分解质因数后的调整

高精度乘法的复杂度分析

- 计算 n 位高进制数与 m 位高进制数的积
  - 需要 n*m 次乘法
  - 积可能是 n+m–1 或 n+m 位高进制数

连乘的复杂度分析（ 1 ）

- 一个例子：计算 5*6*7*8
  - 方法一：顺序连乘
    - 5*6=30， 1*1=1 次乘法
    - 30*7=210， 2*1=2 次乘法
    - 210*8=1680， 3*1=3 次乘法

    共 6 次乘法
  - 方法二：非顺序连乘
    - 5*6=30， 1*1=1 次乘法
    - 7*8=56， 1*1= 1 次乘法
    - 30*56=1680， 2*2=4 次乘法

    共 6 次乘法

特点： n 位数 *m 位数 =n+m 位数

连乘的复杂度分析（２）

- 若“ n 位数 *m 位数 =n+m 位数”，则 n 个单精度数，无论以何种顺序相乘，乘法次数一定为 n(n-1)/2 次
  - 证明：
    - 设 F(n) 表示乘法次数，则 F(1)=0 ，满足题设
    - 设 k<n 时， F(k)=k(k-1)/2 ，现在计算 F(n)
    - 设最后一次乘法计算为“ k 位数 *(n-k) 位数”，则
    - F(n)=F(k)+F(n-k)+k (n-k)=n(n-1)/2 （与 k 的选择无关）

设置缓存（１）

- 一个例子：计算 9*8*3*2
  - 方法一：顺序连乘
    - 9*8=72， 1*1=1 次乘法
    - 72*3=216， 2*1=2 次乘法
    - 216*2=432， 3*1=3 次乘法

    共 6 次乘法
  - 方法二：非顺序连乘
    - 9*8=72， 1*1=1 次乘法
    - 3*2=6， 1*1=1 次乘法
    - 72*6=432， 2*1=2 次乘法

    共 4 次乘法

特点： n 位数 *m 位数可能是 n+m-1 位数

设置缓存（２）

- 考虑 k+t 个单精度数相乘 $a_1*a_2*\ldots*a_k*\mathbf{a_{k+1}}*\ldots*\mathbf{a_{k+t}}$
  - 设 $a_1*a_2*\ldots*a_k$ 结果为 m 位高进制数（假设已经算出）
  - $\mathbf{a_{k+1}}*\ldots*\mathbf{a_{k+t}}$ 结果为 1 位高进制数
  - 若顺序相乘，需要 t 次“ m 位数 *1 位数”，共 mt 次乘法
  - 可以先计算 $\mathbf{a_{k+1}}*\ldots*\mathbf{a_{k+t}}$，再一起乘，只需要 m+t 次乘法

在设置了缓存的前提下，计算 m 个单精度数的积，如果结果为 n 位数，则乘法次数约为 n(n–1)/2 次，与 m 关系不大

- 设 $S=a_1a_2\ldots a_m$， S 是 n 位高进制数
- 可以把乘法的过程近似看做，先将这 m 个数分为 n 组，每组的积仍然是一个单精度数，最后计算后面这 n 个数的积。时间主要集中在求最后 n 个数的积上，这时基本上满足“ n 位数 *m 位数 =n+ m 位数”，故乘法次数可近似的看做 n(n-1)/2 次

设置缓存（３）

- 缓存的大小
  - 设所选标准数据类型最大可以直接处理 t 位十进制数
  - 设缓存为 k 位十进制数，每个小整数段保存 t–k 位十进制数
  - 设最后结果为 n 位十进制数，则乘法次数约为
  - $k/(n–k)\sum_{(i=1..n/k)}i=(n+k)n/(2k(t–k))$，其中 k 远小于 n
  - 要乘法次数最少，只需 k (t–k) 最大，这时 k=t/2
  - 因此，缓存的大小与每个小整数段大小一样时，效率最高
  - 故在一般的连乘运算中，可以用 Comp 作为基本整数类型，每个小整数段为 9 位十进制数，缓存也是 9 位十进制数

分解质因数求阶乘

- 例： $10!=2^8*3^4*5^2*7$
  - n! 分解质因数的复杂度远小于 nlogn ，可以忽略不计
  - 与普通算法相比，分解质因数后，虽然因子个数 m 变多了，但结果的位数 n 没有变，只要使用了缓存，乘法次数还是约为 n(n-1)/2 次
  - 因此，分解质因数不会变慢（这也可以通过实践来说明）
  - 分解质因数之后，出现了大量求乘幂的运算，我们可以优化求乘幂的算法。这样，分解质因数的好处就体现出来了

二分法求乘幂

- 二分法求乘幂，即：
  - $a^{2n+1}=a^{2n}*a$
  - $a^{2n}=(a^n)^2$
  - 其中， a 是单精度数
- 复杂度分析
  - 假定 n 位数与 m 位数的积是 n+m 位数
  - 设用二分法计算 a^n 需要 F(n) 次乘法
  - $F(2n)=F(n)+n^2$， $F(1)=0$
  - 设 $n=2^k$，则有 $F(n)=F(2^k)=\sum_{(i=0..k-1)}4^i=(4^k-1)/3=(n^2-1)/3$
- 与连乘的比较
  - 用连乘需要 n(n-1)/2 次乘法，二分法需要 $(n^2-1)/3$
  - 连乘比二分法耗时仅多 50%
  - 采用二分法，复杂度没有从 n^2 降到 nlogn

二分法求乘幂之优化平方算法

- 怎样优化
  - $(a+b)^2=a^2+2ab+b^2$
  - 例： $12345^2=\underline{\mathbf{123}}^2*10000+\underline{\mathbf{45}}^2+2*\underline{\mathbf{123*45}}*100$
  - 把一个 n 位数分为一个 t 位数和一个 n-t 位数，再求平方
- 怎样分
  - 设求 n 位数的平方需要 F(n) 次乘法
  - F(n)=F(t)+F(n-t)+t(n-t)， F(1)=1
  - 用数学归纳法，可证明 F(n) 恒等于 n(n+1)/2
  - 所以，无论怎样分，效率都是一样
  - 将 n 位数分为一个 1 位数和 n–1 位数，这样处理比较方便

二分法求乘幂之复杂度分析

- 复杂度分析
  - 前面已经求出 $F(n)=n(n+1)/2$ ，下面换一个角度来处理
  - $S^2=(\sum_{(0\le i<n)}a_i10^i)^2=\sum_{(0\le i<n)}a_i^2 10^{2i}+2\sum_{(0\le i<j<n)}a_i a_j 10^{i+j}$
  - 一共做了 $n+C(n,2)=n(n+1)/2$ 次乘法运算
  - 普通算法需要 n^2 次乘法，比改进后的慢 1 倍
- 改进求乘幂的算法
  - 如果在用改进后的方法求平方，则用二分法求乘幂，需要 $(n+4)(n-1)/6$ 次乘法，约是连乘算法 $n(n-1)/2$ 的三分之一

分解质因数后的调整（１）

- 为什么要调整
  - 计算 $S=2^{11}3^{10}$，可以先算 2^{11}，再算 3^{10}，最后求它们的积
  - 也可以根据 $S=2^{11}3^{10}=6^{10}*2$，先算 6^{10}，再乘以 2 即可
  - 两种算法的效率是不同的

分解质因数后的调整（２）

- 什么时候调整
  - 计算 $S=a^{x+k}b^x=(ab)^x a^k$
  - 当 $k<x\log_a b$ 时，采用 $(ab)^x a^k$ 比较好，否则采用 $a^{x+k}b^x$ 更快
  - 证明：
    - 可以先计算两种算法的乘法次数，再解不等式，就可以得到结论
    - 也可以换一个角度来分析。其实，两种算法主要差别在最后一步求积上。由于两种方法，积的位数都是一样的，所以两个因数的差越大，乘法次数就越小
    - ∴ 当 $a^x b^x - a^k > a^{x+k} - b^x$ 时，选用 $(ab)^x a^k$，反之，则采用 $a^{x+k}b^x$。
    - $\therefore a^x b^x - a^k > a^{x+k} - b^x$
    - $\therefore (b^x - a^k)(a^x+1)>0$
    - $\therefore b^x > a^k$
    - 这时 $k<x\log_a b$

总结

- 内容小结
  - 用 Comp 作为每个小整数段的基本整数类型
  - 采用二进制优化算法
  - 高精度连乘时缓存和缓存的设置
  - 改进的求平方算法
  - 二分法求乘幂
  - 分解质因数法求阶乘以及分解质因数后的调整
- 应用
  - 高精度求乘幂（平方）
  - 高精度求连乘（阶乘）
  - 高精度求排列组合

结束语

求 N! 的高精度算法本身并不难，但我们仍然可以从多种角度对它进行优化。

其实，很多经典算法都有优化的余地。我们自己编写的一些程序也不例外。只要用心去优化，说不准你就想出更好的算法来了。

也许你认为本文中的优化毫无价值。确实是这样，竞赛中对高精度的要求很低，根本不需要优化。而我以高精度算法为例，不过想谈谈**如何**优化一个算法。我想说明的只有一点：**算法是可以优化的。**