第3章 算法基本工具和优化技巧

算法设计的基础工作是把人脑思维出的解决问题的方法、步骤，规范化地描述成“机械化的操作”。这就好像自动化生产，其实质就是把人类生产中，较规范的大量重复工作交给了机器去完成。所以有人称计算机带来了“机械”思维的时代。

从已学过的计算机知识中可以了解到，计算机或者说程序设计语言为算法提供的“机械化的操作”其实是很少的。主要有计算、输入和输出操作，流程控制操作（即选择、循环和递归），以及提供了这些操作对象的不同存储模式：变量、数组、结构体（记录）和文件。而软件功能之丰富，使用之方便，却是越来越让人感叹，这都是人类设计出来的算法的功劳。相信读者能够体会到学习算法设计的必要性。

本章就是要讲解怎么样充分利用这些基本的“机械化的操作”设计高质量的算法，在程序设计与算法设计之间起承上启下的作用。

3.1 循环与递归

事实上，在一般情况下只有处理大量的数据才借助于计算机，所以算法设计中很重要的工作就是把对数据的处理归结成较规范的可重复的“机械化的操作”交给计算机去完成。即将重复处理大量数据的步骤抽象成“循环”或“递归”的模式，设计出可以针对不同规模解决问题的算法。

不同于机器生产产品的“机械化的重复操作”，计算机进行数据处理不可能是完全相同操作的重复。所以必须要设计出表现形式不变，但能实现动态处理数据的“机械化的重复操作”。也就是说，在重复操作中，“循环条件”“循环体”都必须是“不变式”，而数据处理对象却是变化的，算法是在渐进地完成处理数据的操作。

循环模式算法设计中，一个重要的工作就是从已建立好的数学模型中，构造出“不变式”的“循环条件”“循环体”。“不变式”主要是依靠变量或数组元素表示的，因为变量名或数组元素是“不变”的，而变量或数组元素中的数据是不断变化的，从而数据处理是动态的、渐进的。

本节通过实例介绍循环不变式和递归不变式的构造过程，简单地说明循环、递归设计的基本方法及应该考虑的因素。

【提示】 请读者回忆高级语言程序设计和数据结构课程中，设计“循环条件不变式”“循环体不变式”的方法和技巧。


循环设计

3.1.1 循环设计要点

循环设计要点很多,下面就以下三点进行讨论。

1. 设计中要注意算法的效率

累加、累乘是学习程序设计语言中接触最多的程序,它就是通过数学模型 $S_n=S_{n-1}+A_n$,$T_n=T_{n-1}\times A_n$,构造出“循环不变”的累加式 $S=S+A$ 和累乘式 $T=T\times A$。下面看一个累加、累乘算法的设计过程。

【例 1】 求 $1/1!-1/3!+1/5!-1/7!+\cdots+(-1)^{n+1}/(2n-1)!$。

问题分析:此问题中既有累加又有累乘,准确地说累加的对象是累乘的结果。

数学模型 1:$S_n=S_{n-1}+(-1)^{n+1}/(2n-1)!$。

【提示】 思考“循环条件”和“循环体”不变式如何表示?

算法设计 1:多数初学者会直接利用题目中累加项通式,构造出循环体不变式为 $S=S+(-1)^{n+1}/(2n-1)!$,再通过二重循环计算 $(-1)^{n+1}/(2n-1)!$ 来完成算法。

算法 1 如下:

```
main( )
{int i,n,j,sign = 1;
float s,t = 1;
  input(n);
  s = 1;
for(i = 2; i <= n; i = i + 1)
  {t = 1;                                  // 求阶乘
  for(j = 1; j <= 2 * i - 1; j = j + 1)
      t = t * j;
  sign = 1;                                // 求(-1)^(n+1)
  for(j = 1; j <= i + 1; j = j + 1)
      sign = sign * (-1);
  s = s + sign/t;
  }
 print("Sum = ",s);
}
```

算法分析 1:以上算法是完全正确的,但算法的效率却太低是 $O(n^2)$。其原因是,当前一次循环已求出 7!,当这次要想求 9! 时,没必要再从 1 去累乘到 9,只需要充分利用前一次的结果,用 7!×8×9 即可得到 9!,模型为 $A_n=A_{n-1}\times 1/((2\times n-2)\times(2\times n-1))$。另外运算 sign=sign×(−1);总共也要进行 $n\times(n-1)/2$ 次乘法,这也是没有必要的。下面就进行改进。

数学模型 2:$S_n=S_{n-1}+(-1)^{n+1}A_n$;$A_n=A_{n-1}\times 1/((2\times n-2)\times(2\times n-1))$。

算法设计 2:利用以上数学模型容易构造累加、累乘不变式,对 $(-1)^{n+1}$ 可以用一个变量 sign 记录其值,每循环一次执行“sign=−sign;”就可以模拟符号的变化过程。这样,只需要一重循环就能解决问题。

算法 2 如下：

```
main( )
{int i,n,sign;
 float s,t = 1;
 input(n);
  s = 1;
  sign = 1;
 for(i = 2; i <= n; i = i + 1)
   {sign = - sign;
   t = t * (2 * i - 2) * (2 * i - 1)};
   s = s + sign/t; }
 print("Sum = ",s);
}
```

或

```
for(i = 1; i <= n - 1; i = i + 1)
{ sign = - sign;
t = t * 2 * i * (2 * i + 1)};
s = s + sign/t;
```

算法说明：构造循环不变式时，一定要注意循环变量的意义，如当 i 不是项数序号时（右边的循环中）有关 t 的累乘式与 i 是项数序号时就不能相同。

算法分析 2：这个算法的时间复杂度为 $O(n)$。

由此例，构造循环不变式时，一定要考虑算法的效率。

2. “自顶向下”的设计方法

对于简单的算法，可以像例 1 一样直接进行算法设计，对于比较难一些的算法，则可以用“自顶向下”的设计方法，特别是有嵌套循环的情况。

自顶向下的方法是从全局走向局部、从概略走向详尽的设计方法。自顶向下是系统分解和细化的过程，也是算法设计的方法。

【例 2】 一个数如果恰好等于它的因子之和（包括 1，但不包括这个数本身），这个数就称为“完数”。

例如，28 的因子为 1，2，4，7，14，而 28＝1＋2＋4＋7＋14。因此 28 是“完数”。编写算法找出 1000 之内的所有完数，并按下面格式输出其因子：28 it's factors are 1，2，4，7，14。

问题分析：这个问题中不是要质因数，所以找到因数后就无须将其从数据中“除掉”。每个因数只记一次，如 8 的因数为 1，2，4 而不是 1，2，2，2，4（注：本题限定因数不包括这个数本身）。

算法设计：“自顶向下”的算法设计方法，就是先概要地设计算法的第一层（即顶层），然后步步深入，逐层细分，逐步求精，直到整个问题可用程序设计语言明确地描述出来为止。

自顶向下设计的步骤：首先对问题进行仔细分析，写出程序运行的主要过程和任务；然后从大的功能方面把一个问题的解决过程分为几个子问题，每个子问题形成一个模块。这样，可以使设计过程中的每一时刻都只需要考虑很少的问题。

本题的设计过程如下。

1）顶层算法

```
for(i = 2; i <= n; i = i + 1)
  {判断 i 是否"完数";
   是"完数"则按格式输出; }
```

2）判断 i 是否“完数”的算法

```
for(j=2; j<i; j=j+1)
   找 i 的因子,并累加;
如果累加值等于 i,i 是"完数"则输出;
```

3）进一步细化——判断 i 是否“完数”算法

```
s=1
for(j=2; j<i; j=j+1)
    if (i mod j=0) (j 是 i 的因数)  s=s+j;
if  (s=i)     i 是"完数";
```

4）考虑输出格式——判断 i 是否“完数”算法

考虑到要按格式输出结果，应该开辟数组存储数据 i 的所有因子，并记录其因子的个数，因此算法细化如下：

```
定义数组 a,变量 s=1,k=0;
for(j=2; j<i; j=j+1)
   if (i mod j=0) (j 是 i 的因素)
      {s=s+j; a[k]=j; k=k+1; }
if    (s=i)
  {按格式输出结果}
```

综合以上逐层设计结果，得到以下算法：

```
main( )
{int i,k,j,s,a[20];
for(i=1; i<=1000; i=i+1)
  {s=1;
   k=0;
  for(j=2; j<i; j=j+1)
   if (i mod j)=0)
     {s=s+j;
      a[k]=j;
      k=k+1; }
  if(i=s)
      {print(s,"it's  factors  are: ",1);
       for(j=0;j<k;j=j+1)
          print(",",a[j]);
      }
  }
}
```

由例题可以看出，自顶向下设计的特点：先整体后局部，先抽象后具体。

【提示】 “自顶向下”通俗地说就是“由粗到细”，可以使设计更简便，思路更清楚，请仔细体会。

下面再看一个例子。

【例 3】 求一个矩阵的鞍点，即在行上最小而在列上最大的点。

算法设计：针对 $n\times n$ 矩阵进行设计，操作逐行进行，行列下标起始为 0。“自顶向下”的设计如下。

1）顶层算法

```
for(i = 0; i < n; i = i + 1)
  {找第 i 行上最小的元素 t 及所在列 minj;
   检验 t 是否为第 minj 列的最大值,是,则输出这个鞍点; }
```

2）找第 i 行上最小的元素 t 及所在列 minj

```
t = a[i][0]; minj = 0;
for(j = 1; j < n; j = j + 1)
  if(a[i][j]< t)
      {t = a[i][j];
       minj = j; }
```

3）检验 t 是否为第 minj 列的最大值,是,则输出这个鞍点

```
for(k = 0; k < n; k = k + 1)
   if(a[k][minj]> t) break;
if(k < n)     continue;
print("the result is a[",i,"][",minj,"] = ",t);
```

综合以上设计结果,得到以下算法：

```
  readmtr(int a[][10],int n)
      {int  i,j;
        print("input n * n matric: ");
        for(i = 0; i < n; i = i + 1)
         for(j = 0; j < n; j = j + 1)
           input(a[i][j]);
     }
    printmtr(int a[][10],int n)
     {int i,j
      for(i = 0; i < n; i = i + 1)
       {for(j = 0; j < n; j = j + 1)
           print(a[i][j]);
       print("换行符")
       }
     }
main( )
       {int a[10][10];
       int i,j,k,minj,t,n = 10,kz = 0;
       readmtr(a,n);
       printmtr(a,n);
       for(i = 0; i < n; i = i + 1)
         {t = a[i][0];
          minj = 0;
          for(j = 1; j < n; j = j + 1)
             if(a[i][j]< t)
               {t = a[i][j];
               minj = j; }
          for(k = 0; k < n; k = k + 1)
             if(a[k][minj]> t)  break;
          if(k < n) continue;
          print("the result is a[",i,"][",minj,"] = ",t);
```

```
    kz = 1;
    break;
    }
if(kz = 0) print("Non solution!");
}
```

算法说明：

(1) 算法中 minj 代表当前行中最小值的列下标，循环变量 i，j 分别代表行、列下标。循环变量 k 也代表行下标，在循环“for(k=1; k<=n; k=k+1)”中只针对 minj 列进行比较。

(2) 考虑到会有无解的情况，设置标志量 kz，kz=0 代表无解，找到一个解后，kz 被赋值为 1，就不再继续找鞍点的工作。请读者考虑是否有多解的可能性吗？若有，请改写算法，找出矩阵中所有的鞍点。

【提示】 程序语言课程讲授过二维数组逐行逐列操作，若已认真理解并掌握，这里就不会感到困难了。应该遵循学习不能拈轻怕重，要循序渐进。

3. 由具体到抽象设计循环结构

对于不太熟悉的问题，其数学模型或“机械化操作步骤”不易抽象，下面看一个由具体到抽象设计循环细节的例题。

【例 4】 编写算法：打印具有下面规律的图形。

```
 1
 5  2
 8  6  3
10  9  7  4
```

问题分析：无论从题意理解，还是从算法的通用性来考虑，算法设计不能只针对图中的 4×4 二维数组进行。下面以任意阶的二维数组 $n \times n$ 讨论。为分析方便，数组的起始下标定为 1。

存储设计：对这样的二维表，一般用二维数组存储。

算法设计：对二维表的操作一般是按行或列进行的，但此图形中数据的排列规律却是按对角线排列的。因此设计的要点就是在二者间找关系。下面根据数据排列的特点，将对角线称为“层”，循环按数据特点逐层进行，层内又有多个数据，算法需要二重循环。

【提示】 当然也可以以行、列作为循环变量，找出数据排列与循环变量的关系，请读者尝试。

容易发现图形中自然数在矩阵中排列的规律，题目中 1,2,3,4 所在位置称为第 1 层(主对角线)，例图中 5,6,7 所在位置称为第二层……。一般地，第一层有 n 个元素，第二层有 $n-1$ 个元素……。

基于以上数据变化规律，以层号作为外层循环，循环变量为 i(范围为 $1 \sim n$)；以层内元素从左上到右下的序号作为内循环，循环变量为 j(范围为 $1 \sim n+1-i$)。这样循环的执行过程正好与“摆放”自然数的顺序相同。用一个变量 k 模拟要“摆放”的数据，下面的问题就是怎么样将数据存储到对应的数组元素。

数组元素的存取，只能是按行、列号操作的。所以下面用由具体到抽象设计循环的“归纳法”，找出数组元素的行号、列号与层号 i 及层内序号 j 的关系。

(1) 每层内元素的列号都与其所在层内的序号 j 是相同的。因为每层的序号是从第一列开始向右下进行。

(2) 元素的行与其所在的层号及在层内的序号均有关系,具体如下:

第一层行号 1～n,行号与 j 相同;

第二层行号 2～n,行号比 j 大 1;

第三层行号 3～n,行号比 j 大 2;

……

行号起点随层号 i 增加而增加,层内其他各行的行号又随层内序号 j 增加而增加,由于编号起始为 1,i 层第 j 个数据的行下标为 $i-1+j$。

综合以上分析,i 层第 j 个数据对应的数组元素是 $a[i-1+j][j]$。

算法如下:

```
main( )
{int i,j,a[100][100],n,k;
 input(n);
 k = 1;
for(i = 1; i < = n; i = i + 1)
  for(j = 1; j < = n + 1 - i; j = j + 1)
   {a[i - 1 + j][j] = k;
    k = k + 1; }
for(i = 1; i < = n; i = i + 1)
  {print("换行符");
   for(j = 1; j < = i; j = j + 1)
      print(a[i][j]);
  }
}
```

算法说明:仅就层内元素个数而言,内层循环变量 j 的变化过程也可以为 i～n,但这样不利于与列下标对应。

由此例注意,要以问题的特点为依据进行算法设计,循环变量的意义不是固定不变的。其范围及引用等细节,由具体的实例进行归纳,可以比较容易地得到抽象的表达式。

【提示】 由此例可以看出为什么数学上会有很多“猜想”,要证明一个猜想很难,但通过实例总结归纳规律的能力大家都是很强的,这是算法设计基本方法之一。

3.1.2　递归设计要点

上一节中,为了处理重复性的操作,采用的办法是构造循环。本小节则介绍另一种方法,采用递归的办法来实现重复性的操作。在程序设计语言中,已经学习了递归的概念,递归(recursion)是一个过程或函数在其定义或说明中又直接或间接调用自身的一种方法。

递归算法设计,就是把一个大型复杂的问题层层转化为一个与原问题相似的规模较小的问题,在逐步求解小问题后,再返回(回溯)得到大问题的解。递归算法只需少量的步骤就可描述出解题过程所需要的多次重复计算,大大地减少了算法的代码量。

【提示】 无论是“数据结构”课程学习递归算法的原理,还是上一章学习递归算法分析,一个明确的事实是递归算法的时间效率、空间效率都是比较低的,那为什么要学习递归设计呢?请思考。

递归算法设计的关键在于找出递归关系(方程)和递归终止(边界)条件。递归关系就是使问题向边界条件转化的规则。递归关系必须能使问题越来越简单,规模越来越小。递归边界条件就是所描述问题最简单的、可解的情况,它本身不再使用递归的定义。

因此,用递归算法解题,通常有 3 个步骤。

(1) 分析问题、寻找递归关系:找出大规模问题与小规模问题的关系,这样通过递归使问题的规模逐渐变小。

(2) 设置边界、控制递归:找出停止条件,即算法可解的最小规模问题。

(3) 设计函数、确定参数:和其他算法模块一样设计函数体中的操作及相关参数。

下面是一个经典的递归例题。

【例 5】 汉诺塔问题:古代有一个梵塔,塔内有 3 个基座 A,B,C,开始时 A 基座上有 64 个盘子,盘子大小不等,大的在下,小的在上。有一个老和尚想把这 64 个盘子从 A 座移到 B 座,但每次只允许移动一个盘子,且在移动过程中,3 个基座上的盘子都始终保持大盘在下,小盘在上。移动过程中可以利用 C 基座做辅助。请编程打印出移动过程。

问题分析:此问题又称为“世界末日问题”,因为以最高效的移动(无不必要的移动)方法,以每秒移动一次的速度 64 个盘子也需要近 5800 亿年的时间。当然不必真的去解 64 阶汉诺塔问题,一般地对任意 n 阶的汉诺塔问题进行讨论。

算法设计:用人类的大脑直接去解 3,4 或 5 阶的汉诺塔问题(当然是以最高效的移动方法)还可以,但更高阶的问题就难以完成了,更不用说是把问题的解法抽象成循环的机械操作了。所以此问题用递归算法来解合理,即使有非递归算法,也是模仿递归算法的执行过程而得到的。下面用递归法解此题,约定盘子自上而下盘子的编号为 1,2,3,…,n。

首先,看一下 2 阶汉诺塔问题的解,不难理解以下移动过程(括号中是基座现有盘子的号):

初始状态为A(1,2)　　B( )　　C( )
第一步后　A(2)　　B( )　　C(1)
第二步后　A( )　　B(2)　　C(1)
第三步后　A( )　　B(1,2)　　C( )

如何找出大规模问题与小规模问题的关系,从而设计出递归算法呢?在已经会做两个盘子的汉诺塔问题后,这个关系就不难找到了。把 n 个盘子抽象地看作“两个盘子”,上面“一个”由 1～n－1 号组成,下面一个就是 n 号盘子。移动过程如下。

第一步:先把上面“一个”盘子以 A 基座为起点借助 B 基座移到 C 基座。

第二步:把下面一个盘子从 A 基座移到 B 基座。

第三步:再把 C 基座上的“一个”盘子借助 A 基座移到 B 基座。

把 n 阶汉诺塔问题记作 hanoi(n,a,b,c),注意这里 a,b,c 并不总代表 A,B,C 3 个基座,其意义为:第二个参数 a 代表每一次移动的起始基座,第三个参数 b 代表每一次移动的终点基座,第四个参数 c 代表每一次移动的辅助基座。由上述约定,n 阶的汉诺塔问题记作 hanoi(n,a,b,c),a,b,c 初值为“A”“B”“C”,以后的操作等价于以下 3 步。

第一步：hanoi($n-1$,a,c,b)；

第二步：把下面“一个”盘子从 A 基座移到 B 基座；

第三步：hanoi($n-1$,c,b,a)。

至此找出了大规模问题与小规模问题的递归关系。操作过程如下：

(1) 将 A 杆上面的 $n-1$ 个盘子，借助 B 杆，移到 C 杆上，如图 3-1(a)所示；

(2) 将 A 杆上剩余的一个 n 号盘子移到 B 杆上，如图 3-1(b)所示；

(3) 将 C 杆上的 $n-1$ 个盘子，借助 A 杆，移到 B 杆上，如图 3-1(c)所示。

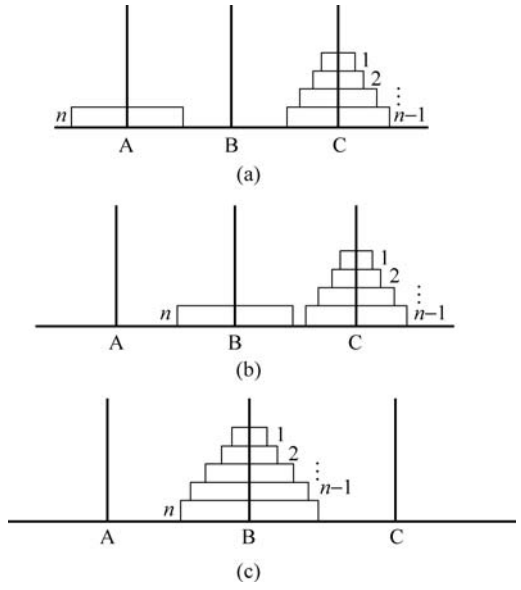

图 3-1　汉诺塔问题求解算法图示

有读者可能想到把 2 阶或 1 阶的汉诺塔问题，当作停止条件，即问题可解的最小规模。其实没有必要，只要用 0 阶的汉诺塔问题当作停止条件即可，这时什么都不需要做。

【提示 1】 设计递归算法时要学会抽象，不要过度思考运行过程，也就是说找到 n 阶问题与 $n-1$ 阶问题的关系后，设计好递归调用关系就好，不用继续思考 $n-1$ 阶问题又如何解决，当然 $n-1$ 阶问题自然调用 $n-2$ 阶……，运行过程由操作系统控制，设计者要学会抽象。

【提示 2】 请读者结合算法理解“参数的意义由位置决定，而不是由变量名决定”这句话。

```
main( )
{int n;
 input(n);
 hanoi (n,"A","B","C");
}
  hanoi (int n,char a,char b,char c)
  { if(n>0)
     {hanoi(n-1,a,c,b);
     输出" Move dish",n."from pile",a," to"b);
```

```
        hanoi(n-1,c,b,a); }
    }
```

递归算法执行中有递归调用的过程和回溯的过程(当然二者是不可分的),递归法就是通过递归调用把问题从大规模归结到小规模,当最小规模得到解决后又把小规模的结果回溯,从而推出大规模的解。

【提示】 如果再思考一下用循环实现此问题,应该发现递归设计要比循环设计简单得多。反过来请尝试将前面一些用循环机制实现的问题用递归机制实现。

下面再看一个比较复杂的递归算法设计。

【例 6】 整数的分划问题。

对于一个正整数 n 的分划,就是把 n 表示成一系列正整数之和的表达式。注意,分划与顺序无关,例如 6=5+1 和 6=1+5 被认为是同一种分划。另外,这个整数 n 本身也算是一种分划。

例如,对于正整数 $n=6$,它可以分划为:

6

5+1

4+2　　4+1+1

3+3　　3+2+1　　3+1+1+1

2+2+2　　2+2+1+1　　2+1+1+1+1

1+1+1+1+1+1

现在的问题是,对于给定的正整数 n,要求编写算法计算出其分划的数目 $P(n)$。

【提示】 这个算法有什么现实意义?

模型建立:这里的目标是要建立递归分划数目的递归公式。

从上面 $n=6$ 的实际例子可以看出,很难找到大规模问题 $P(n)$ 与小规模问题 $P(n-d)$ ($d=1,2,3,\cdots$)的关系。根据 $n=6$ 的实例发现"第一行及以后的数据不超过 6,第二行及以后的数据不超过 5,……,第六行的数据不超过 1"。因此,定义一个函数 $Q(n,m)$,表示整数 n 的"任何加数都不超过 m"的分划的数目,n 的所有分划数目 $P(n)$ 就应该表示为 $Q(n,n)$。

一般地 $Q(n,m)$ 有以下递归关系:

(1) $Q(n,n)=1+Q(n,n-1)$

等式右边的"1"表示 n 只包含一个被加数等于 n 本身的分划;则其余的分划表示 n 的所有其他分划,即最大加数 $m\leqslant n-1$ 的分划。

(2) $Q(n,m)=Q(n,m-1)+Q(n-m,m)$　　$(m<n)$

等式右边的第一部分表示被加数中不包含 m 的分划的数目;第二部分表示被加数中包含(注意不是小于)m 的分划的数目,因为如果确定了一个分划的被加数中包含 m,则剩下的部分就是对 $n-m$ 进行不超过 m 的分划。

到此找到了大规模问题与小规模问题的递归关系,下面是递归的停止条件:

(1) $Q(n,1)=1$,表示当最大的被加数是 1 时,该整数 n 只有一种分划,即 n 个 1 相加;

(2) $Q(1,m)=1$,表示整数 $n=1$ 只有一个分划,不管最大被加数的上限 m 是多大。

算法设计:由以上模型不难写出算法。考虑算法的健壮性,如果 $n<m$,则 $Q(n,m)$ 是

无意义的，因为 n 的分划不可能包含大于 n 的被加数 m，此时令 $Q(n,m)=Q(n,n)$；同样当 $n<1$ 或 $m<1$ 时，$Q(n,m)$也是无意义的。

【提示】 这个整数的分划方法能推广到实数吗？

算法如下：

```
main( )
{int n;
 input( n );
 if(n<1)
    Error("输入参数错误");
 Divinteger(n, n);
}
Divinteger(int n, int m)
{ if(n=1 or m=1)
      return 1;
  else if(n<m)
      return Divinteger(n, n);
  else if(n=m)
      return 1+Divinteger(n,n-1);
  else
      return Divinteger(n,m-1)+Divinteger(n-m,m);
}
```

算法说明：由于算法中，多次进行递归调用，正整数分划的数目随着 n 的增加增长得非常快，大约是以指数级增长，所以此算法不适合对较大的整数进行分划。感兴趣的读者可以在学习完回溯算法后，完成解决此问题的高效算法。

由以上例子可以看出，虽然递归算法与循环设计的思想不同，但由具体实例从“具体到抽象”归纳算法设计的方法是一样的。

【提示】 与求阶层的递归程序比较，这个例题的递归解法是否可以称为“二维递归”？

3.1.3　递归与循环的比较


循环与递归 1

由上一节递归算法设计的例子，不难理解递归也是一种实现“重复操作”的机制。它把“较复杂”操作依次地归结为“较简单”操作，一直归结到“最简单”操作，能方便完成操作为止。在实际运用中，有很多问题的数学模型本来就是递归的，用递归来描述它们不仅非常自然而且证明算法的正确性也相应地比非递归形式容易得多。可以证明：每个迭代算法原则上总可以转换成与它等价的递归算法；反之不然，即并不是每个递归算法都可以转换成与它等价的循环结构算法，例如 3.1.2 节的例 5。

下面通过几个具体的例子来说明循环和递归的差异和优劣。

【例 7】 任给十进制的正整数，请从低位到高位逐位输出各位数字。

循环算法设计：从题目中并不能获知正整数的位数，再看题目的要求，算法应该从低位到高位逐位求出各位数字并输出，详细设计如下。

(1) 求个位数字的算式为 $n \bmod 10$。

(2) 为了保证循环体为“不变式”，求十位数字的算式仍旧为 $n \bmod 10$，这就要通过算式 $n=n\backslash 10$，将 n 的十位数变成个位数。

循环算法如下：

```
main( )
{int n;
 input(n);
  while(n>=10)
   { print(n mod 10);
     n=n\10; }
 print(n);
}
```

递归算法设计：

(1) 同上，算法从低位到高位逐位求出各位数字并输出，求个位数字的算式为 n mod 10，下一步则是递归地求 $n\backslash 10$ 的个位数字。

(2) 当 $n<10$ 时，n 为一位数停止递归。

递归算法如下：

```
main( )
{int n;
 input(n);
 f(n);
}
f(int n)
{if(n<10)
  print(n);
 else
  { print(n mod 10);
  f(n\10); }
}
```

算法分析：循环算法与递归算法无论是时间效率还是空间效率都是前者高。递归算法在运行时，函数递归调用时，需要保存现场，并开辟新的运行资源；返回时，又要回收资源；这都是需要耗费时间的。递归算法的参数 n 表面上是一个变量，实际是一个栈。

【结论 1】 递归工具确实使一些复杂的问题处理起来简单明了；但是，就效率而言，递归算法的实现往往要比循环结构的算法耗费更多的时间和存储空间。所以在具体实现时，方便的情况下应该把递归算法转化成等价的循环结构算法，以提高算法的时空效率。

【例 8】 任给十进制的正整数，请从高位到低位逐位输出各位数字。

循环算法设计：本题目中要求“从高位到低位”逐位输出各位数字，但由于并不知道正整数的位数，因此算法还是“从低位到高位”逐位求出各位数字比较方便。这样就不能边计算边输出，而需要用数组保存计算的结果，最后倒着输出。

循环算法如下：

```
main( )
{int n,j,i=0,a[16];
 input(n);
  while(n>=10)
   { a[i]=n mod 10;
     i=i+1;
```

```
        n = n\10; }
    a[i] = n;
    for(j = i; j >= 0; j = j - 1)
          print(a[j]);
    }
```

递归算法设计：与例 7 不同，递归算法是先递归地求 $n\backslash 10$ 的个位数字，然后再求个位数字 n 的个位数字并输出。这样输出操作是在回溯时完成的。递归停止条件与例 7 相同为 $n<10$。

递归算法如下：

```
main( )
{int n;
 input(n);
 f(n);
}
f(int n)
{if(n < 10)
  print(n);
 else
  { f(n\10);
  print(n mod 10); }
}
```

算法分析：递归算法与循环相比较，它们的空间效率是相等的。虽然时间效率有所差别，但递归程序更简单，可读性好。

【提示】 将以上两个算法中的 10 都用变量 m 代替（当然要做相应的变量说明），以上算法就可以将输入的十进制数 n 转换为 m 进制的数输出，分别思考 m 小于 10 和大于 10 的情况。

下面又是一个有明显需要回溯时完成操作的例题。

【例 9】 任何一个正整数都可以用 2 的幂次方表示。

例如：$137=2^7+2^3+2^0$，同时约定几次方用括号来表示，即 a^b 可表示为 $a(b)$，由此可知，137 可表示为：2(7)＋2(3)＋2(0)，进一步：7＝2(2)＋2＋2(0)（2^1 用 2 表示）3＝2＋2(0)。所以最后 137 可表示为：

2(2(2)＋2＋2(0))＋2(2＋2(0))＋2(0)。

又如：$1315=2^{10}+2^8+2^5+2+1$，所以 1315 最后可表示为：

2(2(2＋2(0))＋2)＋2(2(2＋2(0)))＋2(2(2)＋2(0))＋2＋2(0)。

输入：正整数（$n\leqslant 20\,000$）。

输出：符合约定的 n 的 0，2 表示（在表示中不能有空格）。

算法设计 1：

（1）对复杂问题的操作不要希望一蹴而就，不妨先实现 $137=2^7+2^3+2^0$ 的表示，然后再讨论更复杂的表现形式。

（2）由于不知道数据的位数，加上对数据还是从低位到高位的操作比较简单，而输出显然是由高位到低位进行的，这时就要考虑用递归机制实现算法了。

实现要点：

(1) 比较例 7 和例 8 的算法知道输出操作应该在递归之后。

(2) 为了记录递归的深度，也就是 2 的指数，递归函数的参数应该是两个，一个是当前操作数 n，另一个用来记录递归的深度。

(3) 递归的停止条件本来可以是 0；当 $n==0$ 时，不做任何操作。但由于第一个输出项没有“+”号，其余输出项都有“+”号，所以将递归的停止条件定为 1。

算法 1 如下：

```
main( )
{int n;
 input(n);
 if (n>=1)
   try(n,0);
 else
   print("data error");
}
try(int n,int r)
{if(n=1)
 print("2(",r,")");
 else
   {try(n/2,r+1);
    if(n%2==1)
      print("+2(",r,")");
   }
}
```

算法设计 2：下面处理指数 r 的“2 的幂次方”表示。函数 try(n,0)就是求 n 的“2 的幂次方”表示，所以，递归调用 try(r,0)就可以解决问题。当然，当 $r\leqslant 2$ 时直接按格式输出就可以了。

算法 2 如下：

```
main( )
{int n;
 input(n);
if (n>=1)
try(n,0);
else
  print("data error");
}
try(int n,int r)
{if(n==1)
  switch(r)
  {case 0: print("2(0)"); break;
  case 1: print("2"); break;
  case 2: print("2(2)"); break;
  default: print("2("); try(r,0); print(")");
  }
 else
  {try(n/2,r+1);
```

```
    if(n % 2 == 1)
      switch(r)
        {case 0: print(" + 2(0)"); break;
        case 1: print(" + 2"); break;
        case 2: print(" + 2(2)"); break;
        default: print(" + 2("); try(r,0); print(")");
      }
    }
}
```

【提示】 读者可以尝试用循环控制加数组存储的算法来解决此问题，相信会对下面结论有更深的体会。

【结论 2】 由于递归算法的实现包括递归和回溯两步，当问题需要“后进先出”的操作时，还是用递归算法更有效。如数据结构课程中树的前、中、后序遍历、图的深度优先等算法都是如此。所以不能仅仅从效率上评价两种控制重复操作机制的好坏。

事实上，无论把递归作为一种算法的策略，还是一种实现机制，对设计算法都有很好的帮助。看下面的例子。

【例 10】 找出 n 个自然数(1,2,3,…,n)中取 r 个数的组合。例如，当 $n=5$，$r=3$ 时，所有组合为：


循环与递归 2

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

total=10　{组合的总数}

循环算法设计：分析以上 $n=5$，$r=3$ 的组合实例，5 个数中取 3 个数的 10 组组合，其中每组中的 3 个数有两个特点：(1)互不相同；(2)前面的数小于后面的数。因此，当 $r=3$ 时，可用三重循环模拟每个组合中 3 个数，当满足以上讨论的两个特点时，就得到一组组合。

循环算法如下：

```
main1( )
{int n = 5,i,j,k,t;
 t = 0;
 for(i = 1; i < = n; i = i + 1)
   for(j = 1; j < = n; j = j + 1)
     for(k = 1; k < = n; k = k + 1)
       if ((i < j) and (j < k))
         {t = t + 1;
           print(i,j,k); }
```

```
print('total = ',t);
}
```

或者

```
main2( )
{int n = 5,r = 3,i,j,k,t;
 t = 0;
  for(i = 1; i <= n - r + 1; i = i + 1)
    for(j = i + 1; j <= n - r + 2; j = j + 1)
      for(k = j + 1; k <= n - r + 3; k = k + 1)
        {t = t + 1;
          print(i,j,k); }
 print("total = ",t);
 }
```

循环算法分析：两个算法中，前者穷举了所有可能情形，从中选出符合条件的解，后者则直接按组合中3个数据的特点，确定了相应的循环范围。后者效率更高，但是这两个算法的复杂性均为$O(n^3)$。显然，当n较大时算法的效率是比较低的。

递归算法设计：其实效率问题还是次要的，当要求一个算法能针对不同的r都能给出问题的结果时，由于没有控制循环重数的机制，因此用循环机制解决此问题不具有一般性。而用递归法就不存在以上问题了。

在循环算法设计中，对$n=5$的实例，每个组合中的数据从小到大排列或从大到小排列一样可以设计出相应的算法。但用递归思想进行设计时，每个组合中的数据从大到小排列却是必需的，因为递归算法设计是要找出大规模问题与小规模问题之间的关系。

$n=5,r=3$时，从大到小排列的组合数为：

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

total=10　{组合的总数}

分析以上数据，组合数规律如下：

(1) 固定第一个数5，其后就是求解$n=4,r=2$的组合数，共6个组合。

(2) 固定第一个数4，其后就是求解$n=3,r=2$的组合数，共3个组合。

(3) 固定第一个数3，其后就是求解$n=2,r=2$的组合数，共1个组合。

这就找到了“5个数中3个数的组合”与“4个数中2个数的组合、3个数中2个数的组合、2个数中2个数的组合”的递归关系。

一般地，递归算法的两个步骤为：

（1）n 个数中 r 个数组合递推到“$n-1$ 个数中 $r-1$ 个数的组合，$n-2$ 个数中 $r-1$ 个数的组合……，$r-1$ 个数中 $r-1$ 个数的组合”，共 $n-r+1$ 次递归。

（2）递归的停止条件是 $r=1$。

数据结构设计：本算法的主要操作就是输出，每当递归到 $r=1$ 时就有一组新的组合产生，就应该输出它们和一个换行符。但注意 $n=5$，$r=3$ 的例子中的递归规律，先固定 5，然后要进行多次递归。也就是说，数字 5 要多次输出，所以要用数组存储先确定的一个组合中的数据，以备每一次递归到 $r=1$ 时输出。因为每次向下递归都要用到数组，所以将数组设置为全局变量。

递归算法如下：

```
int a[100];
comb(int m,int k)
{ int i,j;
for (i=m; i>=k; i=i-1)
{ a[k]=i;
  if (k>1)
    comb(i-1,k-1);
  else
    { for (j=a[0]; j>0; j=j-1)
        print(a[j]);
      print("换行符");
    }
  }
}
main3( )
  {int n,r;
   print("n,r=");
   input(n,r);
   if(r>n)
     print("Input n,r error!");
   else
     {a[0]=r;
     comb(n,r); }                    // 调用递归过程
  }
```

递归算法分析：递归算法的递归深度是 r，每个算法要递归 $m-k+1$ 次，所以时间复杂度是 $O(r\times n)$。由这个例题可以看出递归的层次是可以控制的，而循环嵌套的层次只能是固定的。

【结论 3】 递归是一种强有力的算法设计工具。递归是一种比循环更强、更好用的实现“重复操作”的机制。因为递归不需要编程者自己构造“循环不变式”，而只需要找出递归关系和最小问题的解。递归在很多算法策略中得以运用，如分治策略、动态规划、图的搜索等算法策略。

综合以上讨论，结论如表 3-1 所示。

表 3-1 递归与非递归的比较

指　标	递　归	非 递 归
程序可读性	易	难
代码量大小	小	大
时间	长	短
占用空间	大	小
适用范围	广	窄
设计难度	易	难

由此可见,在强调软件维护优先于软件效率的今天,除了像求阶层和斐波那契数列那样的尾递归程序(就是递归调用在程序末尾,可以不设置栈用循环机制实现),其他需要设置栈才能转换为非递归程序的递归程序就没有转换非递归的必要。

3.2 算法与数据结构

现代计算机可以解决的问题种类繁多,计算机解决问题的实质是对“数据”进行加工处理,这里的数据意义是非常广泛的,包括数值、字符串、表格、图形、图像、声音等。而算法也可以定义为算法是对数据运算的描述。

计算机处理的问题类型,粗略地可以分成数值计算问题和非数值性问题。前者主要涉及的运算对象是简单的整型、实型或布尔型数据。程序设计者的主要精力集中于算法设计的技巧,数据结构的选择也比较重要。

随着计算机应用领域的扩大和软、硬件的发展,“非数值性问题”越来越显得重要。据统计,当今处理非数值性问题占用了 90%以上的机器时间,这类问题涉及的数据结构就较为复杂,数据元素之间的相互关系往往无法用数学方程式加以描述。因此,解决此类问题的关键不仅仅是问题分析、数学建模和算法设计,还必须设计出合适的数据结构,才能有效地解决问题。

算法设计的实质是对实际问题要处理的数据选择一种恰当的存储结构,并在选定的存储结构上设计一个好的算法,实现对数据的处理。算法中的操作是以确定的存储结构为前提的,所以,在算法设计中选择好数据结构是非常重要的,选择了数据结构,算法才随之确定。好的算法在很大程度上取决于问题中数据所采用的数据结构。

一般来说,一个实际问题可以建立不同的数据结构。评价这些数据结构,可以从两个基本方面入手:是否准确、完整地刻画了问题的基本特征,是否易于数据存储和对数据处理的实现。

具体可以考虑以下几个问题:

(1) 逻辑结构要能准确表示数据的 3 个层次:数据项、数据元素、数据元素的关系;

(2) 逻辑结构要便于存储实现;

(3) 存储实现方式的选择,要特别考虑数据的规模;

(4) 数据结构一定要方便处理功能的实现;

(5) 数据结构还要利于提高算法的时空性能。

常用的存储结构可以分为连续存储和链式存储。连续存储的空间，如程序设计语言提供的数组，是一个连续的整体，数组名是这个整体的标识，要想使用数组中的某个元素，是通过下标来标识的，而下标可以是变量，这样结合 for 循环，能方便地存储和访问大量数据。且下标可以体现数据间的有“位置”关系的信息。链式存储方式是把逻辑上相邻的结点存储在物理上可能不相邻或相邻的存储单元里，结点间的逻辑关系是由附加的指针字段表示。其优点是可以充分利用所有存储单元，不会出现碎片现象。缺点是不能随机地存储、访问其中的结点。

连续存储又分为静态分配和动态分配两种，静态连续存储就是程序设计语言提供的构造类数据类型——数组（顺序表），需要先说明其大小（数组元素的个数）和数组元素的基本类型然后才能使用。数组的存储空间在编译时就有了逻辑空间，程序装入内存时，就分配了存储空间。所以，若要改变程序中静态数组的大小，必须将程序重新编译才能实现，很不方便，而一个程序或算法要有一定的通用性，可以解决不同规模的同一类型的问题，因此，现在多数的程序设计语言同时也提供了动态存储功能。

动态连续存储就是通过程序设计语言提供的动态存储功能，申请得到的一组指定大小的连续的存储空间。它是在程序运行时才申请内存空间的，可以通过程序交互功能，了解问题的规模后，再确定动态数组的大小。

对连续存储（以静态连续存储方式的顺序表为例）和链式离散存储各有优缺点，比较如下。

1）基于存储的考虑

顺序表的存储空间是静态分配的，在程序执行之前必须明确规定它的存储规模，也就是说事先对“MAXSIZE”要有合适的设定，过大造成浪费，过小造成溢出。可见对线性表的长度或存储规模难以估计时，不宜采用顺序表；链表不用事先估计存储规模，但链表的存储密度较低，存储密度是指一个结点中数据元素所占的存储单元和整个结点所占的存储单元之比。显然链式存储结构的存储密度是小于 1 的。

2）基于运算的考虑

在顺序表中按序号访问 a_i 的时间性能 $O(1)$时，而链表中按序号访问的时间性能 $O(n)$，所以如果经常做的运算是按序号访问数据元素，显然顺序表优于链表；而在顺序表中做插入、删除时平均移动表中一半的元素，当数据元素的信息量较大且表较长时，这一点是不应忽视的；在链表中作插入、删除，虽然也要找插入位置，但操作主要是比较操作，从这个角度考虑显然后者优于前者。

3）基于环境的考虑

顺序表容易实现，任何高级语言中都有数组类型，链表的操作是基于指针（或引用等机制）的，相对而言前者简单些，这也是用户考虑的一个因素。

总之，两种存储结构各有长短，选择哪一种由具体的问题决定。通常“较稳定”的线性表选择顺序存储，而频繁做插入删除的即动态性较强的线性表宜选择链式存储。

具体到现实的软件中，数据在操作前是存储在外存中，所以对于软件的不同模块可采用不同的存储方式将它们存储在内存中，如查询、统计等模块选择顺序存储，而插入、删除等模块选择链式存储。

链式存储结构还可以应用于较复杂的数据结构中，如树、图等。这些结构中的结点间有

多种逻辑关系,用连续存储不足以表示结点间的关系,一般都采用链式存储结构。

由于链表的操作基于指针(或引用等)数据类型,对应算法的可读性较差,所以本书在可能的情况下以连续存储作为存储方式。

下面介绍信息存储中的一些技巧。

3.2.1 原始信息与处理结果的对应存储

解决一个问题时,往往存在多方面的信息。就算法而言,一般有输入信息、输出信息和信息加工处理过程中的中间信息。那么哪些信息需要用数组进行存储,数组元素下标与信息怎么样对应等问题的确定,在很大程度上影响着算法的编写效率和运行效率。

下面的例子恰当地选择了用数组存储的信息,并把题目中的有关信息作为下标使用,使算法的实现过程大大简化。

【例 11】 某校决定由全校学生选举自己的学生会主席。有 5 个候选人,编号分别为 1,2,3,4,5,选举其中一人为学生会主席,每个学生一张选票,只能填写一人。请编程完成统计选票的工作。

算法设计:

(1) 虽然选票发放的数量一般是已知的,但收回的数量通常是无法预知的,所以,算法应该采用随机循环,设计停止标志为“-1”。

(2) 统计过程一般为:先为 5 个候选人各自设置 5 个计数器 S1,S2,S3,S4,S5,然后根据录入数据,通过多分支语句或嵌套条件语句决定为某个计数器累加 1。这样做效率其实很低。

把 5 个计数器用一个具有 5 个元素的数组代替,选票中候选人的编号 xp 正好作下标,这样执行 A(xp)=A(xp)+1 就可方便地将选票结果累加到相应的计数器中。也就是说数组用于存储统计结果,而其下标正好是输入的原始信息。

(3) 考虑到算法的健壮性,要排除对 1~5 之外的数据进行统计。

综上算法如下:

```
main( )
{ int i,xp,a[6] = {0,0,0,0,0,0};
  print("input data until input - 1");
  input(xp);
  while(xp <> - 1)
     {if (xp > = 1 and xp < = 5)
         a[xp] = a[xp] + 1;
      else
      print(xp,"input error!");
      input(xp);
     }
  for (i = 1; i < = 5; i = i + 1)
     print(i,"number get",a[i],"votes");
}
```

此题目中选举的原始信息正好可作为数组下标利用,在实际应用中的多数数据可能没有这样好的规律,不过经算术变化后也可较好地利用这一技巧。

【例 12】 编程统计身高(单位为厘米)。统计分 150～154、155～159、160～164、165～169、170～174、175～179、低于 150 和高于 179,共 8 档次进行。

算法设计:输入的身高可能为 50～250,若用输入的身高数据直接作为数组下标进行统计,即使是用 PASCAL 语言可设置上、下标的下界,也要开辟 200 多个空间,而统计是分 8 个档次进行的,这样是完全没有必要的。

由于多数统计区间的大小都固定为 5,这样用"身高/5-29"做下标,则只需要开辟 8 个元素的数组,对应 8 个统计档次,即可完成统计工作。算法如下:

```
main( )
{ int i,sg,a[8] = {0,0,0,0,0,0,0,0};
  print("input height data until input -1");
  input(sg);
  while (sg<>-1)
    {if (sg>179)
       a[7] = a[7] + 1;
     else
       if (sg<150)
          a[0] = a[0] + 1;
        else
          a[sg/5-29] = a[sg/5-29] + 1;
     input(sg);
    }
  for (i = 0; i<=7; i = i + 1)
    print(i + 1,"field the number of people: ",a[i]);
}
```

算法说明:算法中除了利用数学运算 t=sg/5-29 标识了 7 类统计区间外,还与数组下标对应,减少不必要的条件判断,提高了算法效率。

下面这个例题给出两种算法,它们都利用数组作为存储结构,但存储的对象不同,因此算法的功效和复杂程度也大不相同。

【例 13】 一次考试共考了语文、代数和外语 3 科。某小组共有 9 人,考后各科及格名单如表 3-2 所示,请编写算法找出 3 科全及格的学生的名单(学号)。

表 3-2　各科目及格名单

科目	及格学生学号
语文	1,9,6,8,4,3,7
代数	5,2,9,1,3,7
外语	8,1,6,7,3,5,4,9

算法设计 1:从语文名单中逐一抽出及格学生学号,先在代数名单中查找,若有该学号,说明代数也及格了,再在外语名单中继续查找,看该学号是否外语也及格了,若仍在,说明该学号学生 3 科全及格,否则至少有一科不及格的。语文名单中就没有的学号,不可能 3 科全及格,所以,语文名单处理完后算法就可以结束了。

这其实就是枚举尝试。用 a,b,c 三个数组分别存放语文、代数、外语及格名单,尝试范围为三重循环:

```
i 循环,    初值 0,    终值 6,    步长 1
j 循环,    初值 0,    终值 5,    步长 1
k 循环,    初值 0,    终值 7,    步长 1
```

定解条件：

```
a[i] = b[j] = c[k]
```

共尝试 7×6×8＝336 次。

算法 1 如下：

```
main( )
 {int a[7],b[6],c[8],i,j,k,v,flag;
  for(i = 0; i <= 6; i = i + 1)
    input(a[i]);
  for(i = 0; i <= 5; i = i + 1)
    input(b[i]);
  for(i = 0; i <= 7; i = i + 1)
    input(c[i]);
  for(i = 0; i <= 6; i = i + 1)
    {v = a[i];
     for(j = 0; j <= 5; j = j + 1)
       if (b[j] = v)
         for(k = 0; k <= 7; k = k + 1)
           if(c[k] = v)
             {print(v);
               break; }
     }
}
```

算法设计 2：分析 3 科及格名单，有 9 名学生，开辟 9 个元素的数组 a，作为各学号考生及格科目的计数器。将 3 科及格名单通过键盘录入，无须用数组存储，只要同时用数组 a 累加对应学号的及格科目个数即可。最后，凡计数器的值为 3，就是全及格的学生，否则，至少有一科不及格。

基于以上设计，算法 2 主要包括以下两步：

(1) 用下标计数器累加各学号学生及格科数；

(2) 尝试、输出部分。

累加部分为一重循环，初值 1，终值为 3 科及格的总人次，包括重复部分。计 7＋6＋8＝21，步长 1。

尝试部分的尝试范围为一重循环，初值 1，终值 9，步长 1。定解条件：$a[i]=3$。

算法 2 如下：

```
main( )
{int a[10] = {0,0,0,0,0,0,0,0,0,0},i,xh;
for(i = 1; i <= 21; i = i + 1)
   {input(xh);
   a[xh] = a[xh] + 1; }
for(xh = 1; xh <= 9; xh = xh + 1)
   if(a[xh] = 3)
     print(xh);
}
```

算法分析：该例题的两种解法中，由于数组存储的信息不同，算法的操作方法和复杂度

也各不相同。算法 2 简单扼要且效率较高。

从这个例题读者应该进一步认识到：算法与数据结构(数据的存储方式)是密切相关的。

【提示】 本节介绍的选择存储方式很有限，好的存储方式不仅算法设计简单，还可以大大提高算法的效率，在以后算法设计中还会得到应用。

3.2.2 数组使信息有序化

当题目中的数据缺乏规律时，很难把重复的工作抽象成循环不变式来完成，但先用数组存储这些信息后，它们就变得有序了，问题也就迎刃而解了。

【例 14】 编写算法将数字编号“翻译”成英文编号。

例如：将编号 35706“翻译”成英文编号 three-five-seven-zero-six。

算法设计 1：

(1) 编号一般位数较多，可按长整型输入和存储。

(2) 将英文的“zero～nine”存储在数组中，对应下标为 0～9。这样无数值规律可循的单词，通过下标就可以方便地进行存取、访问了。

(3) 通过求余、取整运算，可以取到编号的各个位数字。用这个数字作下标，正好能找到对应的英文数字。

(4) 考虑输出翻译结果是从高位到低位进行的，而取各位数字，比较简单的方法是从低位开始通过求余和整除运算逐步完成的。所以还要开辟另外一个数组，用来存储从低位到高位翻译好的结果，并同时设置变量记录编号的位数，最后倒着从高位到低位输出结果。

算法 1 如下：

```
main( )
{ int i,a[10],ind;
  long num1,num2;
  char eng[10][6] = {"zero","one","two","three "," four"," five","six","seven","eight","nine"};
  print("Input a num");
  input(num1);
  num2 = num1;
  ind = 0;
  while (num2 <> 0)
  {a[ind] = num2 mod 10;
   ind = ind + 1;
   num2 = num2/10; }
  print(num1,"English_exp: ",eng[a[ind-1]]);
  for(i = ind - 2; i >= 0; i = i - 1)
    print(" - ",eng[a[i]]);
}
```

算法说明 1： 为了方便地取编号的各位数字，数字存入数组是从低位到高位存储的，所以输出时要倒着进行。考虑到输出格式，在循环之前先输出最高位，其余的内容在循环体中输出。

【提示】 以上算法当输入的数据 num1＝0 时，是不能正常运行的。请读者考虑如何

修改。

算法设计 2：编号按字符串类型输入，更符合实际需求。因为：

(1) 用数值类型是无法正确存储以 0 开头的编号，如编号“00001”若按数值类型存储，只能存储为 1。

(2) 按字符串处理编号可以方便地从左到右(从高位到低位)进行，与输出过程相符合，不需要开辟数组存储翻译结果。

(3) 无须通过算术运算取编号的各位数字。

实现要点：由字符串中取出的“数字”编号是字符型，根据字符'0'的 ASCII 码值为 48，则“字符-48”就是字符所对应的数字值，用它作下标就可完成翻译工作了。

算法 2 如下：

```
main( )
 {int i = 0,n;
  char num[40];
  char eng[10][6] = {"zero","one","two","three "," four"," five","six","seven","eight","nine"};
  print("Input a number: ");
  input(num);
  n = strlen(num);      // 取字符串长度
  if(n = 0)
    print("input error!");
  else
    {print(num,"English_exp: ",eng[num[0] - 48]);
     for(i = 1; i <= n - 1; i = i + 1)
       print(" - ",eng[num[i] - 48]);
    }
}
```

算法说明 2：不同的程序设计语言对于字符串的操作差别比较大，算法中以 C 语言为例，给出算法，若用其他语言实现，可以用对应的函数(如取字符的 ASCII 码函数等)完成相应的操作。

把数值类型数据按字符串存储的技巧，在后面一些较复杂的问题中还会用到，如在 3.2.4 节中高精度数据计算问题等。

【例 15】 一个顾客买了价值 x 元的商品(不考虑角、分)，并将 y 元的钱交给售货员。编写算法：在各种币值的钱都很充分的情况下，使售货员能用张数最少的钱币找给顾客。

问题分析：无论买商品的价值 x 是多少，找给他的钱最多需要以下 6 种币值，即 50，20，10，5，2，1。

算法设计：

(1) 为了能达到找给顾客钱的张数最少的目的，应该先尽量多地取大面额的币种，由大面额到小面额币种逐渐进行。

(2) 6 种面额是没有等差规律的一组数据，为了能构造出循环不变式，将 6 种币值存储在数组 B 中。这样，6 种币值就可表示为 $B[i]$，$i=1,2,3,4,5,6$。为了能达到尽量多地找大面额币种的目的，6 种币值应该由大到小存储。

(3) 另外，为统计 6 种面额的数量，还应设置有 6 个元素的累加器数组 S。

综上算法如下：

```
main( )
{int i,j,x,y,z,a,b[7] = {0,50,20,10,5,2,1},s[7];
 input(x,y);
 z = y - x;
 for(i = 1; i <= 6; i = i + 1)
   {a = z\b[i];
    s[i] = a;
    z = z-a * b[i]; }
 print(y," - "x," = ",z: );
 for(i = 1; i <= 6; i = i + 1)
     if (s[i]<> 0)
         print(b[i]," ---- ",s[i]);
 }
```

算法说明：

(1) 每求出一种面额所需的张数后，一定要把这部分金额减去："y＝y－a * b[j]；"，否则将会重复计算。

(2) 算法无论要找的钱 z 是多大，都从 50 元开始统计，所以在输出时要注意合理性，不要输出无用的张数为 0 的信息。

算法分析：问题的规模是常量，时间复杂性肯定为 $O(1)$。

【提示】 这个例题不符合当前流行的支付方式，但这里学习的是算法设计思想，请读者注意：不要认为有用时才学习，有时可能你感受不到例题的应用意义，也要认真学习和体会才能有收获。

3.2.3　数组记录状态信息

有的问题会限定现有数据每个数据只能被使用一次，怎么样区别一个数据"使用过"还是没有"使用过"？学过 PASCAL 语言的读者一定会想到，将已用过的数据存储在定义好的集合类型变量中，以后处理数据时，判断数据是否重复使用，就看它是否包含于该集合即可。

而 C 语言无集合类型，要想判断数据是否重复使用，一个朴素的想法是：用数组存储已使用过的数据，然后每处理一个新数据就与前面的数据逐一比较看是否重复。这样做，当数据量比较大时，判断是否重复的工作效率就会越来越低。

这里介绍一种方法，就是开辟一个状态数组，专门记录数据使用情况。并且将数据信息与状态数组下标对应(好像数据结构课程中介绍的散列存储一样)，就能较好地完成判断是否重复使用的操作。看下面的例子。

【例 16】 求 x，使 x^2 为一个各位数字互不相同的 9 位数。

算法设计：只能用枚举法尝试完成此题。由 x^2 为一个 9 位数，估算 x 应为 10 000～32 000。

(1) 设置 10 个元素的状态数组 p，记录数字 0～9 在 x^2 中出现的情况。数组元素都赋初值为 1，表示数字 0～9 没有被使用过。

(2) 对尝试的每一个数 x，求 $x \times x$，并取其各个位数字，数字作为数组的下标，若对应元素为 1，则该数字第一次出现，将对应的元素赋为 0，表示该数字已出现一次。否则，若对

应元素为0,则说明有重复数字,结束这次尝试。

(3) 容易理解当状态数组 p 中有9个元素为0时,就找到了问题的解。但这样判定有解,需要扫描一遍数组 p。为避免这个步骤,设置一个计数器 k,在取 $x \times x$ 各个位数字的过程中记录不同的数字的个数,当 $k=9$ 时就找到了问题的解。

综上算法如下:

```
main( )
{long x,y1,y2;
 int p[10],2,i,t,k,num = 0;
 for (x = 10 000; x < 32 000; x = x + 1)
  { for(i = 0; i <= 9; i = i + 1)
      p[i] = 1;
    y1 = x * x;
    y2 = y1;
    k = 0;
    for(i = 1; i <= 9; i = i + 1)
      {t = y2 mod 10;
       y2 = y2/10;
       if(p[t] = 1)
        {k = k + 1;
         p[t] = 0; }
      else
        break;
      }
    if(k = 9)
       {num = num + 1;
        print ("No.",num,": n = ",x,"n² = ",y1); }
    }
}
```

算法说明:

(1) 要注意数据类型的选取。

(2) for 循环语句的第二部分是循环条件。根据需要,可以是和循环变量有关或无关的逻辑表达式。

【例17】 游戏问题:

12个小朋友手拉手站成一个圆圈,从某一个小朋友开始报数,报到7的那个小朋友退到圈外,然后他的下一位重新报"1"。这样继续下去,直到最后只剩下一个小朋友。求解这个小朋友原来站在什么位置上。

算法设计: 这个问题初看起来很复杂,又是手拉手,又是站成圈,报数到7时退出圈……似乎很难入手。仔细分析,首先应该解决的是如何表示哪些小朋友还站在圈内,开辟12个元素的数组,记录12个小朋友的状态,开始时将12个元素的数组值均赋为1,表示大家都在圈内。这样小朋友报数就用累加数组元素的值来模拟,累加到7时,该元素所代表的小朋友退到圈外,将相应元素的值改赋为0,这样再累加到该元素时和不会改变,从而模拟了已出圈外的状态。

为了算法的通用性，算法允许对游戏的总人数、报数的起点、退出圈外的报数点任意输入。其中 n 表示做游戏的总人数，k 表示开始报数人的编号及状态数组的下标变量，m 表示退出圈外人的报数点，即报 m 的人出队，p 表示已退出圈外的人数。

算法如下：

```
main( )
{int a[100],i,k,p,m;
 print("input numbers of game: ");
 input(n);
 print("input serial number of game start: ");
 input(k);
 print("input number of out_ring: ");
 input(m);
 for(i=1; i<=n; i=i+1)
  a[i]=1;
 p=0;
 k=k-1;
 print("wash out: ");
 while (p<n-1)
 { x=0;
   while (x<m)
      {k=k+1;
       if(k>n)
        k=1;
       x=x+a[k]; }
  print(k);
  a[k]=0;
  p=p+1;
 }
 for(i=1; i<=n; i=i+1)
   if (a[i]=1)
     print("i=",i);
}
```

算法说明：

(1) 算法中当变量 $k>n$ 时，赋 $k=1$，表示 n 号报完数就该 1 号报数。模拟了将 n 个人连成了一个“圈”的情况(3.3.1 节算术运算的妙用中还介绍了其他技巧，请参考)。

(2) x 表示正在“报”的数，$x=m$ 时输出退出圈外人的下标 k，将 $a[k]$赋值为 0。

(3) $p=n-1$ 时游戏结束。

(4) 最后检测还在圈上 $a[i]=1$ 的人，输出其下标值即编号(原来位置)。

【提示】 请将算法修改为：

(1) 可选择顺时针或逆时针方向报数。

(2) 随机产生游戏的总人数、报数的起点、退出圈外的报数点。

3.2.4　高精度数据存储及运算

计算机存储数据是按数据类型分配存储空间的。在微型机上一般为整型提供与机器字

长相同的空间大小,16 位机分配 2 字节的存储空间,32 位机分配 4 字节的存储空间(有同学一定说我的 64 位机中整型分配的也是 4 字节,那是因为 C 或者 C++编译系统将 64 位机模拟成 32 位机了),虽然现在工作站或小型机等机型上都有更高精度的数值类型,但这些机型价格昂贵,只有大型科研机构才有可能拥有,一般不易接触。当需要在个人计算机上,对超过程序语言整型的多位数(简称为"高精度数据")操作时,只能借助于数组才能精确存储、计算。

在用数组存储高精度数据时,由计算的方便性决定将数据是由低到高还是由高到低存储到数组中;可以每位占一个数组元素空间,也可几位数字占一个数组元素空间。若需从键盘输入要处理的高精度数据,一般用字符型数组存储,这样无须对高精度数据进行分段输入。当然这样存储后,需要有类型转换的操作,不同语言转换的操作差别虽然较大,但都是利用数字字符的 ASCII 码进行的。其他的技巧和注意事项通过下面的例子来说明。本节只针对高精度大整数的计算进行讨论,对高精度实数的计算可以仿照进行。

为了好理解,先讨论一个高精度数据与一个长整数精度范围内的数(一般数)的乘法。

【例 18】 高精度数据×长整数。

算法设计:

(1) 用字符型数组存储从键盘输入的高精度数据,这样无须对高精度数据进行分段输入。

(2) 乘法是按由低位到高位运算的,且常常需要向高位进位,所以高精度数据应该由低位到高位存储在数组 a 中,每个元素存储一位数字。

(3) 每一位仅运算一次,所以乘法的结果也可以存储在数组 a 中。

实现要点:

(1) 要完成高精度数据从字符型数组存储方式到数值数组存储方式的转换。和前面例题看到的一样"数字字符-48"就转换为数值数字,不过字符型数组是由高位到低位存储,而数值数组是由低位到高位存储,用两个下标控制就可方便完成转换操作。

(2) 一个高精度数据与一个自然数的乘法的运算过程,用一重循环来实现,循环变量 i 代表当前参与运算的数组下标,d 存储进位。

算法如下:

```
main( )
{long b,c,d;
int a[256],i,j,n;
char s1[256];
print("Input a great number\n");
input(s1);
print("Input a long integer number\n");
input(c);
n = strlen(s1);                    // 求字符串长度
d = 0;
for (i = 0,j = n - 1; i < n; i++ ,j-- )
  {b=(s1[j]-48) * c+d;
   a[i]=b mod 10;
   d=b/10; }
while (d<>0)
```

```
 {a[n]=d mod 10;
  d=d/10;
  n=n+1; }
for (i=n-1; i>=0; i--)
  print(a[i]);
}
```

算法说明：

(1) 算法中没有独立进行从字符型数组存储方式到数值数组存储方式的转换，而是和运算过程一起进行的。

(2) 有的读者可能注意到了，程序中的乘法并非是像手工运算时那样，乘数、被乘数都按位进行的，程序中高精度数据确实是按位运算，而另一个乘数 c 是整体直接与高精度数据的每一位做乘法运算的。

(3) 由上分析，变量 d 存储的进位可能是较大的(在长整型范围之内)，因此运算完乘法后，不能直接将 d 的值存入结果数组 a 的一个整型存储单元中，而需要将 d 逐位存入结果数组不同的存储单元中。除非，将数组定义为长整型，但那样太浪费存储空间。

【例 19】 编程求当 $n \leqslant 100$ 时，$n!$ 的准确值。

问题分析： 问题要求对输入的正整数 n，计算 $n!$ 的准确值，而 $n!$ 的增长速度高于指数增长的速度，所以这是一个高精度计算问题。请看两个例子。

```
9!=362 880
100!      =   93    326 215    443 944    152 681    699 263
856 266   700 490   715 968    264 381    621 468    592 963
895 217   599 993   229 915    608 914    463 976    156 578
286 253   697 920   827 223    758 251    185 210    916 864
000 000   000 000   000 000    000 000
```

【提示】 虽然前面的提示强调“有时可能你感受不到例题的应用意义，也要认真学习和体会才能有收获”，但这里还是请大家思考求阶层的应用意义！

算法设计： 同上一个例题一样，累乘的结果是按由低位到高位存储在数组 a 中；不同的是由于计算结果位数可能很多，若存储结果的数组中每个存储空间只存储一位数字需要的存储空间太多，且对每一位进行累乘次数也太多。所以将数组定义为长整型，每个元素存储 6 位数字。

一个高精度数据与一个自然数的累乘运算用二重循环来实现，其中一个循环变量 i 代表要累乘的数据，循环变量 j 代表当前累乘结果的数组下标。数据 b 存储计算的中间结果，d 存储超过 6 位数后的进位，数组 a 存储每次累乘的结果，每个元素存储 6 位数字。

算法如下：

```
main( )
{long a[256],b,d;
 int  m,n,i,j,r;
 input(n);
 m = log(n) * n/6 + 2;
```

```
a[1] = 1;
for(i = 2; i <= m; i = i + 1)
  a[i] = 0;
d = 0;
for(i = 2; i <= n; i = i + 1)
{for (j = 1; j <= m; j = j + 1)
  {b = a[j] * i + d;
   a[j] = b mod 1 000 000;
   d = b/1 000 000; }
if(d <> 0)
  a[j] = d;
}
for (i = m; i >= 1; i = i - 1)
    if(a[i] = 0)
       continue;
    else
       {r = i;
       break; }
print(n,"!= ");
print(a[r]," ");
for(i = r - 1; i >= 1; i = i - 1)
   {if (a[i]> 99 999)
     {print(a[i]," ");
     continue; }
   if (a[i]> 9999)
     {print("0",a[i]," ");
     continue; }
   if (a[i]> 999)
     {print("00",a[i]," ");
     continue; }
   if (a[i]> 99)
     {print("000",a[i]," ");
     continue; }
   if (a[i]> 9)
     {print("0000",a[i]," ");
     continue; }
     print("00000",a[i]," ");
   }//for
}
```

算法说明：

(1) 算法中“m＝log(n) * n/6＋2；”是对 $n!$ 位数的粗略估计。这样做算法简单，但效率较低，因为有许多不必要的乘 0 运算。其实，也可以在算法的计算过程中实时记录当前积所占数组元素的个数 m。其初值为 1，每次有进位时 m 增加 1。也就是将算法的第三个 for 循环(内层)中的 if 语句改为：

```
if(d<>0)
    {a[j]=d;
     m=m+1;}
```

这样就提高了算法的效率。

(2) 输出时,首先计算结果的准确位数 r,然后输出最高位数据,在输出其他存储单元的数据时要特别注意,如若计算结果是 123 000 001,a[2]中存储的是 123,而 a[1]中存储的不是 000001,而是 1。所以在输出时,通过多个条件语句才能保证输出的正确性。

【提示】 输出设计也是算法实现或者说软件质量重要的指标,请读者注意!

3.2.5　构造趣味矩阵

现实中的很多二维表格需要用二维数组存储,所以二维数组的应用也是很广泛的。关于二维数组的基础应用程序设计课程中已经做了介绍,这里就不重复了。

趣味矩阵是为了训练学生的观察和分析能力而设计的一种智巧类问题,有些趣味矩阵或图形可以通过程序设计语言提供的字符串函数和定位输出函数来实现,但还有一些趣味矩阵的规律是无法通过字符串函数模拟的。发现趣味方阵中字符或数据的规律一般是很容易的,但要按规律直接在计算机显示或打印出它们却不是那么容易;因为无论是屏幕显示还是打印机输出,比较方便的操作都是从上到下,从左到右进行。

解决问题的办法是:根据趣味矩阵中的数据规律,设计算法把要输出的数据先存储到一个二维数组中,最后按行输出该数组中的元素。这类练习,对大家熟练掌握二维数组的操作很有帮助。

先复习一些有关二维表和二维数组的基本常识:

(1) 当对二维表按行进行操作时,应该"外层循环控制行;内层循环控制列";反之若要对二维表按列进行操作时,应该"外层循环控制列;内层循环控制行"。

(2) 二维表和二维数组的显示输出,只能按行从上到下连续进行,每行各列则只能从左到右连续输出。所以,只能用"外层循环控制行;内层循环控制列"。

(3) 用 i 代表行下标,以 j 代表列下标(除特别声明以后都遵守此约定),则对 $n\times n$ 矩阵有以下常识:

主对角线元素 $i=j$;

副对角线元素下标下界为 1 时 $i+j=n+1$,下标下界为 0 时 $i+j=n-1$;

主上三角◥元素 $i\leqslant j$;

主下三角◣元素 $i\geqslant j$;

次上三角◤元素:下标下界为 1 时 $i+j\leqslant n+1$,下标下界为 0 时 $i+j\leqslant n-1$;

次下三角◢元素:下标下界为 1 时 $i+j\geqslant n+1$,下标下界为 0 时 $i+j\geqslant n-1$。

下面通过例子学习和掌握趣味矩阵的构造。

【例 20】 编程打印有如下规律的 $n\times n$ 方阵。

使左对角线和右对角线上的元素为 0,它们上方的元素为 1,左边的元素为 2,下方元素为 3,右边元素为 4,下图是一个符合条件的五阶矩阵。

```
0  1  1  1  0
2  0  1  0  4
2  2  0  4  4
2  0  3  0  4
0  3  3  3  0
```

算法设计：根据数据分布的特点，利用以上关于二维数组的基本常识，只考虑可读性的情况。

算法如下：

```
main( )
{int i,j,a[100][100],n;
 input(n);
 for(i=1; i<=n; i=i+1)
  for(j=1; j<=n; j=j+1)
     {if (i=j or i+j=n+1) a[i][j]=0;
      if (i+j<n+1 and i<j) a[i][j]=1;
      if (i+j<n+1 and i>j) a[i][j]=2;
      if (i+j>n+1 and i>j) a[i][j]=3;
      if (i+j>n+1 and i<j) a[i][j]=4;
      }
for(i=1; i<=n; i=i+1)
  {print("换行符");
   for(j=1; j<=n; j=j+1)
     print(a[i][j]);
  }
}
```

算法分析：为了算法的可读性，以上算法没有考虑算法效率，对每一对(i,j)都要进行5次判断，若用嵌套if语句则可以改善。

【例21】 螺旋阵：任意给定n值，按如下螺旋的方式输出方阵。

$n=3$　输出：1　8　7
　　2　9　6
　　3　4　5

$n=4$　输出：1　12　11　10
　　2　13　16　9
　　3　14　15　8
　　4　5　6　7

算法设计1：此例可以按照“摆放”数据的过程，逐层(圈)分别处理每圈的左侧、下方、右侧、上方的数据。以$n=4$为例详细设计如下：

把“1～12”看作一层(一圈)，“13～16”看作二层……以层作为外层循环，下标变量为i。由以上两个例子，$n=3$、4均为两层，用$n\backslash 2$表示下取整，$(n+1)/2$表示对$n/2$上取整。所以下标变量i的范围为$1\sim(n+1)/2$。

i层内“摆放”数据的4个过程为(四角元素分别归4个边)：

(1) i列(左侧)，从i行到$n-i$行　　($n=4,i=1$时“摆放1,2,3”)。

(2) $n+1-i$ 行(下方),从 i 列到 $n-i$ 列　　($n=4,i=1$ 时“摆放 4,5,6”)。

(3) $n+1-i$ 列(右侧),从 $n+1-i$ 行到 $i+1$ 行　　($n=4,i=1$ 时“摆放 7,8,9”)。

(4) i 行(上方),从 $n+1-i$ 列到 $i+1$ 列　　($n=4,i=1$ 时“摆放 10,11,12”)。

4 个过程通过 4 个循环实现,用 j 表示 i 层内每边中行或列的下标。

算法 1 如下:

```
main( )
{int i,j,a[100][100],n,k;
 input(n);
 k=1;
  for(i=1; i<=n/2; i=i+1)
   {for(j=i; j<=n-i; j=j+1)            // 左侧
      {a[j][i]=k;
       k=k+1; }
    for(j=i; j<=n-i; j=j+1)            // 下方
       {a[n+1-i][j]=k;
        k=k+1; }
    for(j=n-i+1; j>=i+1; j=j-1)        // 右侧
       {a[j][n+1-i]=k;
        k=k+1; }
    for(j=n-i+1; j>=i+1; j=j-1)        // 上方
       {a[i][j]=k;
        k=k+1; }
   }
 if (n mod 2=1)
   {i=(n+1)/2;
    a[i][i]=n*n; }
 for(i=1; i<=n; i=i+1)
   {print("换行符");
    for(j=1; j<=n; j=j+1)
     print(a[i][j]);
 }
}
```

算法 1 说明:

(1) 当 n 为奇数时,中间一层只有一个数据,需要特殊处理,这就是算法中 if 语句的功能。

(2) 算法中没有考虑输出数据的位数,输出效果并不好,用程序设计实现算法时要考虑这方面的问题。

算法设计 2:下面通过设置变量标识一圈中不同方位的处理差别,并通过算术运算将 4 个方位的处理归结成一个循环完成。这是一个比较复杂的构造循环“不变式”的过程,读者注意学习、体会其中的方法和技巧。

这里还是模拟手工“摆放”数据的过程将 1 到 $n\times n$ 依次存入数组 $a[n][n]$中,在螺旋方阵的元素被放入时,关键是要确定一圈中行、列下标 i 和 j 的变化过程和范围。约定让四角数据分别属于先处理的行或列,如:$n=4$ 时“4”属于第 1 列,“7”属于第 4 行……用 k 记录行或列处理的元素个数,以下为存放最外一圈的情况。

$j=1$　　$i=i+1$　　$1\sim n$　　$k=n$　　//左侧
$i=n$　　$j=j+1$　　$2\sim n$　　$k=n-1$　　//下方
$j=n$　　$i=i-1$　　$n-1\sim 1$　　$k=n-1$　　//右侧
$i=1$　　$j=j-1$　　$n-1\sim 2$　　$k=n-2$　　//上方

从上面 i,j 的变化可以发现这样的规律：往圈内“摆放”数据时,在处理某圈的前半圈(左侧和下方)时,下标 i,j 的变化是一致的,都加 1；在处理某圈的后半圈(右侧和上方)时,下标 i,j 的变化也是一致的,都在减 1。不同的是变化的范围不一致。

为了用统一的表达式表示循环变量的范围,引入变量 k,k 表示在某一方向上数据的个数,k 的初值是 n,每当数据存放到左下角时,k 就减 1,又存放到右上角时,k 又减 1；此时的 k 值又恰好是下一圈左侧的数据个数。

同样,在向下(左侧)和向上(右侧)“摆放”数据时,为了将行下标 i 的变化用统一的表达式表示；同时在向右(下方)向左(上方)“摆放”数据时,也将列下标 j 的变化用统一的表达式表示。引入符号变量 t,t 的初值为 1,表示处理前半圈：在左侧行下标 i 向下变大,在下方列下标 j 向右变大；t 就变为 -1；表示处理后半圈：在右侧行下标 i 向上变小,在上方列下标 j 向左变小。于是一圈内下标的变化情况如下：

$j=1$　　$i=i+t$　　$1\sim n$　　$k=n$
$i=n$　　$j=j+t$　　$2\sim n$　　前半圈共 $2\times k-1$ 个
$t=-t$　　$k=k-1$
$j=n$　　$i=i+t$　　$n-1\sim 1$
$i=1$　　$j=j+t$　　$n-1\sim 2$　　后半圈共 $2\times k-1$ 个
$t=-t$　　$k=k-1$

再看下一圈,同样前半圈共 $2\times k-1$ 个数据,执行 $k=k-1$ 后,后半圈也是共 $2\times k-1$ 个数据,是很好的循环不变式。关于行列下标的循环不变式,看完算法设计结果再进行解释。

用 x 模拟“摆放”的数据；用 $y(1\sim 2\times k-1)$ 作循环变量,模拟半圈内数据的处理的过程。

算法 2 如下：

```
main( )
{int i,j,k,n,a[100][100],b[2],x,y;
     input(n);
     b[0] = 0;
     b[1] = 1;
     k = n;
     t = 1;
     x = 1;
    while (x<=n*n)
      {for (y = 1; y<=2*k-1; y = y+1)        // t = 1 时处理左下角,t = -1 时处理右上角
          { b[y/(k+1)] = b[y/(k+1)] + t;
            a[b[0]][b[1]] = x;
            x = x+1; }
       k = k-1;
       t = -t;
      }
```

```
for(i = 1; i <= n; i = i + 1)
  {print("换行符");
   for(j = 1; j <= n; j = j + 1)
     print(a[i][j]);
  }
}
```

算法 2 说明：在“算法设计”中没有介绍数组 b，这里说明它的用途。由算法中的“a[b[0]][b[1]]＝x；”语句可以体会到，数组元素 $b[0]$表示存储矩阵的数组 a 的行下标，数组元素 $b[1]$是数组 a 的列下标。那么为什么不用习惯的 i，j 作行、列的下标变量呢？使用数组元素作下标变量的必要性是什么？表达式“b[y/(k＋1)]＝b[y/(k＋1)]＋t；”的意义又是什么？下面逐步解释。

“算法设计”中已说明，y 用作循环变量，模拟半圈内数据的处理的过程。变化范围是 $1\sim2\times k-1$。而每个在半圈里：

(1) 当 $y=1\sim k$ 时，表达式 $y/(k+1)$的值始终为 0，这样列下标 $b[1]$不会变化，而行下标 $b[0]$通过表达式 $b[y/(k+1)]=b[y/(k+1)]+t$；在变化(加 1 或减 1，由 t 决定)。

(2) 当 $y=k+1\sim2\times k-1$ 时，表达式 $y/(k+1)$的值始终为 1，这样行下标 $b[0]$不会变化，而列下标 $b[1]$ 通过表达式 $b[y/(k+1)]=b[y/(k+1)]+t$；在变化(加 1 或减 1，由 t 决定)。

这又是 3.2.2 节介绍的利用一维数组“构造循环”技巧的一个实例，当然也离不开用算术运算对数组下标进行构造。

综上所述，引入变量 t，k 和数组 b 后，通过算术运算将一圈中的上下左右 4 种不同的变化情况，归结构造成了一个循环“不变式”。

【例 22】 魔方阵是我国古代发明的一种数字游戏：n 阶魔方是指这样一种方阵，它的每一行、每一列以及对角线上的各数之和为一个相同的常数，这个常数是：$n\times(n^2+1)/2$，此常数被称为魔方阵常数。由于偶次阶魔方阵(n＝偶数)求解起来比较困难，这里只考虑 n 为奇数的情况。

以下就是一个 $n=3$ 的魔方阵：

6	1	8
7	5	3
2	9	4

它的各行、各列及对角线上的元素之和为 15。

问题分析：如果采用穷举方法，对数据的各种分布进行判断是否满足魔方阵条件，那么当 n 比较大时，即使用计算机也需要用很长时间才能找出解来。

算法设计：有趣的是如果将 $1,2,\cdots,n^2$ 按某种规则依次填入到方阵中，得到的恰好是奇次魔方阵，这个规则可以描述如下。

(1) 将 1 填在方阵第一行的中间，即$(1,(n+1)/2)$的位置。

(2) 下一个数填在上一个数的主对角线的上方，若上一个数的位置是(i,j)，下一个数应填在(i_1,j_1)，其中 $i_1=i-1$、$j_1=j-1$。

(3) 若应填写的位置下标出界，则出界的值用 n 来替代，即若 $i-1=0$，则取 $i_1=n$，若 $j-1=0$，则取 $j_1=n$。

(4) 若应填的位置虽然没有出界,但是已经填有数据,则应填在上一个数的下面(行减1,列不变),即取 $i_1=i-1$,$j_1=j$。

(5) 这样循环填数,直到把 $n\times n$ 个数全部填入方阵中,最后得到的是一个 n 阶魔方阵。

算法如下:

```
main( )
{int i,j,i1,j1,x,n,a[100][100];
print("input an odd number: ");
input(n);
if (n mod 2 = 0)
  {print("input error!");
    return; }
for(i = 1; i <= n; i = i + 1)
    for(j = 1; j <= n; j = j + 1)
      a[i][j] = 0;
i = 1;
j = int((n + 1)/2);
x = 1;
while (x <= n * n)
   {a[i][j] = x;
   x = x + 1;
   i1 = i;
   j1 = j;
   i = i - 1;
   j = j - 1;
   if (i = 0) i = n;
   if (j = 0) j = n;
   if (a[i][j]<> 0)
     { i = i1 + 1;
       j = j1; }
   }
for(i = 1; i <= n; i = i + 1)
  {print("换行符");
   for(j = 1; j <= n; j = j + 1)
    print(a[i][j]);
  }
}
```

算法说明:若当前位置已经填有数,则应填在上一个数的下面,所以需要用变量记录上一个数据填入的位置,算法中 $i1$,$j1$ 的功能就是记录上一个数据填入的位置。

算法分析:算法的时间复杂度为 $O(n^2)$。

【提示】 此算法完全依赖数学家发现的魔方阵数据规律,在大数据时代,很多数据处理方法依赖数学基础,所以应该重视数学的学习和应用。

3.2.6 一维与二维的选择

至此可以发现,一维数组在算法设计中发挥了较强的作用,二维数组是否仅能存储与二维表有关的信息呢?回答是否定的,根据数据信息的特点,二维数组同样可以在算法设计和

实现中起到重要的作用。

下面例题的原始数据表面看是一维信息，与二维数组无关，但使用二维数组存储有关信息后，更容易设计算法。

【例 23】 统计问题：找链环数字对的出现频率。

输入 n(2≤n≤100)个数字(即 0～9 的数据)，然后统计出这组数中相邻两数字组成的链环数字对出现的次数。如：当既有(0,3)又有(3,0)出现时，称它们为链环数字对，算法要统计输出它们各自的出现次数。例如：

输入：n=20 {表示要输入数的数目}

0　1　5　9　8　7　2　2　2　3　2　7　8　7　8　7　9　6　5　9

输出：(7,8)=2 (8,7)=3 {指(7,8),(8,7)数字对出现次数分别为 2 次、3 次}

(7,2)=1 (2,7)=1

(2,2)=2

(2,3)=1 (3,2)=1

数据结构设计：由于，事先不知道存在哪些链环数字对，只能为所有可能的数字对设置计数器，可能的数字对有如下几种。

(0,0)(0,1)…(0,9)

(1,0)(1,1)…(1,9)

⋮

(9,0)(9,1)…(9,9)

共有 100 个数字对，需要 100 个计数器。

这时若用 100 个元素的一维数组作为计数器，很难将数组下标与数字对进行对应。而用 10×10 的二维数组 a(行、列下标均为 0～9)作为计数器，i 行 j 列存储数据正好对应数字对(i,j)出现的次数。

由于每个时刻只须处理两个原始数据，且不必重复处理这些数据，所以原始数据不需要用数组进行存储。

算法设计：有了以上的存储结构，算法就很简单了，每输入一个原始数据(第一次输入两个)，用其做下标对数组 a 的相应元素累加 1。这样在数据输入的同时就能统计出问题的结果。

算法如下：

```
main( )
    {int a[10][10],m,i,j,k1,k0;
     for(i = 0; i <= 9; i = i + 1)
       for(j = 0; j <= 9; j = j + 1)
          a[i][j] = 0;
    print("How many is numbers");
     input(n);
    print("Please input these numbers: ");
     input(k0);
    for (i = 2; i <= n; i = i + 1)
     {input(k1);
      a[k0][k1] = a[k0][k1] + 1;
```

```
        k0 = k1;
        }
      for (i = 0; i <= 9; i = i + 1)
        for (j = 0; j <= 9; j = j + 1)
          if (a[i][j]<> 0 and a[j][i]<> 0);
            print("(",i,j,") = ",a[i][j],",(",j,i,") = ",a[j][i]);
}
```

算法分析：算法的时间效率是很高的，只是空间上有些浪费，因为问题中不出现的数字对也设置了计数器空间。

下面这个例题的数据也没有二维的属性，算法中使用二维数组存储原始信息，并在其上进行了操作，使算法的效率更高(减少了统计、计算和判断过程)。

【例 24】 有 $3n$ 个花盆，红色、蓝色和黄色的各 n 个。开始时排列的顺序是混乱的，如黄、红、蓝、黄、黄、蓝、黄、红、红、黄、蓝、红、黄、红、黄、蓝、蓝、红、红、红、黄、蓝、蓝、黄、黄、黄、红、红、蓝、蓝、蓝。

请编写一程序：将各花盆按红、黄、蓝、红、黄、蓝……的顺序排列，而且要求花盆之间的交换次数最少。

问题分析：本题是按约定排序的问题，目标是将红花盆送到序号为 $3i-2$ 的元素，也就是 1,4,7,10,…，将黄花盆送到序号为 $3i-1$ 的元素，也就是 2,5,8,…，将蓝花盆送到序号为 $3i(i=1,2,\cdots,n)$的元素，也就是 3,6,9,…，并有特殊要求：花盆之间交换次数最少。

现用数字 1 表示红花盆，2 表示黄花盆，3 表示蓝花盆。

交换两个变量 A、B 的值可以表示成 $D=A$，$A=B$，$B=D$(D 是中间变量)。若 3 个变量 A、B、C 作循环式交换，则可示意成 $D=A$，$A=B$，$B=C$，$C=D$。因此，直接交换两个花盆的运算为 3 次，那么 3 个花盆作循环式交换运算为 4 次。

所以为了满足题目的要求，必须保证：

(1) 原序号为 $3i-2$ 的红花盆，为 $3i-1$ 的黄花盆，为 $3i$ 的蓝花盆($i=1,2,\cdots,n$)保持原位置不变。

(2) 应尽量进行两个花盆之间的直接交换，如：应放红花盆处的黄花盆，应尽量与放黄花盆的红花盆直接交换。

(3) 最后才进行必要的 3 个花盆作循环式交换工作，如：将红花盆处放的黄花盆，黄花盆处的蓝花盆，蓝花盆处的红花盆作循环式交换。

算法设计：为了知道红、黄花盆直接交换次数，红、蓝花盆直接交换次数，黄、蓝花盆直接交换次数以及循环交换的次数。可预先统计出该放红花盆处的黄、蓝花盆个数 $R1$,$R2$，该放黄花盆处的红、蓝花盆个数 $S1$,$S2$ 和该放蓝花盆处的红、黄花盆个数 $T1$,$T2$。

红花盆和黄花盆的直接交换次数 $n1=\min(R1,S1)$；

红花盆和蓝花盆的直接交换次数 $n2=\min(R2,T1)$；

黄花盆和蓝花盆的直接交换次数 $n3=\min(S2,T2)$。

若 $R1>S1$，则此时红花盆处还有黄花盆，显然此时黄花盆处必然有蓝花盆……因此循环交换次数为：$N4=R1-S1$。

实现要点：实现中并不用专门统计交换次数，而是将 $3n$ 个数据存储在 $n\times3$ 的二维数组空间中，这样第一列应放红花盆、第二列应放黄花盆、第三列应放蓝花盆。程序主要由直

接交换和循环交换两部分组成。

直接交换由二重循环完成，定解条件为（i、j 代表需要交换花盆的行数）：

```
if (a[i][1] = 2 && a[j][2] = 1)        // 红、黄花盆直接交换
if (a[i][1] = 3 && a[j][3] = 1)        // 红、蓝花盆直接交换
if (a[i][1] = 3 && a[j][3] = 2)        // 黄、蓝花盆直接交换
```

循环交换由三重循环完成，定解条件为（i、j、k 代表需要交换花盆的行数）：

```
if (a[i][1] = 2 && a[j][2] = 3 && a[k][3] = 1)
      {v = [i][1]; a[i][1] = a[k][3]; a[k][3] = a[j][2]; a[j][2] = v; }
if (a[i][1] = 3 && a[j][2] = 1 && a[k][3] = 2)
      {v = a[i][1]; a[i][1] = a[j][2]; a[j][2] = a[k][3]; a[k][3] = v}
```

算法如下：

```
main( )
{int n,a[100][4],i,j,k,t,m = 0;
 input(n);
 for(i = 1; i <= n; i++ )
  for(j=1; j<=3; j++ )
    input (a[i][j]);
for(i=1; i<=n; i++ )
  {if (a[i][1]=2)
     for(j=1; j<=n; j++ )
      if (a[j][2]=1)
        {t=a[i][1];
         a[i][1]=a[j][2];
         a[j][2]=t;
         m=m+3;
         break; }
  if (a[i][1]=3)
    for(j=1; j<=n; j++ )
       if (a[j][3]=1)
         {t=a[i][1];
          a[i][1]=a[j][3];
          a[j][3]=t;
          m=m+3;
          break; }
  if (a[i][2]=3)
    for(j=1; j<=n; j++ )
       if (a[j][3]=2)
         {t=a[i][2];
          a[i][2]=a[j][3];
          a[j][3]=t;
          m=m+3;
          break; }
  }
for(i=1; i<=n; i++ )
  {if (a[i][1]=2)
     for(j=1; j<=n; j++ )
      if (a[j][2]=3)
```

```
        for(k=1; k<=n; k++)
          if (a[j][3]=1)
            {t=a[i][1];
             a[i][1]=a[j][2];
             a[j][2]=a[k][3];
             a[k][3]=t;
             m=m+4;
             break; }
    if (a[i][1]=3)
      for(j=1; j<=n; j++)
        if (a[j][2]=1)
         for(k=1; k<=n; k++)
          if (a[j][3]=2)
            {t=a[i][1];
             a[i][1]=a[j][2];
             a[j][2]=a[k][3];
             a[k][3]=t;
             m=m+4;
             break; }
    }
  print ("move=",m);
  for(i=1; i<=n; i++)
    { print ("换行符");
      for(j=1; j<=3; j++)
        print (a[i][j]);
    }
  }
```

【提示】 当前的各类程序设计语言都支持三维以上的多维数组存储方式,请读者思考多维数组可能的用途。本书 4.5.1 节的例题中用到了三维数组。

3.3 优化算法的基本技巧

3.3.1 算术运算的妙用

有关算术运算的妙用,在前面的许多例题中已有体现。例如:3.2.1 节的例 12,通过算术运算把数据信息归类后与下标对应。又如 3.2.5 节“构造趣味矩阵”中例 21 的算法 2,通过算术运算构造循环不变式。

总之,通过恰当的算术运算可以很好地提高编程效率,以及相关算法的运行效率。值得认真总结学习。

【例 25】 一次考试,共考了 5 门课。统计 50 个学生中至少有 3 门课成绩高于 90 分的人数。

问题分析:一个学生 5 门课的成绩分别记为:$a1$,$a2$,$a3$,$a4$,$a5$,要表示有 3 门课成绩高于 90 分,有 $C_5^3=10$ 组逻辑表达式,每组逻辑表达式中有 3 个关系表达式。无论书写还是运行,效率都极低。但通过算法运算就能很简便地解决这类问题。

算法设计：

(1) 对每个同学，先计算其成绩高于 90 分的课程数目，若超过 3，则累加到满足条件的人数中。

(2) 用二重循环实现以上过程，外层循环模拟 50 个同学，内层循环模拟 5 门课程。

算法如下：

```
main( )
{ int a[5],i,j,s,num = 0;
  for (i = 1; i <= 50; i = i + 1)
   {s = 0;
    for(j = 0; j <= 4; j = j + 1)
     {input(a[j]);
      if(a[j]>= 90)
        s = s + 1;
     }
     if(s >= 3)
       num = num + 1;
    }
  print("The number is",num);
}
```

算法说明：因为当成绩≥90 时，C 语言规定关系表达式“成绩>=90”的值为 1，所以计算某人成绩高于 90 分的课程数目，还可以简单地实现如下。

```
s = 0;
for(j = 0; j <= 4; j = j + 1)
    {input(a[j]);
     s = s + (a[j]>= 90);
    }
```

又当已知各门课的成绩最高不超过 179 分时(一门课程成绩最多可能包含一个 90)，还可以用语句“s=s+a[j]\90；”代替语句“s=s+(a[j]>=90)；”，这样适合更多语言实现。

这个算法通过加法运算避免了复杂的逻辑表达式。下面的例子通过简单的算术运算，模拟了状态变化，很好地避免了条件判断。

【例 26】 开灯问题：有从 1 到 n 依次编号的 n 个同学和 n 盏灯。1 号同学将所有的灯都关掉；2 号同学将编号为 2 的倍数的灯都打开；3 号同学则将编号为 3 的倍数的灯做相反处理(该号灯如打开的，则关掉；如关闭的，则打开)；以后的同学都将自己编号的倍数的灯，做相反处理。问经 n 个同学操作后，哪些灯是打开的？

问题分析：

(1) 前面已经学过用数组表示多个对象的某种状态，这里就定义有 n 个元素的 a 数组，它的每个下标变量 $a[i]$视为一灯，i 表示其编号。$a[i]=1$ 表示第 i 盏灯处于打开状态，$a[i]=0$ 表示第 i 盏灯处于关闭状态。

(2) 那么如何实现将第 i 盏灯做相反处理的开关灯操作呢？大家马上想到的是用条件语句 if 表示：当 $a[i]$为 1 时，$a[i]$被重新赋为 0；当 $a[i]$为 0 时，$a[i]$被重新赋为 1。这里要介绍的是，通过算术运算 $a[i]=1-a[i]$，就能很好地模拟“开关”灯的操作。把这种形式的赋值语句形象地称为“乒乓开关”，大家在以后的算法设计中可以借鉴。

算法如下：

```
main( )
{int n,a[1000],i,k;
 print("input a number");
  input(n);
 for(i=1; i<=n; i=i+1)
    a[i]=0;
 for(i=2; i<=n; i=i+1)
    {k=1;
     while (i*k<=n)
       {a[i*k]=1-a[i*k];
        k=k+1; }
    }
 for(i=1; i<=n; i=i+1)
    if (a[i]=1) print(i);
}
```

算法说明：算法中第二个 for 循环 i 枚举的不是灯的编号，而是编号为 i 的同学，其内层循环中，就将包含 i 因数的灯，也就是编号为“i * k”的灯，改变其状态。算法中还用计算省去了用 if 语句判断编号能被哪些数整除的过程。

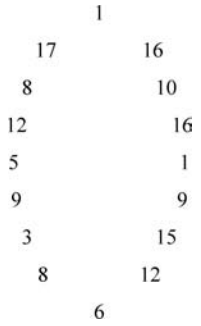

图 3-2　数字圆圈

【提示】 算术运算 $a[i]=1-a[i]$，又称为开关运算，无须判断当前值，模拟开关操作，效率高。首先提醒同学应该记忆这类技巧，其次有同学有疑问仅提高效率，考研又不评价效率，有必要记忆吗？其一对于纸质的考研试题确实多数不评价效率，但现在已经有很多学校考研开始加试上机考试；其二学习算法不应该仅仅为了应试。

【例 27】 图 3-2 所示的圆圈中，把相隔一个数据的两个数(如图 3-2 中的 1 和 10，3 和 5，3 和 6)称作是“一对数”，编写算法求出乘积最大的一对数和乘积最小的一对数。输出格式如下：

```
max=?*?=?
min=?*?=?
```

其中“?”表示：找到的满足条件的数和乘积。

算法设计：

(1) 题目中的数据有前后“位置”关系，因此必须用数组来存储。设数组定义为 $a[num]$，则有 $a[0]$～$a[num-1]$共 num 个元素。

(2) 用 i 代表下标，题目就是顺序将 $a(i-1)$与 $a(i+1)$相乘，求出乘积的最大值和最小值即可。

(3) 关键问题是要使 $i=num-1$ 时，保证 $i+1$ 的“值”是 0；当 $i=0$ 时，保证 $i-1$ 的“值”是 num－1，即怎样将线性的数组当成圆圈操作呢？不难看出，把数组当成圆圈操作通过求余运算很容易实现，例如：

当 $i=num-1$ 时，$(i+1)$ mod num 等于 0；

当 $i=0$ 时，$(num+i-1)$ mod num 等于 num－1。

这样的表达式，当 i 为其他值时，也是同样适用的。

通过求余运算，就“避免了”判别数组起点与终点的操作，其实在“数据结构”课程中的循环队列就是利用这种技巧实现的。

(4) 用变量 max 记录当前最大的乘积，m、n 为对应的两个乘数；变量 min 记录当前最小的乘积，s、t 为对应的两个乘数。

算法如下：

```
main( )
{int max = 1,min = 32767,a[100],num,i,k,m,n,s,t;
 print("input a number");
input(num);
 for(i = 0; i < num; i = i + 1)
   input(a[i]);
 for(i = 0; i < num; i = i + 1)
   { p = (num + i - 1) mod num;
     q = (i + 1) mod num;
     k = a[p] * a[q];
     if (k > max)
      {max = k;
       m = a[p];
       n = a[q]; }
     if (k < min)
        {min = k;
         s = a[p];
         t = a[q]; }
   }
print("max = " m," * ",n," = ",max);
print("min = " s," * ",t," = ",min);
}
```

【提示】 请总结一下求余运算 mod 的作用。并写出判断两个数 a，b 为一奇一偶最简单的表达式。

3.3.2　标志量的妙用

在学习了程序设计语言后，已知道用逻辑表达式可以表示不同情况，从而通过选择语句实现在不同情况下，进行不同的操作。前面曾介绍了通过算术运算或数组实现简化逻辑表达式或减少条件判断的技巧。这里介绍的标志量方法是另一种表示不同情况或状态的实用方法，下面通过例子来理解标志量的“标志”功效。

【例 28】 冒泡排序算法的改进。

问题分析： 冒泡排序算法的基本思想就是相邻数据比较，若逆序则交换，经过 $n-1$ 趟比较交换后，逐渐将小数据冒到数组的前部，大的数据则沉到数组的后部，从而完成排序工作。

现在假设原有的数据本来就是从小到大有序的，则原有算法仍要做 $n-1$ 趟比较操作，事实上一趟比较下来，若发现没有进行过交换，就说明数据已经全部有序，无须进行其后的比较操作了。

当然数据原本有序的概率并不高，但经过少于 $n-1$ 趟比较交换操作后，数据就已经有

序的概率却非常高。因此,为提高效率可以对冒泡排序算法进行改进,当发现某趟没有交换后就停止下一趟的比较操作。

人类可以用眼睛"宏观"地发现一组数据是否有序的状态;但算法只能通过逐一比较才能明确数据是否处于有序状态,这用逻辑表达式是不可能实现的。这时就考虑用标志量来记录每趟交换数据的情况,如 flag=0 表示没有进行过交换,一旦有数据进行交换则置 flag 为 1,表示已进行过交换。当一趟比较交换完成后,若 flag 为 0,则无须进行下一趟操作,若 flag 为 1,继续进行下一趟操作。

改进后的算法如下:

```
main( )
{ int i,j,t,n,a[100],flag;
  print("input data number(<100): ");
  input(n);
  print("input data: ");
  for(i = 0; i<n; i = i + 1)
      input(a[i]);
  flag = 1;
  for(i = 1; i<= n - 1 and flag = 1; i = i + 1)
        {flag = 0;
         for(j = n - 1; j>= i; j = j - 1)
              if(a[j]<a[j - 1])
                { t = a[j];
                  a[j] = a[j - 1];
                  a[j - 1] = t;
                  flag = 1; }
         }
  for(i = 0; i<n; i = i + 1)
         print(a[i]);
}
```

算法说明:

(1) 排序前"for(i=1; i<=n-1 and flag=1; i=i+1)"for 循环之前的 flag=1; 是为了保证循环的开始。

(2) 内层循环外的 flag=0; 是假设这趟比较中没有交换,一旦发生交换操作在内层循环中就置 flag=1;,标志将继续进行下一趟操作。

【提示】 请结合这个例题思考在什么情况下需要使用标志量。

【例 29】 编程判定从键盘输入 n 个数据互不相等。

算法设计: 在一维数组应用中,也有关于不重复使用数据的讨论,但只是针对指定的有一定规律的数据而进行的。这里要判定 n 个数,是无任何限定的数据。

若用逻辑表达式表示需要:

$$(n-1)+(n-2)+(n-3)+\cdots+1$$

(1 号与 2～n 号不同)(2 号与 3～n 号不同)(3 号与 4～n 号不同) ……

共 $n\times(n-1)/2$ 个关系表达式。当 $n=2$ 时有 1 个关系表达式;当 $n=5$ 则有 10 个关系表达式,虽然表达式的书写具有一定的规律性,可以自动生成字符串表达式,但算法的运行效率却得不到保证。下面,通过引入标志量记录数据是否有重复的情况,避免了复杂的逻辑表

达式。

算法如下：

```
main( )
 {int a[100],i,j,t,n;
  input(n);
  for (i = 1; i <= n; i = i + 1)
    input(a[i]);
  t = 1;
  for (i = 1; i <= n - 1; i = i + 1)
    for (j = i + 1; j <= n; j = j + 1)
      if (a[i] = a[j])
       {t = 0;
        break; }
  if (t = 1)
    print("Non repeat");
  else
    print("repeat");
}
```

算法说明：算法中通过二重循环，交叉比较所有数据，用标志变量 $t=0$ 标识已有重复数据。若循环结束，t 仍为 1，说明数据没有重复，互不相同。

【例 30】 输入 3 个数值，判断以它们为边长是否能构成三角形。如能构成，则判断是否属于特殊三角形，并判断是哪种（等边、等腰或直角）。

问题分析：这个题目表面看起来非常简单，但要做到合理输出却并不容易。这里先讨论一下可能的输出情况。

(1) 不构成三角形。

(2) 构成等边三角形。

(3) 构成等腰三角形。

(4) 构成直角三角形。

(5) 构成一般三角形。

其中情况(3)、(4)可能同时输出，而其他几种若同时输出就不合理了。

情况(1)与其他情况互斥容易分支。

情况(5)是在三角形不属于(2)、(3)、(4)3 种情况时的输出。

关键是(4)与(2)、(3)不是简单的互斥关系，所以仅用多分支 if 或嵌套 if 就无法进行合理输出。下面通过标志量实现合理的输出。

算法设计：算法中需要避免情况(5)与情况(2)、(3)、(4)之一同时输出。设置一标志变量 flag，当数据能构成三角形时就置 flag＝0 表示情况(5)，一旦测试出数据属于情况(2)、(3)、(4)中的一种情况时，就置 flag＝1 表示构成了特殊三角形，最后就不必输出“构成一般三角形”了；若 flag 最后仍保持 0，则输出“构成一般三角形”。

算法如下：

```
main( )
  { int a,b,c,flag;
  print("Input 3 number: ");
```

```
    input(a,b,c);
    if(a>=b+c or b>=a+c or c>=a+b)
        print("don't form a triangle");
    else
        {flag=0;
        if (a*a=b*b+c*c or b*b=a*a+c*c or c*c=a*a+b*b)
          {print("form a right-angle triangle");
            flag=1; }
        if(a=b and b=c)
          {print("form a equilateral triangle");
            flag=1; }
        else if(a=b or b=c or c=a)
              {print("form a equal haunch triangle");
               flag=1; }
        if(flag=0)
            print("form a triangle");
        }
}
```

算法说明：从算法中可以看出，分析中所讨论的不易分支表现在后几个 if 语句之间不是简单的“否则”关系。所以通过标志量来“标志”是否是特殊三角形，算法得到了合理输出。

下面再看一个例子。

【例 31】 编写算法，求任意 3 个数的最小公倍数。

问题分析：看完题目，一定有读者回忆起，小学时用短除法求 3 个整数的最小公倍数的方法，下面就用算法来模拟这个过程。

算法设计：

(1) 用短除法求 3 个已知数的最小公倍数的过程就是求它们的因数之积，这个因数可能是 3 个数共有的、两个数共有或一个数独有的 3 种情况。

(2) 在手工完成这个问题时，大脑可以判断 3 个数含有哪些因数以及属于哪种情况。用算法实现就只能利用尝试法了。尝试的范围应该是 2、3 个数中最大数之间的因数。

无论因数属于以下 3 种情况之一，都只算作一个因数，累乘一次。

① 若某个数是 3 个数的共有的因数，如 2 是 2,14,6 中所有数的因数；

② 若某个数是其中两个数的因数，如 2 是 2,5,6 中两个数的因数；

③ 若某个数是其中某一个数的因数，如 2 是 2,5,9 中一个数的因数。

以上 3 种情况例子中，因数 2 都只累乘一次。

(3) 再看例子 2,4,8 中 2 是所有数的因数，为避免因数重复计算，一定要用 2 整除这 3 个数得到 1,2,4。注意到 2 仍是(1,2,4)的因数，所以在尝试某数是否是 3 个数的因数时，不是用条件语句 if，而是要用循环语句 while，以保证将 3 个数中所含的某个因数能全部被找出，直到 3 个数都不含这个数做因数时循环结束。

(4) 由于某数 i 是已知 3 个数的因数有多种情况，以上讨论了 3 大类，后两类又能细分出更多小的类别。如是其中两个数共有的因数时，可能是第一、三个数的因数，或是第一、二个数的因数，或是第二、二个数的因数。总之，很难用一个简单的逻辑表达式来表示各种复杂的情况。

不过，借助 3.3.1 节“算术运算的妙用”中介绍的方法，用表达式：

```
k = (x1 mod i = 0) + (x2 mod i = 0) + (x3 mod i = 0)
```

的值，可以区分某数 i 是否为已知 3 个数的因数，$k=0$ 表示 i 不是 3 个数的因数，$k>0$ 表示 i 是 3 个数的因数。

为避免因数重复计算，每次都需要除掉 3 个整数中已找到的因数（即用因数去除含有它的整数）。而以上逻辑表达式无法识别 i 具体是哪一个数的因数，要对哪个数进行整除 i 的运算。下面采用标致量的方法来解决这里的问题。

算法如下：

```
max(int x,int y,int z)
{ if(x>y and x>z) return(x);
  else if (y>x and y>z)return(y);
        else return(z);
}
main( )
{   int x1,x2,x3,t=1,i,flag,x0;
   print("Input 3 number: ");
   input(x1,x2,x3);
   x0=max(x1,x2,x3);
   for (i=2; i<=x0; i=i+1)
    {flag=1;
     while(flag=1)
       { flag=0;
        if (x1 mod i=0)
          {x1=x1/i;
          flag=1; }
        if(x2 mod i=0)
          {x2=x2/i;
          flag=1; }
        if(x3 mod i=0)
          {x3=x3/i;
          flag=1; }
        if (flag=1)
          t=t*i;
       }//while 结束符
     x0=max(x1,x2,x3);
     }//for 结束符
     print("The result is ",t);
}
```

算法说明：在 while 循环体外将 flag 置为 1，是为了能进入循环。一进入循环马上将其置为 0，表示假设 i 不是 3 个数的因数，以下用 3 个条件语句测试：发现 i 是某个数的因数，则用因数去除对应整数，并将 flag 置为 1，表示 i 是某个数的因数；循环体最后测试 flag 的值，若为 1 则累乘 i 因数；否则，i 不是任意一个数的因数。

为了提高运行效率，需要进一步找出除掉因数后 3 个数的最大值，以决定是否继续进行 for 循环。

【提示】 其实求最大公约数的方法很多，这里介绍的也不是最好的方法，只是想启发大家一个简单的道理：不要认为算法很神秘，“解决问题的方法就在你的大脑里”，只是需要将

其转化为“机械”操作的算法而已。

3.3.3 信息数字化

学到这里,已经处理了许多数值计算方面的问题。计算机算法还能帮助做什么?从学到的计算机基础知识知道,计算机能存储和处理各种多媒体信息,如图形、图像、声音等,当然前提条件是将这些信息进行数字化。本书已声明不研究专业性强的算法,所以对多媒体信息的数字化及其压缩存储、处理等相关的算法,请读者阅读别的参考资料。下面就一些表面上看是非数值的问题,但经过数字化后,就可方便进行算法设计的问题做简单介绍。

【例 32】 警察局抓了 a,b,c,d 4 名偷窃嫌疑犯,其中只有 1 人是小偷。审问中,

a 说:我不是小偷。

b 说:c 是小偷。

c 说:小偷肯定是 d。

d 说:c 在冤枉人。

现在已经知道 4 人中 3 人说的是真话,1 人说的是假话,问到底谁是小偷?

问题分析:将 a,b,c,d 4 人进行编号,号码分别为 1,2,3,4,则问题可用枚举尝试法来解决。

算法设计:用变量 x 存放小偷的编号,则 x 的取值范围从 1 取到 4,就假设了他们中的某人是小偷的所有情况。4 人所说的话就可以分别写成如下所示。

a 说的话:x <> 1 或 not(x=1)。

b 说的话:x=3。

c 说的话:x=4。

d 说的话:x <> 4 或 not(x=4)。

再利用 3.3.1 节“算术运算的妙用”中学习到的技巧,在 x 的枚举过程中,当这 4 个逻辑式的值相加等于 3 时,即可以表示“4 人中 3 人说的是真话,1 人说的是假话”。

算法如下:

```
main( )
{ int x;
    for (x = 1; x < = 4; x = x + 1)
  if  ((x <> 1) + (x = 3) + (x = 4) + (x <> 4) = 3)
    print(chr(64 + x),"is a thief .");
}
```

运行结果:

```
c is a thief .
```

算法说明:为了算法的方便运行,对人名进行了数字化,但结果的形式还要符合题目的描述,所以输出时,用程序设计语言提供的库函数 chr( )将数字转化为对应的字母。这个算法可以方便地改写成 PASCAL 或 BASIC 算法,就 C 语言而言算法还可直接写成如下形式。

```
main( )
{int x;
 for (x = 'a'; x <= 'd'; x = x + 1)
     if (((x!= 'a') + (x = 'c') + (x = 'd') + (x!= 'd')) = 3)
              print(x," is a thief .");
}
```

【提示】 想必大家在学习C语言时，并不理解关系表达式为真时为什么值为1，这里体会到了吧！所以还是那句话："不要认为有用时才学习，有时可能你感受不到例题(或程序语言机制)的应用意义，也要认真学习和体会才能有收获。"

【例33】 3位老师对某次数学竞赛进行了预测。他们的预测如下。

甲说：学生a得第一名，学生b得第三名。

乙说：学生c得第一名，学生d得第四名。

丙说：学生d得第二名，学生a得第三名。

竞赛结果表明，他们都说对了一半，说错了一半，并且无并列名次，试编程输出a，b，c，d各自的名次。

问题分析：用数1，2，3，4分别代表学生a，b，c，d获得的名次，问题就可以利用三重循环把所有的情况枚举出来。

算法设计：

(1) 用a，b，c，d代表4个同学，其存储的值代表他们的名次。

设置第一层计数循环a的范围从1到4；

设置第二层计数循环b的范围从1到4；

设置内计数循环c的范围从1到4；

由于无并列名次，名次的和为1＋2＋3＋4＝10，由此可计算出d的名次值为10－a－b－c。

(2) 问题的已知内容，可以表示成以下几个条件式：

```
① (a = 1) + (b = 3) = 1
② (c = 1) + (d = 4) = 1
③ (d = 2) + (a = 3) = 1
```

若3个条件均满足，则输出结果，若不满足，继续循环搜索，直至循环正常结束。

算法如下：

```
main( )
{int  a,b,c,d;
 for(a = 1; a <= 4; a = a + 1)
  for(b = 1; b <= 4; b = b + 1)
   if (a <> b)
    for(c = 1; c <= 4; c = c + 1)
      if (c <> a and c <> b)
         {d = 10 - a - b - c;
          if (d <> a and d <> b and d <> c)
                 if (((a = 1) + (b = 3)) = 1 and ((c = 1) + (d = 4)) = 1 and ((d = 2) + (a = 3)) = 1)
                          print("a,b,c,d = ",a,b,c,d);
         }
}
```

运行结果：

```
a=4,b=3,c=1,d=2
```

【例 34】 填写运算符。

输入任意 5 个数 $x1,x2,x3,x4,x5$ 每两个相邻数之间填上一个运算符。在填入 4 个运算符“+、-、*、/”后，使得表达式值为一个指定值 y（y 由键盘输入）。求出所有满足条件的表达式。

问题分析：看了题目之后，发现难以找到好的解法，不妨在每两个相邻数之间尝试所有的运算符，从中找出问题的答案。

算法设计：

(1) 枚举尝试法解题：先填 4 个“+”。检验条件表达式 $x1+x2+x3+x4+x5=y$，如果不成立，则将第四个运算符改为“-”，以后改“*”、改“/”。轮完一遍，把第三个运算符改为“-”，第四个运算符按“+、-、*、/”顺序再轮一遍……如此下去，直至第一个运算符，由“+”至“/”轮完为止。

每两个相邻数之间 4 个运算符均按“+、-、*、/”尝试一遍，则要组织四重循环。

(2) 若当前运算符是“/”，则注意考虑运算符右端的数必须非零，因为零不可以作为除数。

(3) 现在接着考虑“+、-、*、/”应如何表示，才能方便算法对表达式的求值？

为了便于循环，在算法中，把“+、-、*、/”数字化作 1,2,3,4。5 个数据间需 4 个运算符，这次不用 4 个普通变量，而是用一个有 4 个元素数组 $i[1]\cdots i[4]$ 来代表它们，道理在算法说明中解释。例如 $i[3]=4$ 表示第三个运算符为“/”。

(4) 如何在运算时保证“先乘除/后加减”的优先顺序？

模拟计算：为了解决运算的优先级问题，设置如下变量。

f——符号标志。减法运算时，置 f=-1，否则 f=1；

q——若当前运算符为+(-)时，q 存储运算符的左项值；若当前运算符为 *(/)时，q 存储两数乘(除)后结果；

p——累加器。每填一个+(-)算符，就累加上一次运算的结果 f*q，表达式为 p=p+f*q。

在每次尝试填入 4 个算符前，设置 f=1、q=第 1 项的值，p=0，然后由左至右计算表达式的值。当分析至第四个算符时，f 是第四次运算符的符号标志，q 是第五项值，p 是前 3 次运算的累加值。此时只要检验 p+f*q=y 是否成立。若成立，则为一个解方案；否则重新循环搜索新的一组算符。

算法如下：

```
main( )
{int j,k,f,i[5],total;
 float n[6],p,q;
 char c[5]={",'+','-','*','/'};
 print("input five number");
 for(j=1; j<=5; j=j+1)
    input(n[j]);
 print(" input result: ");
```

```
input(n[0]);
total = 0;
for (i[1] = 1; i[1]< = 4; i[1] = i[1] + 1)
 if ((i[1]< 4) or (n[2]<> 0))
  for (i[2] = 1; i[2]< = 4; i[2] = i[2] + 1)
   if((i[2]< 4) or (n[3]<> 0))
    for (i[3] = 1; i[3]< = 4; i[3] = i[3] + 1)
     if ((i[3]< 4) or (n[4]<> 0))
       for (i[4] = 1; i[4]< = 4; i[4] = i[4] + 1)
        if((i[4]< 4) or (n[5]<> 0))
          {p = 0; q = n[1]; f = 1;
           for (k = 1; k < = 4; k = k + 1)
             switch (i[k])
              {case 1:     p = p + f * q; f = 1; q = n[k + 1]; break;
               case 2:     p = p + f * q; f = - 1; q = n[k + 1]; break;
               case 3:     q = q * n[k + 1]; break;
               case 4:     q = q/n[k + 1]; }
           if (p + f * q = n[0])
              {total = total + 1;
               print ("total",total,": ");
               for (k = 1; k < = 4; k = k + 1)
                   print (n[k],c[i[k]]);
               print (n[5]," = ",n[0]);
               }
            }
  if (total = 0)
     print("Non solution");
}
```

算法说明：

(1) 算法中 4 个 for 循环后的 4 个 if 语句，是为了保证不进行除数为 0 的运算。

(2) 在枚举 4 个运算符时，用一个数组的 4 个元素 $i[1]$，$i[2]$，$i[3]$，$i[4]$代表 4 个运算符，相信大部分读者已清楚这样做的道理。因为，若用 4 个普通变量代表 4 个运算符，则运算过程需使用 4 个 switch 语句，每个 switch 语句中 4 个 case 语句；而用一个数组存储 4 个运算符，算法就可以通过 4 次循环完成运算过程了。这又一次运用了数组构造循环“不变式”的技巧。

【提示】 请读者仔细体会此算法中应用了前几节的哪些技巧？

【例 35】 有 10 箱产品，每箱有 1000 件，正品每件 100 克。其中有几箱是次品，每件次品比正品轻 10 克，问能否用秤只称一次，就找出哪几箱是次品。

问题分析： 从表面上看，问题中的已知条件都是数据，无须数字化。但事实上要较好地解决此问题，不但需要信息数字化方法，而且需要一点数字化的技巧。

(1) 先把问题难度降低，假设只有一箱是次品，这个问题容易找到解决方法：先将箱子编码 1，2，…，10。再从 1 号箱取 1 件产品，2 号箱取 2 件产品，3 号箱取 3 件产品……10 号箱取 10 件产品。称它们的重量，若比标准重量轻 10 克则 1 号箱为次品；比标准重量轻 20 克，则 2 号箱为次品；比标准重量轻 30 克，则 3 号箱为次品……比标准重量轻 100 克，则 10 号箱为次品。

设取出产品重量为 w,则次品的箱号为((1+2+3+…+10)×100−w)/10。

(2) 注意如若不止一箱次品时,以上的方法就行不通了,例如:若称出的重量比标准重量轻 30 克时,可能 1,2 号两箱是次品,也可能 3 号一箱是次品。

不过以上方法的基本思想还是可以利用的,即根据取出每箱产品数量的差异和取出产品的总重量与标准重量误差,来分析推算次品的箱号。

数字化过程:同样,先将箱子编码 1,2,…,10。用枚举方法分析问题,由小到大讨论。

从 1 号箱取 1 件产品,若最后总重量比标准重量轻 10 克,则 1 号箱为次品。

从 2 号箱取 2 件产品,若最后总重量比标准重量轻 20 克,则 2 号箱为次品。

从 3 号箱取 3 件产品,若最后总重量比标准重量轻 30 克,无法识别哪些箱是次品。但从 3 号箱取 4 件产品,若最后总重量比标准重量轻 30 克,则 1 号、2 号箱为次品;轻 40 克,肯定 3 号箱为次品。

再看 4 号箱,

① 取 5 件产品,若最后总重量比标准重量轻 50 克,无法识别哪些是次品,可能是 1 号箱、3 号箱(分别取 1 件、4 件)或 5 号箱为次品。

② 取 6 件产品,若最后总重量比标准重量轻 60 克,无法识别哪些是次品,可能是 2 号箱、3 号箱(分别取 2 件、4 件)或 5 号箱为次品。

③ 取 7 件产品,若最后总重量比标准重量轻 70 克,也无法识别哪些是次品,可能是 1 号箱、2 号箱、3 号箱(分别取 1 件、2 件、4 件)或 5 号箱为次品。

④ 取 8 件产品,则最后总重量比标准重量轻 80 克,则可以肯定 4 号箱为次品。

无须继续枚举就可看出,1,2,…,10 号箱取产品的件数分别为 2^0,2^1,2^2,2^3,2^4,2^5,2^6,2^7,2^8,2^9,即 1,2,4,8,16,32,64,128,256,512 件。

[注:当箱子数量较大时每箱取出的产品数量呈指数增长,数目太大时,有可能箱子中没有那么多产品,所以此算法并不现实。实际当中,若允许多次称量时可先采用开始介绍的只有一箱次品的方法编号,取产品,最后称量。分析出可能有次品的箱子,再在其中进一步进行编号、取产品、称量工作。这里就不深入讨论了。]

(3) 根据以上方法,取出产品称量。

轻 10 克 1 号箱为次品。

轻 20 克 2 号箱为次品。

轻 30 克 1,2 号箱为次品。

轻 40 克 3 号箱为次品。

轻 50 克 1,3 号箱为次品。

轻 60 克 2,3 号箱为次品。

轻 70 克 1,2,3 号箱为次品。

轻 80 克 4 号箱为次品。

轻 90 克 1,4 号箱为次品。

……

算法设计:怎样用算式或算法来识别误差与次品箱号的关系呢?首先计算标准总重量,存储在变量 $w1$ 中。输入称取的实际总重量,存储在变量 $w2$ 中。然后计算出比标准重量轻多少,仍存储在变量 $w1$ 中。

当 $w/10=2^k$ 时，则 $k+1$ 号箱为唯一的一箱次品。

当 $w/10>2^k$ 时，k 取最大时记为 $k1$，$k1+1$ 号箱为次品箱。

继续讨论 $w/10-2^{k1}>2^k$ 时，k 取最大时记为 $k2$，$k2+1$ 号箱为次品箱。

……

直到取等号 $w/10-2^{ki-1}=2^{ki}$ 时，$ki+1$ 号箱为次品箱，此时不再有次品。

【提示】 从具体到抽象的方法虽然效率低但很实用。

算法 1 如下：

```
main( )
{ int i,k,n,t;
  long w1,w2;
  print("Input the number of boxes: ");
  input(n);
  t = 1;
  for (i = 1; i <= n; i = i + 1)
    {print(i,"box take",t,"units.");
     w1 = w1 + t;
     t = t * 2;
     }
  w1 = w1 * 100;
  print("normal weight ",w1);
  print(" Input reality weight");
  input(w2);
  w1 = (w1 - w2)/10;
  while (w1 <> 0)
        {k = 0;
        t = 1;
        while (w1 - t > 0)
              {t = t * 2;
              k = k + 1; }
              print(k,"box is bad ");
              w1 = w1 - t/2;
       }
}
```

算法说明：算法中的最后一个内层 while 循环及其后语句，是用于计算最接近当前重量 $w1$ 的 2^k，k 号箱就是次品箱。这个过程可以通过使用程序设计语言提供的库函数来完成吗？回答是肯定的。

算法 2 如下：

```
main( )
{ int i,k,n,t;
  long w1,w2;
  print("Input the number of boxes: ");
input(n);
t = 1;
for (i = 1; i <= n; i = i + 1)
   {print(i,"box take",t,"letter.");
    w1 = w1 + t;
```

```
    t = t * 2;
    }
w1 = w1 * 100;
print("\n normal weight ",w1);
print("Input reality weight");
input(w2);
w1 = (w1 - w2)/10;
while(w1 <> 0)
   {k = log(w1)/log(2);        // 以 2 为底取对数
    print(k + 1,"is bad");
    w1 = w1 - pow(2,k);        // pow(2,k)的功能是求 2 的 k 次方
    }
}
```

算法分析：算法 2 利用数学知识节省了设计时间，比算法 1 简单，好理解；但算法 2 由于要进行函数调用，效率比算法 1 稍低。

【例 36】 编写算法对输入的一个整数，判断它能否被 3,5,7 整除，并输出以下信息之一：

能同时被 3,5,7 整除；

能被其中两个数(要指出哪两个)整除；

能被其中一个数(要指出哪一个)整除；

不能被 3,5,7 任一个整除。

算法设计：要用逻辑表达式表示第一、四两种情况很简单，但要准确表示第二、三两种情况就比较复杂了。若用 3.3.1 节“算术运算的妙用”中“算术运算”的技巧就能避免复杂的逻辑表达式了，如下面的算法 1。但算法 1 只能输出输入能被 3,5,7 中几个数整除的数据，而不能指出输入数据具体能被 3,5,7 中哪几个数整除。而结合“信息数字化”技巧能较好地解决此问题，如下面的算法 2。

算法 1 如下：

```
main( )
{long n;
 int k;
 print("Please enter a number: ");
 input(n);
 k = (n mod 3 = 0) + (n mod 5 = 0) + (n mod 7 = 0)
 switch (k)
   {case 3: print("All"); break;
    case 2: print("two"); break;
    case 1: print("one"); break;
    case 0: print("none"); break; }
}
```

算法 2 如下：

```
main( )
{long n;
 int k;
 print("Please enter a number: ");
```

```
input(n);
k = (n mod 3 = 0) + (n mod 5 = 0) * 2 + (n mod 7 = 0) * 4
switch (k)
   {  case  7: print("All"); break;
      case  6: print("5 and 7"); break;
      case  5: print("3 and 7"); break;
      case  4: print("7"); break;
      case  3: print("3 and 5"); break;
      case  2: print("5"); break;
      case  1: print("3"); break;
      case  0: print("none"); break; }
}
```

算法说明：算法中 k 表示整除的情况值。算法 1 中，k 的范围是 0～3 可以表示 4 种情况，而算法 2 中，为 k 赋值的表达式是“(n mod 3＝0)＋(n mod 5＝0) * 2＋(n mod 7＝0) * 4”，k 的范围是 0～7，可以表示 8 种情况。

算法 2 中，既运用算术运算的技巧，又较好地运用了“数字化”标识信息的技巧。所以变量 k 的信息含量更高，从而输出的结果更具体。

【提示】 相信由上一个例题，大家能理解表达式

```
k = (n mod 3 = 0) + (n mod 5 = 0) * 2 + (n mod 7 = 0) * 4
```

中，3 个条件表达式的系数为什么分别是 1，2，4。正是这 3 个系数使 3 个条件具有了不同的信息量，标志了 8 种不同的情况。

3.4　优化算法的数学模型

本节介绍一些数学模型在算法设计中的应用。通过选择恰当的数学模型，可以使算法的效率更高、占用空间更合理或使算法更简洁。

1. 认识数学模型和数学建模


数学模型

说到数学建模，有好多人感到不知所云，下面看一个很简单的例子。

已知有 5 个数，求前 4 个数与第 5 个数分别相乘后的最大数。给出两个算法分别如下：

```
max1(int a,int b,int c,int d,int e)
  {int x;
  a = a * e;
  b = b * e;
  c = c * e;
  d = d * e;
  if (a>b)
      x = a;
  else
      x = b;
  if (c>x)
      x = c;
  if (d>x)
```

```
max2(int a,int b,int c,int d,int e)
{int x;
if (a>b)
   x = a;
else
   x = b;
if (c>x)
   x = c;
if (d>x)
   x = d;
x = x * e;
print(x);
}
```

```
    x = d;
print(x);
}
```

算法分析：表 3-3 给出了两个算法基本操作的次数。

表 3-3 算法操作次数比较

算　法	乘　法	赋　值	条件判断
max1	4	7	3
max2	1	4	3

两个算法的思路是一样的。4 个数据取最大值，先选出前两个数据中较大的存储在变量 x 中，然后和后两个数据进行比较，发现大的数据就将其存储在 x 中，则最后 x 存储的就是 4 个数据中最大的数据。效率之所以不同是由于两个算法基于的数学模型是不同的。

算法 max1 可以说没有用数学知识，按题目描述先求积，再求解 4 个积的最大值。而算法 max2，利用数学上的一个简单常识，两个数乘以相同的数，原来大数的积就大，原来小数的积就小。基于这样的数学模型，算法 max2 先求出 4 个数的最大值，然后再求积。由于使用了不同的数学模型，一个算法先求积再求最大值，另一个算法先求最大值再求积，导致后一个算法的效率明显要高于前一个算法。

由上面的例子可以看出，要想对问题建立数学模型，必须掌握必要的数学知识。所谓“掌握”，在本节中更多的是指“巩固和复习”已学过的知识和术语，如：多数学生明白质数的概念，但说到素数就不知所云，其实它们是同一个概念；对于因数、质因数、最大公约数和最小公倍数概念，都是小学的知识，可以说无人不知无人不晓，但用这些概念去建立模型的能力就比较匮乏了。本节先介绍一些数学建模基本概念，然后在小节中通过一些例子了解数学建模在算法设计的重要性。

什么是数学模型？数学模型(mathematical model)是利用数学语言(符号、式子与图像)模拟现实的模型。把现实模型抽象、简化为某种数学结构是数学模型的基本特征。它或者能解释特定现象的现实状态，或者能预测到对象的未来状况，或者能提供处理对象的最优决策或控制。

数学建模就是把现实世界中的实际问题加以提炼，构造出数学模型，求出模型的解，验证模型的合理性，并用该数学模型所提供的解答来解释现实问题，把数学知识的这一应用过程称为数学建模。流程框图如图 3-3 所示。

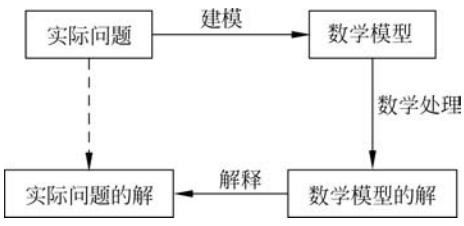

图 3-3　流程框图

数学建模是一个专门的学科，数学建模中还有很多复杂的数学知识，如模糊数学、线性规划、非线性规划、随机规划、概率与数理统计、最优化理论、图论和组合数学，这些知识都有专门书籍供大家深入学习。

2. 数学建模的基本方法

通用而简单的数学建模方法，主要是归纳法。从分析问题的几种简单的、特殊的情况中，发现一般规律或做出某种猜想，从而找到解决问题的途径。这种研究问题方法叫作归纳法。归纳法是从简单到复杂，由个别到一般的一种研究方法，也就是俗称的“找规律”。

在分析问题时，一般是从问题的初始情况开始，人工地枚举问题的发展情况，从而找到到达目标的方法。但也有的问题是要倒过来进行的，在固定问题规模的情况下，倒着考虑最后一步是怎么样得到的，从而发现解决问题的方法。

在算法设计中建立数学模型，还需要具备一定的数学、物理等领域的知识。下面就看一些与数学建模相关的算法设计问题。

3.4.1　杨辉三角形的应用

有些数学知识，是大家所熟知的，但知识间彼此的关系却不太明了。例如，n 次二项式和杨辉三角形虽然都是熟悉的概念，若不知道二者关系，可能会设计出效率较低的算法。看下面的例子。

【例 37】 求 n 次二项式各项的系数，已知二项式的展开式为：

$$(a+b)^n=C_n^0a^n+C_n^1a^{n-1}b+C_n^2a^{n-2}b^2+\cdots+C_n^nb^n$$

模型建立：若只利用组合数学的知识，直接建模。

$$C_n^k=\frac{n!}{k!\ (n-k)!}\qquad k=0,1,2,3,\cdots,n$$

用这个公式去计算，$n+1$ 个系数，将需要进行很多次累乘，当 n 比较大时，算法将因时间效率太低，而不被接受。即使考虑到了前后系数之间的数值关系。

$$C_n^{k+1}=C_n^k\cdot\frac{n-k}{k+1}$$

算法中也会有大量的乘法和除法运算，效率也是比较低的。

有一个数学常识：各阶多项式的系数，呈杨辉三角形的规律，具体如下。

$(a+b)^0$　　1

$(a+b)^1$　　1　　1

$(a+b)^2$　　1　　2　　1

$(a+b)^3$　　1　　3　　3　　1

$(a+b)^4$　　1　　4　　6　　4　　1

$(a+b)^5$　　…

以这个知识为基础，求 n 次二项式的系数的数学模型就是求 n 阶杨辉三角形，这时算法中只须做一些简单的加法运算，效率就很高了。具体算法设计如下。

算法设计要点：这些系数间有明显的规律，即除了首尾两项系数为 1 外，当 $n>1$ 时，$(a+b)^n$ 的中间各项系数是 $(a+b)^{n-1}$ 的相应两项系数之和，如果把 $(a+b)^n$ 的 $n+1$ 的系数列为数组 c，则除了 $c(1)$，$c(n+1)$ 恒为 1 外，设 $(a+b)^n$ 的系数为 $c(i)$，$(a+b)^{n-1}$ 的系数设为 $c'(i)$。则有如下代码。

```
c(i) = c'(i) + c'(i - 1)
```

而当 $n=1$ 时,只有两个系数 $c(1)$和 $c(2)$(值都为 1)。不难看出,对任何 n,$(a+b)^n$ 的二项式系数可由 $(a+b)^{n-1}$ 的系数求得,直到 $n=1$ 时,两个系数有确定值,故可写成递归子算法。

算法如下:

```
coeff(int a[ ],int n)
{if(n = 1)
  {a[1] = 1;
   a[2] = 1; }
 else
    {coeff(a,n - 1)
     a[n + 1] = 1
     for (i = n; i >= 2; i = i - 1)
      a[i] = a[i] + a[i - 1];
     a[1] = 1;
    }
}
main( )
{int a[100],i,n;
 input(n);
 for(i = 1; i <= n; i = i + 1)
    input(a[i]);
 coeff(a,n);
 for(i = 1; i <= n; i = i + 1)
 print(a[i]);
}
```

算法分析:算法的主要操作为加法,复杂度为 $O(n^2)$。

【提示】 用非递归方法,二维数组也可以完成该算法,请读者自己尝试。

3.4.2 最大公约数的应用

建立数学模型需要具备一定的抽象能力,抽象能力不仅仅要靠天生的感悟,还需要从具体的实例中,归纳总结出问题规律性的、本质性的东西。也就是说认识问题应该是从具体到抽象,不可能有空中楼阁。看下面的例子。

【例 38】 数组中有 n 个数据,要将它们顺序循环向后移 k 位,即前面的元素向后移 k 位,后面的元素则循环向前移 k 位,例:0,1,2,3,4 循环移 3 位后为 2,3,4,0,1。考虑到 n 会很大,不允许用 $2\times n$ 以上个空间来完成此题。

问题分析 1:若题目中没有关于存储空间的限制,可以方便地开辟两个一维数组,一个存储原始数据,另一个存储移动后的数据。由于在“循环”后移的过程中,开始的元素确实是从前向后移,但数组中后 k 个元素实际是在从后向前移,由数学知识,通过求余运算就可以解决“循环”后移的问题。

算法 1 如下:

```
main( )
{int a[100],b[100],i,n,k;
 input(n,k);
```

```
 for(i = 0; i < n; i = i + 1)
    input(a[i]);
 for(i = 0; i < n; i = i + 1)
    b[(k + i) mod n] = a[i];
 for(i = 0; i < n; i = i + 1)
    print(b[i]);
}
```

这个算法的时间效率是 $O(n)$，空间效率也是 $O(n)$。

问题分析 2：在算法有空间限制的情况下，一种简单的方法是：

(1) 将最后一个存储空间的数据，存储在临时存储空间中；

(2) 其余数据逐个向后移动一位。

这样操作一次所有数据移动一位，共操作 k 次就能完成问题的要求。

算法 2 如下：

```
main( )
{   int a[100],b[100],i,j,n,k,temp;
    input(n,k);
    for(i = 0; i < n; i = i + 1)
       input(a[i]);
    for(i = 0; i < k; i = i + 1)
       {temp = a[n - 1];
        for(j = n - 1; j > 0; j = j - 1)
           a[j] = a[j - 1];
        a[0] = temp;
       }
    for(i = 0; i < n; i = i + 1)
       print(b[i]);
}
```

若要考虑到有 $k > n$ 的可能性，这样的移动会出现重复操作，可以在输入数据后，执行 $k = k \bmod n$；操作，可以保证不出现重复移动的情况。这时算法的移动(赋值)次数为 $k * n$。当 n 较大时，算法的效率比较低。

问题分析 3：能不能只利用一个临时存储空间，把每一个数据一次移动到位呢？抽象地考虑这个问题有点难，用具体的数据来讨论。

(1) 一组循环移动的情况。

通过计算可以确定某个元素移动后的具体位置，如 $n = 5, k = 3$ 时，0,1,2,3,4 循环移 3 位后为 2,3,4,0,1。

可通过计算得出 0 移到 3 的位置，3 移到 1 的位置，1 移到 4 的位置，4 移到 2 的位置，2 移到 0 的位置；一组移动(0-3-1-4-2-0)正好将全部数据按要求进行了移动。这样只需要一个辅助变量，每个数据只需要一次移动就可完成整个移动过程。

如果算法就这样按一组移动去解决问题，就错了。因为还有其他情况。

(2) 多组循环移动的情况，再看下一个例子，当 $n = 6, k = 3$ 时，0,1,2,3,4,5 经移动的结果是 3,4,5,0,1,2。

0 移到 3 的位置，3 移到 0 的位置，并不像上一个例子，一组循环移动(0-3-0)没有将全

部数据移动到位。还需要(1-4-1,2-5-2)两组移动,共要进行 3 组循环移动(1-4,2-5,3-6)才能将全部数据操作完毕。

那么,循环移动的组数,与 k,n 是怎么样的关系呢?实例太少还不容易看出规律,下面再看一个例子,当 $n=6$,$k=2$ 时,0,1,2,3,4,5 经移动的结果是 4,5,0,1,2,3。

0 移到 2 的位置,2 移到 4 的位置,4 移到 0 的位置,一组移动(0-2-4-0)完成了 3 个数据的移动,接下来,还有一组 1-3-5-1。共进行两组循环移动,就能将全部数据移动完毕。

相信有这 3 个实例,就可以建立以下的数学模型了。

数学模型:由以上的分析和实例以及数学知识应该"感知",问题与 n 和 k 最大公约数有关,即循环移动的组数等于 n 与 k 的最大公约数。这就是利用数学知识建模的过程。"感知"是否正确可以通过数学方法证明(就像哥德巴赫猜想先有猜想的结论,再力求证明),或通过算法进行大量的数据验证。简单一些可采用后者。

算法设计:

(1) 编写函数,完成求 n,k 最大公约数 m 的功能。

(2) 进行 m 组循环移动。

(3) 每组移动进行 n/m 次。通过计算可以确定某个元素移动后的具体位置。在移动之前,用临时变量存储需要被覆盖的数据。

综上算法 3 如下:

```
ff(int a,int b)
{int i,t = 1;
  for (i = 2; i <= a and i <= b; i = i + 1)
    while (a mod i = 0 and b mod i = 0)
        {t = t * i;
         a = a/i;
         b = b/i; }
  return(t);
}
  main( )
  {int a[100],b0,b1,i,j,n,k,m,tt;
   print("input the number of data");
   input(n);
   print("input the distant of moving");
   input(k);
   for(i = 0; i < n; i = i + 1)
     input(a[i]);
   m = ff(n,k);
   for(j = 0; j < m; j = j + 1)
     {b0 = a[j];
      tt = j;
      for(i = 0; i < n/m; i = i + 1)
          {tt = (tt + k) mod n;
           b1 = a[tt];
           a[tt] = b0;
           b0 = b1; }
     }
   for(i = 0; i < n; i = i + 1)
```

```
        print(a[i]);
    }
```

算法分析：算法中，每组循环移动都是从前向后进行的，这样每次移动之前，都需要将后面的数据先存入辅助存储空间中，一次移动需要两次赋值，总体大约需要赋值 $2n$ 次。

能不能继续提高效率为只赋值 n 次呢？请考虑改进每组循环移动的方式为从后开始移动，以提高运行效率。例如：

$n=6$，$k=2$ 时，第一组循环移动 0-2-4，在算法 3 中是这样实现的：

```
a[0]=>b0,
a[2]=>b1,b0(a[0])=>a[2],b1=>b0;
a[4]=>b1,b0(a[2])=>a[4],b1=>b0;
a[0]=>b1,b0(a[4])=>a[0],b1=>b0;
```

改进后（和算法 2 类似）：

```
a[4]=>b,a[2]=>a[4],a[0]=>a[2],b=>a[0]
```

将每组最后一个数据元素存储在辅助存储空间，以后就可以安全地覆盖后面的数组元素了（注意覆盖顺序）。这样，一组循环移动只需要一次将数据存入辅助存储空间，其后一次移动只需要一次赋值，全部工作大约需要赋值 n 次就可完成。请读者尝试完成。

【提示】 有读者总是认为自己缺乏算法设计的灵感，从这个问题的数学模型建立过程，相信应该理解为什么说“勤奋是成功的基础”。

3.4.3　公倍数的应用

利用公倍数解决一些数学问题在小学阶段已学过。例如“某数分别除以 3，5，8 的余数均为 2，求满足条件的最小自然数”是一个同余问题，问题的解是：[3,5,8]的最小公倍数＋2，其中道理不难理解。对于余数不相同的问题，在小学的数学中没有介绍，但能通过余数和公倍数的知识，建立数学模型。看下面的例子。

【例 39】 编写算法完成下面给“余”猜数的游戏。

心里先想好一个 1～100 的整数 x，将它分别除以 3，5 和 7 并得到 3 个余数。把这 3 个余数输入计算机，计算机能马上猜出这个数。游戏过程如下：

```
Please think of a number between 1 and 100
Your number divided by 3 has a remainder of 1
Your number divided by 5 has a remainder of 0
Your number divided by 7 has a remainder of 5
let me think a moment…
Your number was 40
```

问题分析：一种简单的办法就是从 1～100 逐一去尝试问题的解。这里通过“找出余数与求解数之间的关系”，也就是建立问题数学模型来解决。

数学模型：先看基本的数学常识。

（1）当 $s=u+3\times v+3\times w$ 时，s 除以 3 的余数与 u 除以 3 的余数是一样的。

（2）对 $s=cu+3\times v+3\times w$，当 c 除以 3 余数为 1 的数时，s 除以 3 的余数与 u 除以 3 的余数也是一样的。证明如下：

c 除以 3 余数为 1,记 $c=3\times k+1$,则 $s=u+3\times k\times u+3\times v+3\times w$,由(1)的结论,上述结论正确。

为了好讲解,先给出问题的数学模型,再讲解模型建立的道理。记 a,b,c 分别为所猜数据 d 除以 3,5,7 后的余数,则 $d=70\times a+21\times b+15\times c$ 为问题的数学模型,其中 70 称作 a 的系数,21 称作 b 的系数,15 称作 c 的系数。下面来看一下建立模型的道理。

由以上数学常识,a、b、c 的系数必须满足:

(1) b、c 的系数能被 3 整除,且 a 的系数被 3 整除余 1,这样 d 除以 3 的余数与 a 相同。

(2) a、c 的系数能被 5 整除,且 b 的系数被 5 整除余 1,这样 d 除以 5 的余数与 b 相同。

(3) a、b 的系数能被 7 整除,且 c 的系数被 7 整除余 1,这样 d 除以 7 的余数与 c 相同。

由此可见:c 的系数是 3 和 5 的公倍数且被 7 整除余 1,正好是 15;

a 的系数是 7 和 5 的公倍数且被 3 整除余 1,最小只能是 70;

b 的系数是 7 和 3 的公倍数且被 5 整除余 1,正好是 21。

算法设计:用以上模型求解的数 d,可能比 100 大,这时只要减去 3,5,7 的最小公倍数就是问题的解了。

游戏算法如下:

```
main( )
{int a,b,c,d;
 print("please think of a number between 1 and 100.");
 print("your number divided by 3 has a remainder of");
 input(a);
 print("your number divided by 5 has a remainder of");
 input(b);
 print("your number divided by 7 has a remainder of");
 input(c);
 print("let me think a moment…");
 d = 70 * a + 21 * b + 15 * c;
 while (d>105)
     d = d - 105;
 print("your number was ",d);
}
```

【提示】 思考若 3 个除数和余数都是由用户给出,如何设计算法?

3.4.4 斐波那契数列的应用

斐波那契(Fibonacci Leonardo,约 1175—1250)是意大利著名数学家。他最重要的研究成果是在不定分析和数论方面,他的“斐波那契数列”成为世人热衷研究的问题。斐波那契数列有如下特点:

$a1$,$a2$ 已知

$a(n)=a(n-1)+a(n-2)$ $n\geqslant 3$

这个数列在许多问题中都会出现,如兔子繁殖问题、树枝问题、上楼方式问题、蜂房问题、声音问题、花瓣问题……看下面的例子。

【例 40】 楼梯上有 n 阶台阶，上楼时可以一步上 1 阶，也可以一步上 2 阶，编写算法计算共有多少种不同的上楼梯方法。

数学模型：此问题如果按照习惯，从前向后思考，也就是从第一阶开始，考虑怎么样走到第二阶、第三阶、第四阶……则很难找出问题的规律；而反过来先思考“到第 n 阶有哪几种情况?”，答案就简单了，只有“两种”情况。

（1）从第 $n-1$ 阶到第 n 阶；

（2）从第 $n-2$ 阶到第 n 阶。

记 n 阶台阶的走法数为 $f(n)$，则

$$f(n)=\begin{cases}1 & n=1\\ 2 & n=2\\ f(n-1)+f(n-2) & n>2\end{cases}$$

读者要通过此例题学会“反向分析法”的应用，其实这种方法和递归设计一样，就是找出大规模问题与小规模问题之间的关系，而这个关系正好符合斐波那契数列的规律。

算法设计：根据数学模型，算法可以用递归或循环完成，下面是问题的递归算法，非递归算法课后完成。

算法如下：

```
main( )
    {int n: ;
     print('n = ');
     input(n);
     print('f(',n,') = ',f(n));
    }
f(int n)
    {if (n = 1)
        return(1);
     if (n = 2)
        return(2);
     else
        return(f(n - 1) + f(n - 2));
    }
```

一个算法的数学模型非常重要。这个问题如果是用后面将要学习到的枚举或搜索等复杂的算法策略，反而会把问题搞得很复杂，且效率很低。

3.4.5　特征根求解递推方程

【例 41】 核反应堆中有 α 和 β 两种粒子，每秒钟内一个 α 粒子变化为 3 个 β 粒子，而一个 β 粒子可以变化为 1 个 α 粒子和 2 个 β 粒子。若在 $t=0$ 时刻，反应堆中只有一个 α 粒子，求在 t 时刻的反应堆中 α 粒子和 β 粒子数。

数学模型 1：本题中共涉及两个变量，设在 i 时刻 α 粒子数为 n_i，β 粒子数为 m_i，则有 $n_0=1$，$m_0=0$，$n_i=m_{i-1}$，$m_i=3n_{i-1}+2m_{i-1}$。

算法设计：由以上数学模型，本题便转化为求数列 n_i 和 m_i 的第 t 项，可用递推的方法求得 n_t 和 m_t，此模型的算法如下：

```
main( )
 {int n[100],m[100],t,i;
  input(t);
  n[0] = 1;                        // 初始化操作
  m[0] = 0;
  for (i = 1; i <= t; i = i + 1)   // 进行 t 次递推
    { n[i] = m[i - 1];
      m[i] = 3 * n[i - 1] + 2 * m[i - 1]; }
  print(n[t]);                     // 输出结果
  print(m[t]);
 }
```

算法分析：此模型的空间需求较小，时间复杂度为 $O(n)$，但随着 n 的增大，所需时间越来越大。

数学模型 2：设在 t 时刻的 α 粒子数为 $f(t)$，β 粒子数为 $g(t)$，依题可知：

$$\begin{cases} g(t)=3f(t-1)+2g(t-1) & (1)\\ f(t)=g(t-1) & (2)\\ g(0)=0, f(0)=1 \end{cases}$$

下面求解这个递归函数的非递归形式。

由(2)得

$$f(t-1)=g(t-2) \tag{3}$$

将(3)代入(1)得

$$g(t)=3g(t-2)+2g(t-1) \quad (t\geqslant 2) \tag{4}$$
$$g(0)=0, \quad g(1)=3$$

(4)式的特征根方程为：

$$x^2-2x-3=0$$

其特征根为 $x_1=3, x_2=-1$。

所以该式的递推关系的通解为：

$$g(t)=C_1\cdot 3^t+C_2\cdot(-1)^t$$

代入初值 $g(0)=0, g(1)=3$ 得：

$$C_1+C_2=0$$
$$3C_1-C_2=3$$

解此方程组：

$$C_1=3/4, \quad C_2=-3/4$$

所以该递推关系的解为：

$$g(t)=\frac{3}{4}\cdot 3^t-\frac{3}{4}\cdot(-1)^t$$

$$f(t)=g(t-1)=\frac{3}{4}\cdot 3^{t-1}-\frac{3}{4}\cdot(-1)^{t-1}$$

即

$$f(t)=\frac{3^t}{4}+\frac{3}{4}\cdot(-1)^t$$

由数学模型 2，设计算法 2 如下：

```
main( )
{ int t,i;
  input(t);
  n = int(exp(t * ln(3)));
  m = int(exp((t + 1) * ln(3)));
  if (t mod 2 = 1)
     {n = n - 3;
      m = m + 3; }
  else
      {n = n + 3;
       m = m - 3; }
  n = n\4;              // 4 整除 n
  m = m\4;              // 4 整除 m
  print(n);
  print(m);
}
```

算法分析：在数学模型 2 中，运用数学的方法建立了递归函数并转化为非递归函数。它的优点是算法的复杂性与问题的规模无关。针对某一具体数据，问题的规模对时间的影响微乎其微。

通过以上两个模型可以看出，模型 2 抓住了问题的本质，尤其成功地运用了组合数学中关于常系数线性齐次递推关系求解的有关知识，因而使算法本身既具有通用性和可计算性，同时又达到了零信息冗余。

习题

（1）求 $2+22+222+2222+\cdots+\underbrace{22\cdots22}_{n\text{个}2}$（精确计算）。

（2）编写一个算法，其功能是给一维数组 a 输入任意 6 个整数，假设为 5，7，4，8，9，1，然后建立一个具有如图 3-4 所示的方阵，并打印出来（屏幕输出）。

（3）编程打印形如图 3-5 所示的 $n\times n$ 方阵。

5	7	4	8	9	1
1	5	7	4	8	9
9	1	5	7	4	8
8	9	1	5	7	4
4	8	9	1	5	7
7	4	8	9	1	5

图 3-4　方阵 1

1	2	3	4	5	6	7
24	25	26	27	28	29	8
23	40	41	42	43	30	9
22	39	48	49	44	31	10
21	38	47	46	45	32	11
20	37	36	35	34	33	12
19	18	17	16	15	14	13

图 3-5　方阵 2

（4）编程打印形如图 3-6 所示的 $n\times n$ 方阵的上三角阵。

（5）编写程序打印形如图 3-7 和图 3-8 所示的 $n\times n$ 方阵。

1	3	6	10	15
2	5	9	14	
4	8	13		
7	12			
11				

图 3-6 上三角阵

1	1	1	1	1	1
1	2	2	2	2	1
1	2	3	3	2	1
1	2	3	3	2	1
1	2	2	2	2	1
1	1	1	1	1	1

图 3-7 方阵 3

1	1	1	1	1
1	2	2	2	1
1	2	3	2	1
1	2	2	2	1
1	1	1	1	1

图 3-8 方阵 4

(6) 键盘输入一个含有括号的四则运算表达式,可能含有多余的括号,编程整理该表达式,去掉所有多余的括号,原表达式中所有变量和运算符相对位置保持不变,并保持与原表达式等价。

例:输入表达式　　应输出表达式

```
a+(b+c)          a+b+c
(a*b)+c/d        a*b+c/d
a+b/(c-d)        a+b/(c-d)
```

注意:输入 $a+b$ 时不能输出 $b+a$。表达式以字符串输入,长度不超过 255,输入不要判错。所有变量为单个小写字母。只是要求去掉所有多余括号,不要求对表达式化简。

(7) 写出计算 ackermann 函数 ack(m,n)的递归计算函数。对于 $m \geqslant 0$,$n \geqslant 0$,ack(m,n)定义为:

```
ack(0,n) = n + 1;
ack(m,0) = ack(m - 1,1);
ack(m,n) = ack(m - 1,ack(m,n - 1));
```

(8) 判断 s 字符串是否为"回文"的递归函数。

(9) 有一只经过训练的蜜蜂只能爬向右侧相邻的蜂房,不能反向爬行,如图 3-9 所示。试求出蜜蜂从蜂房 a 爬到蜂房 b 的可能路线数($0<a<b<100$)。

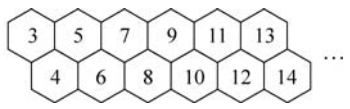

图 3-9 蜂房

(10) 狼找兔子问题:一座山周围有 n 个洞,顺时针编号为 0,1,2,3,4,…,$n-1$。一只狼从 0 号洞开始,顺时针方向计数,每当经过第 m 个洞时,就进洞找兔子。例如 $n=5$,$m=3$,狼经过的洞依次为 0,3,1,4,2,0。输入 m,n。试问兔子有没有幸免的机会?如果有,该藏在哪儿?

(11) 请编程求 $1\times2\times3\times\cdots\times n$ 所得的数末尾有多少个 0?(n 由键盘输入,$1000<n<10\,000$)

(12) 有 52 张牌,使它们全部正面朝上,第一轮是从第 2 张开始,凡是 2 的倍数位置上的牌翻成正面朝下;第二轮从第 3 张牌开始,凡是 3 的倍数位置上的牌,正面朝上的翻成正面朝下,正面朝下的翻成正面朝上;第三轮从第 4 张牌开始,凡是 4 的倍数位置上的牌按上面相同规则翻转,以此类推,直到翻的牌超过 104 张为止。统计最后有几张牌正面朝上,以及它们的位置号。

(13) A,B,C,D,E 5 人为某次竞赛的前五名,他们在名次公布前猜名次。

A 说:B 得第三名,C 得第五名。

B 说:D 得第二名,E 得第四名。

C 说：B 得第一名，E 得第四名。

D 说：C 得第一名，B 得第二名。

E 说：D 得第二名，A 得第三名。

结果每个人都猜对了一半，实际名次是什么呢？

(14) 编写算法求满足以下条件的 3 位整数 n：它是完全平方数，其中又有两位数字相同，如 144、676 等。

(15) 两个乒乓球队进行比赛，各出 3 人。甲队为 A、B、C 3 人，乙队为 X、Y、Z 3 人，已抽签决定比赛名单。有人向队员打听比赛的名单，A 说他不和 X 比，C 说他不和 X、Z 比，请编写算法找出 3 对赛手的名单？

(16) 编写算法对输入的一个整数，判断它能否被 4，7，9 整除，并输出以下信息之一：

能同时被 4，7，9 整除；

能被其中两个数(要指出哪两个)整除；

能被其中一个数(要指出哪一个)整除；

不能被 4，7，9 任一个整除。

(17) 完成给“余”猜数的游戏：

心里先想好一个 1～100 的整数 x，将它分别除以 3，4 和 7 并得到 3 个余数。把这 3 个余数输入计算机，计算机能马上猜出这个数。

```
Please think of a number between 1 and 100
Your number divided by 3 has a remainder of 1
Your number divided by 4 has a remainder of 0
Your number divided by 7 has a remainder of 5
Let me think a moment…
Your number was 40
```

(18) 求这样的两个数据：5 位数＝2×4 位数，9 个数字互不相同。

(19) 编写一函数，输入一个十六进制数，输出相应的十进制数。

(20) 用 1，2，3，4，5，6，7，8，9 这 9 个数字，填入□中使等式成立，每个数字恰好用一次：□□×□□□＝□□□□。

(21) 求这样的 6 位数：SQRT(6 位数)＝3 位数，9 个数字互不相同(SQRT 表示开平方)。

(22) 键盘输入 n 个正整数，把它们看作一个“数圈”，求其中连续 4 个数之和最大者。

(23) 输入一个 5 位数以内的正整数，完成以下操作：

① 判断它是几位数。

② 请按序输出其各位数字。

③ 逆序输出其各位数字。

(24) 乘式还原，有乘法运算如下：

$$\begin{array}{r} \bigcirc\bigcirc\bigcirc \\ \times\quad\ \bigcirc \\ \hline \bigcirc\bigcirc\bigcirc\bigcirc \end{array}$$

式中 8 个○位置上的数字全部是素数，请还原这算式。

(25) 乘式还原,有乘法运算如下:

```
   ○○○
 ×  ○○
-------
  ○○○○
 ○○○○
-------
 ○○○○○
```

式中18个○位置上的数字全部是素数(2,3,5或7),请还原这算式。

(26) 读入自然数 m 和 n($0 \leqslant m < n \leqslant 1000$),判断分数 m/n 是有限小数还是循环小数。如果 m/n 是有限小数,则输出分数的值;如果 m/n 为循环小数,则把循环部分括在括号中打印输出。

(27) 月份翻译:编程当输入数据为1～12时,翻译成对应的英语月份,如输入“3”翻译输出“march”。