第1章　算法及其基本设计方法

计算机科学的发展，为科学计算及数据处理提供了高速和高精度的计算工具。但计算机只能机械地执行人的指令，它本身不会主动地进行思维，也不可能发挥任何创造性。因此，在用计算机解决实际问题之前，首先要进行程序设计。通常，程序设计主要包括两个方面：行为特性的设计与结构特性的设计。所谓行为特性的设计，通常是指将解决问题的过程的每一个细节准确地加以定义，并且还应当将全部的解题过程用某种工具完整地描述出来。这一过程也称为算法的设计。所谓结构特性的设计是指为问题的解决确定合适的数据结构。数据结构与算法之间有着密切的关系。特别是对于数据处理问题，算法的效率通常与数据在计算机内的表示形式有着直接的关系。

本章主要讨论算法的基本概念、本书所采用的算法描述语言以及算法的基本设计方法。

1.1　算法的基本概念

概括地说，“算法”是指解题方案的准确而完整的描述。

对于一个问题，如果可以通过一个计算机程序，在有限的存储空间内运行有限长的时间而得到正确的结果，则称该问题是算法可解的。但算法不等于程序，也不等于计算方法。程序可以作为算法的一种描述，但程序通常还需考虑很多与方法和分析无关的细节问题，这是因为在编写程序时要受到计算机系统运行环境的限制。通常，程序设计的质量不可能优于算法的设计。

1.1.1　算法的一般特征

算法实际上是一种抽象的解题方法，它具有动态性。因此，算法的行为非常重要。作为一个算法，应具有以下四个特征。

(1) 能行性(effectiveness)

算法的能行性包括两个方面：一是算法中的每一个步骤必须是能实现的。例如，在算法中，不允许出现分母为零的情况；在实数范围内不能求一个负数的平方根等。二是算法执行的结果要能达到预期的目的。通常，针对实际问题设计的算法，人们总是希望能够得到满意的结果。

(2) 确定性(definiteness)

算法的确定性，是指算法中的每一个步骤都必须是有明确定义的，不允许有模棱两可的解释，也不允许有多义性。这一特征也反映了算法与数学公式的明显差异。在解决实际问题时，可能会出现这样的情况：针对某种特殊问题，数学公式是正确的，但按此数学公式设计的计算过程可能会使计算机系统无所适从，这是因为，根据数学公式设计的计算过程

只考虑了正常使用的情况，而当出现异常情况时，该计算过程就不能适应了。例如，某计算工具规定：大于100的数认为是比1大很多，而小于10的数不能认为是比1大很多；且在正常情况下出现的数或是大于100，或是小于10。但指令“输入一个X，若X比1大很多，则输出数字1，否则输出数字0”是不确定的。这是因为，在正常的输入情况下，这一指令的执行可以得到正确的结果，但在异常情况下（输入的X在10与100之间），这一指令执行的结果就不确定了。

（3）有穷性(finiteness)

算法的有穷性是指算法必须能在有限的时间内执行完，即算法必须能在执行有限个步骤之后终止。数学中的无穷级数，在实际计算时只能取有限项，即计算无穷级数的过程只能是有穷的。因此，一个数的无穷级数的表示只是一种计算公式，而根据精度要求确定的计算过程才是有穷的算法。

算法的有穷性还应包括合理的执行时间的含义。如果一个算法的执行时间是有穷的，但却需要执行千万年，显然这就失去了算法的实用价值。例如，克莱姆(Cramer)规则是求解线性代数方程组的一种数学方法，但不能以此为算法，这是因为，虽然总可以根据克莱姆规则设计出一个计算过程用于计算所有可能出现的行列式，但这样的计算过程所需的时间实际上是不能容忍的。还例如，从理论上讲，总可以写出一个正确的弈棋程序，而且这也并不是一件很困难的工作。由于在一个棋盘上安排棋子的方式总是有限的，而且，根据一定的规则，在有限次移动棋子之后比赛一定结束。因此，弈棋程序可以考虑计算机每一次可能的移动，它的对手每一次可能的应答，以及计算机对这些移动的可能应答等等，直到每个可能的移动停止下来为止。此外，由于计算机可以知道每次移动的结果，因此总可以选择一种最好的移动方式。但即使如此，这种弈棋程序还是不可能执行，因为所有这些可能移动的次数太多，所要花费的时间不能容忍。由上述两个例子可以看出，虽然许多计算过程是有限的，但仍有可能无实用价值。

（4）算法必须拥有足够的情报

一个算法是否有效，还取决于为算法的执行所提供的情报是否足够。例如，对于指令“如果小明是学生，则输出字母Y，否则输出N”。当算法执行过程中提供了小明一定不是学生的某种信息时，执行的结果将输出字母N；当提供的只是部分学生的名单，且小明恰在此名单之中，则执行的结果将输出字母Y。但如果在提供的部分学生的名单中找不到小明的名字，则在执行该指令时无法确定小明是否是学生。

通常，算法中的各种运算总是要施加到各个运算对象上，而这些运算对象又可能具有某种初始状态，这是算法执行的起点或是依据。因此，一个算法执行的结果总是与输入的初始数据有关，不同的输入将会有不同的结果输出。如果输入不够或输入错误，则算法本身也就无法执行或执行有错。一般来说，只有当算法拥有足够的情报时，该算法才是有效的；而如果提供的情报不够，则算法并不是有效的。

综上所述，所谓算法，是一组严谨地定义运算顺序的规则，并且每一个规则都是有效的且是明确的，此顺序将在有限的次数下终止。

1.1.2 数值型算法的特点

算法的执行总是与特定的计算工具有关，而每一种计算工具的有效数字的位数总是有限的。因此，在实际的数值计算过程中，所有参与运算的数通常都是近似的。

作为数值型算法，还具有以下五个特点。

(1) 理论上的精确运算与实际运算之间存在着差异。

例如，某计算工具具有七位有效数字（如 FORTRAN 中的单精度运算），在计算下列三个量

$$A = 10^{12},\ B = 1,\ C = -10^{12}$$

的和时，如果采用不同的运算顺序，就会得到不同的结果，即

$$A + B + C = 10^{12} + 1 + (-10^{12}) = 0$$

$$A + C + B = 10^{12} + (-10^{12}) + 1 = 1$$

而在数学上，$A+B+C$ 与 $A+C+B$ 是完全等价的。

由此可以看出，算法与计算公式是有差别的，数学上的一些运算规则、恒等变换公式在算法中不一定适用。

(2) 理论上的解题方案与实际能用性之间存在着差异。

例如，1.1.1 节中提到的克莱姆规则可以用于求解线性代数方程组，但由于用这种方法求解工程中实际的线性代数方程组时，所需要的执行时间不能容忍，因此，对于求解线性代数方程组，克莱姆规则并不实用。

还例如，要求一元二次方程

$$x^2 - (10^{12} + 1)x + 10^{12} = 0$$

的两个实根。如果用通常的求根公式，并且在计算过程中还假设计算工具具有七位有效数字，则计算过程为

$$x_{1,2} = \frac{1}{2}\left((10^{12} + 1) \pm \sqrt{(10^{12} + 1)^2 - 4 \times 10^{12}}\right)$$

$$= \frac{1}{2}\left(10^{12} \pm \sqrt{10^{24} - 4 \times 10^{12}}\right)$$

$$= \frac{1}{2}(10^{12} \pm 10^{12}) = \begin{cases} 10^{12} \\ 0 \end{cases}$$

对于最后的计算结果：

$$x_1 = 10^{12},\ x_2 = 0$$

经检验，$x_1 = 10^{12}$ 是满足原方程的，而 $x_2 = 0$ 不满足原方程。那么，怎么能可靠地求出一元二次方程的两个根呢？一般来说，对于一元二次方程

$$ax^2 + bx + c = 0$$

可以用下述方法求出两个根：

$$\begin{cases} x_1 = \dfrac{1}{2a}\left(-b - \mathrm{sgn}(b)\sqrt{b^2 - 4ac}\right) \\ x_2 = \dfrac{c}{ax_1} \end{cases}$$

其中 sgn(b)为 b 的符号函数，即

$$\mathrm{sgn}(b) = \begin{cases} 1, & b \geqslant 0 \\ -1, & b < 0 \end{cases}$$

由这个例子可以看出，有些数学上的计算公式实际上不能真正用于计算。这些公式本身是正确的，但参与公式中各种运算的数，由于受计算工具的有效数字位数的限制，都是近似的，也就是说，它们都是有误差的，而误差在计算过程中将会发生各种效应(关于这一点，将在第 2 章详细介绍)。

(3) 精确解法与近似解法往往没有本质区别。

所谓精确解法是指公式解法或直接解法。所谓近似解法一般是指迭代解法。前面已经提到，在算法执行过程中，参与各种运算的数通常都有误差，而这些误差通过运算后最终会影响结果。因此，即使用精确的计算公式，其最后得到的结果一般也是不准确的。在近似解法中，一般都给定精度要求，在计算过程中一般都要保证最后结果满足事先给定的精度要求。因此，精确解法所得的结果未必准确，而近似解法所得的结果未必不准确，两者没有本质的区别。

(4) 大多数问题不能简单地用某个标准算法来解决。

在工程中，对于同一类问题中的许多问题，其算法也往往是千差万别的，不可能用一个算法去解决同类中的所有问题，总是要根据不同问题的特点来设计不同的算法。例如，Romberg 求积法对于解决一般的积分问题是有效的，但不能解决高振荡函数的积分问题。还例如，Runge-Kutta 法可以有效地积分一般的常微分方程初值问题，但这种方法不适用于刚性方程。

(5) 小规模问题往往存在快速算法。

在数值计算领域中，有些小规模问题的计算过程已经被人们所熟悉并经常使用，对于这些计算过程通常还可以减少运算次数，从而在大量调用这些计算过程时可以大大减少计算工作量；有时还可以利用这些过程来构造解决大规模问题的快速算法。

例如，计算

$$A = ac + ad + bc + bd$$

按照通常的方法需要作 4 次乘法和 3 次加法。但如果将上式写成下列等价形式：

$$A = (a + b)(c + d)$$

则只需要作 1 次乘法和 2 次加法。显然，在需要对各种不同数据 a,b,c,d 来计算 A 的时候，用后一种等价形式来代替原来的算式是有实际意义的。

又例如，在许多实际问题中经常要遇到复数运算。一般的复数运算总是将它们的实部和虚部分别进行运算，因此，两个复数之间的 1 次运算，通常要进行多次的实数间的四则运算才能完成。为了提高复数运算的速度，一般的方法是减少相应的实数运算的次数。下面以两个复数的乘法为例来说明这个问题。

通常，两个复数相乘是这样来运算的：

$$\begin{aligned} e + \mathrm{j}f &= (a + \mathrm{j}b)(c + \mathrm{j}d) \\ &= (ac - bd) + \mathrm{j}(bc + ad) \end{aligned}$$

可见需要 4 次实数乘法才能完成 1 次复数乘法。但实际上，两个复数的乘积结果中，e 和 f

可以表示成

$$e = (ac - bd) = (a-b)d + a(c-d)$$
$$f = (bc + ad) = (a-b)d + b(c+d)$$

若令

$$P = (a-b)d,\ Q = a(c-d),\ R = b(c+d)$$

则

$$\begin{cases} e = P + Q \\ f = P + R \end{cases}$$

由此可以看出，为了计算两个复数乘积中的实部 e 和虚部 f，只需要作 3 次乘法就够了，比通常的方法少了 1 次。当然，这是以增加加减法次数为代价的，但这也是划算的，因为作 1 次乘法所花的时间一般要比作 1 次加减法的时间多得多。

1.2 算法描述语言

算法描述语言是表示算法的一种工具，它只面向读者，不能直接用于计算机，但很容易转换为计算机上能执行的程序。在本书中，绝大部分的算法都将采用本节所介绍的算法描述语言来描述。但为了方便，有的算法也可能直接用自然语言或流程图的形式来描述。在用算法描述语言描述一个算法时，如果被描述的算法可以用过程来实现，则一开始总是先要对输入与输出参数作必要的说明，并且第一行总是为：

PROCEDURE　过程名(输入输出参数表)

在算法正文中，必要时对算法中的每一行用自然数依次编上行号，以便在叙述中对算法进行说明。

下面简要叙述本书采用的算法描述语言中的各个语句并加以必要的说明。

1．符号与表达式

符号是以字母开头的字母和数字的有限串，用以表示变量名、数组名等，必要时也用来表示语句标号。

在语句标号后应跟随一个冒号，然后是语句。例如

loop：$i \leftarrow i+1$

在算法中，变量或数组的类型通常可从上下文看出，因此一般不需要作说明，除非在特殊情况下才作必要的说明。

有时，算法中的某些指令或某个子功能直接用叙述的方式给出，以便使算法更简洁。例如，“设 x 是 A 中的最大项”(其中 A 为一个数组)，“将 x 插入 L 中”(其中 L 是某个表数组)，等等。

算术运算符通常用＋，－，＊，/和 ↑，其中＊和 ↑ 一般可以省略，而沿用通常的数学表示法。

关系运算符用＝，≠，＜，＞，≤和≥来表示。

逻辑运算符用 and(与)，or(或)和 not(非)来表示。

2．赋值语句

试读结束：需要全本请在线购买： www.ertongbook.com

赋值语句的形式为

$a \leftarrow e$

其中 a 表示变量名或数组元素，e 表示算术表达式或逻辑表达式。如果 a 和 b 都是变量名或数组元素，则可用记号

$a \rightleftharpoons b$

表示将 a 与 b 的内容互换。而用记号

$a \leftarrow b \leftarrow e$

表示将表达式 e 的值同时赋给 a 和 b。

3. 控制转移语句

无条件转移语句用如下形式：

GOTO　标号

它导致转到具有指定标号的语句去执行。

条件控制语句有以下两种形式：

IF　C　THEN　S

或

IF　C　THEN　S_1
　　　　ELSE　S_2

其中 C 是一个逻辑表达式；S，S_1 和 S_2 可以是单一的语句，也可以是用一对花括号{ }括起来的语句组。如果 C 为"真"，则 S 或 S_1 被执行一次；如果 C 为"假"，在第一种形式中，IF 语句完成，而在第二种形式中，将执行 S_2 一次。在一般情况下，IF 语句执行完后，控制将进行到下一语句，除非在 S，S_1 或 S_2 中有 GOTO 语句使控制转到了别的地方。

4. 循环语句

循环语句有两种形式：一是 WHILE 语句，二是 FOR 语句。

(1) WHILE 语句的形式为

WHILE　C　DO　S

其中 C 是逻辑表达式；S 可以是单一的语句，也可以是用一对花括号{ }括起来的语句组。如果 C 为"真"，则执行 S，且在每次执行 S 之后都要重新检查 C；如果 C 为"假"，控制就转到紧跟在 WHILE 语句后面的语句。当控制第一次到达 WHILE 语句时 C 为"假"，则 S 一次也不执行。

WHILE 语句的功能等价于如下的 IF 语句：

```
loop: IF   C   THEN
        {     S
          GOTO   loop
        }
```

(2) FOR 语句的形式为

FOR　$i = init$　TO　$limit$　BY　$step$　DO　S

其中 i 是循环控制变量，$init$、$limit$ 和 $step$ 分别表示循环的初值、终值和步长，它们都可以是算术表达式；S 可以是单一的语句，也可以是用一对花括号{ }括起来的语句组。

当 $step>0$ 时，FOR 语句的功能等价于如下的 IF 语句：

```
        i=init
loop:  IF  i≤limit  THEN
          {    S
            i←i+step
            GOTO  loop
          }
```

当 $step<0$ 时，FOR 语句的功能等价于如下的 IF 语句：

```
        i=init
loop:  IF  i≥limit  THEN
          {    S
            i←i+step
            GOTO  loop
          }
```

在 FOR 语句中，如果 $step=1$，则 BY *step* 可以省略，变为

FOR　*i*=*init*　TO　*limit*　DO　*S*

5. 其它语句

在算法描述中，还可能要用到其它一些语句，读者完全可以了解它们的含义。下面再列出几个常见的语句。

EXIT 语句通常用于退出某个循环，使控制转到包含 EXIT 语句的最内层的 WHILE 或转到 FOR 循环后面的一个语句。

RETURN 语句用于结束算法的执行。如果算法是在最后一行的语句之后结束，则有时省略最后的 RETURN 语句。并且，在 RETURN 语句后面还允许使用用引号括起来的注释信息。

READ(或 INPUT)和 OUTPUT(或 WRITE)语句用于输入和输出。

最后还需说明：算法中的注释总是用一对方括号括起来；几个短语句可以写在一行上，但彼此之间用分号隔开；复合语句总是用一对花括号{ }括起来作为语句组。

1.3　算法的基本设计方法

计算机解题的过程实际上是在实施某种算法，因此，算法通常是指计算机算法。计算机算法不同于人工处理的算法。例如，为了计算定积分

$$S=\int_a^b f(x)\mathrm{d}x$$

人工处理的步骤为：

(1) 找出被积函数 $f(x)$ 的源函数 $F(x)$；

(2) 利用牛-莱公式计算 $S=F(b)-F(a)$。

但若利用计算机计算积分就不能按照上述步骤，这是因为通常很难用程序来寻找被积函

数的源函数，而且实际上也没有这个必要，实际问题中的被积函数往往不存在源函数。因此，计算机计算定积分通常采用数值积分法，根据实际的被积函数类型及精度要求选择相应的算法。

本节介绍几种常用的算法设计方法，有些方法之间有一定的联系。

1.3.1 列举法

列举法的基本思想是根据提出的问题，列举所有可能情况，并用问题中提出的条件检验哪些是需要的，哪些是不需要的。因此，列举法常用于解决“是否存在”或“有多少种可能”等类型的问题。例如，求解不定方程的问题可以采用列举法。

列举法的特点是算法比较简单，但当列举的可能情况较多时，执行列举算法的工作量将会很大。因此，在用列举法设计算法时，应该重点注意使方案优化，尽量减少运算工作量。通常，只要对实际问题作详细的分析，将与问题有关的知识条理化、完备化、系统化，从中找出规律，或对所有可能的情况进行分类，引出一些有用的信息，列举量是可以减少的。

下面的例子说明了列举法的基本思想，以及如何减少列举量。

例 1.1 设每只母鸡值 3 元，每只公鸡值 2 元，两只小鸡值 1 元。现用 100 元钱买 100 只鸡，问能买母鸡、公鸡、小鸡各多少只？

这实际上是一个不定方程的问题。设母鸡、公鸡、小鸡数分别为 I, J, K，则应满足如下条件：

$$\begin{cases} I + J + K = 100 \\ 3I + 2J + 0.5K = 100 \end{cases}$$

现有两个方程三个未知数，可以用列举法解决。粗略的列举算法如下。

算法 1.1 求解百鸡问题的粗略算法。

```
FOR  I=0  TO  100  DO
FOR  J=0  TO  100  DO
FOR  K=0  TO  100  DO
  { M←I+J+K
    N←3I+2J+0.5K
    IF  (M=0) and (N=100)  THEN
        OUTPUT I,J,K
  }
RETURN
```

在算法 1.1 中，各层循环均需循环 101 次，因此总循环次数为 101^3，列举量太大。但只要对问题稍作分析，很容易发现这个算法还可以改进，减少不必要的循环次数。

首先，考虑到母鸡为 3 元一只，因此，最多只能买 33 只母鸡，即算法 1.1 中的外循环只需要从 0 到 33 就可以了，没有必要从 0 到 100。

其次，考虑到公鸡为 2 元一只，因此，最多只能买 50 只公鸡。又考虑到对公鸡的列举是在算法的第二层循环中，且买一只母鸡的价钱相当于买 1.5 只公鸡。因此，在第一层循环中已确定买 I 只母鸡的情况下，公鸡最多只能买 $50-1.5I$ 只，即第二层中对 J 的循环只需要从 0 到 $50-1.5I$ 就可以了。

最后，考虑到总共买 100 只鸡，因此，在第一层循环中已确定买 I 只母鸡，且第二层已确定买 J 只公鸡的情况下，买小鸡的数量只能是 $K=100-I-J$，即第三层的循环已没有必要了。并且，在这种情况下，条件 $I+J+K=100$ 自然已经满足。

根据以上分析，可以将求解百鸡问题的算法改写成如下的算法。

算法 1.2 求解百鸡问题的改进算法。

```
FOR  I=0  TO  33  DO
FOR  J=0  TO  50-1.5I  DO
  {  K←100-I-J
     IF  (3I+2J+0.5K=100)  THEN
        OUTPUT  I, J, K
  }
RETURN
```

不难分析，算法 1.2 的列举量(即总循环次数)为

$$\sum_{i=0}^{33}(51-1.5i)=894$$

列举原理是计算机应用领域中十分重要的原理。许多实际问题，若采用人工列举是不可想象的，但由于计算机的运算速度快，擅长重复操作，因而可以进行大量的列举。可见，列举算法是计算机算法中的一个基础算法。实际上，列举就是从某个集合中一一列举各个元素，如果为了证明一个命题为真，则只要一一列举对于每一种可能情况命题均为真就行了。

列举法是比较笨拙而原始的方法，它最大的缺点是运算量比较大，但在有些实际问题中，局部使用列举法却是很有效的。例如，对于寻找路径、查找搜索等问题，局部使用列举法是一种行之有效的方法。

1.3.2 归纳法

列举算法具有结构简单的优点，但有两个主要缺点：一是对于某些实际问题，列举法的效率太低，特别是当列举量大到不能容忍时，列举法就不适用了；二是在很多实际问题中，列举量为无限，此时的列举算法是无效的。因此，列举法只适用于列举量为有限且不太大的情况。

归纳法的基本思想是通过列举少量的特殊情况，经过分析，最后找出一般的关系。显然，归纳法要比列举法更能反映问题的本质。但是，要从一个实际问题中总结归纳出一般的关系，并不是一件容易的事情，尤其是要归纳出一个数学模型就更为困难。而且，归纳过程通常也没有一定的规则可供遵循。从本质上讲，归纳就是通过观察一些简单而特殊的情况，最后总结出有用的结论或解决问题的有效途径。通常，归纳的过程分为以下四个步骤：

(1) 细心的观察；

(2) 丰富的联想；

(3) 继续尝试；

(4) 总结归纳出结论。

归纳是一种抽象，即从特殊现象中找出一般关系。由于在归纳的过程中不可能对所有

的可能情况进行列举，因而最后得到的结论还只是一种猜测(即归纳假设)。所以，对于归纳假设还必须加以严格的证明。

下面是归纳法的一个典型例子。

例 1.2 求前 n 个自然数的平方之和：

$$S_n = 1^2 + 2^2 + 3^2 + \cdots + n^2$$

首先，列举 S_n 的一些值：

$$S_1 = 1^2 = 1$$
$$S_2 = 1^2 + 2^2 = 5$$
$$S_3 = 1^2 + 2^2 + 3^2 = 14$$
$$S_4 = 1^2 + 2^2 + 3^2 + 4^2 = 30$$
$$S_5 = 1^2 + 2^2 + 3^2 + 4^2 + 5^2 = 55$$
$$S_6 = 1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2 = 91$$

由上述几个值还不能明显地看出 S_n 与 n 之间的关系。G. Polya 曾经提出如表 1.1 所示的归纳公式。由表 1.1 就很容易看出其中的规律性了，可以作出如下猜测：

$$(1^2 + 2^2 + \cdots + n^2)/(1 + 2 + \cdots + n) = (2n + 1)/3$$

又由于 $1+2+\cdots+n=n(n+1)/2$，因此得到

$$1^2 + 2^2 + \cdots + n^2 = n(n + 1)(2n + 1)/6$$

但这只是通过总结归纳而得到的一种猜测，是否正确还需证明。对归纳假设的证明通常采用数学归纳法。

表 1.1

n	1	2	3	4	5	6	……
$1+2+\cdots+n$	1	3	6	10	15	21	……
$1^2+2^2+\cdots+n^2$	1	5	14	30	55	91	……
$\frac{1^2+2^2+\cdots+n^2}{1+2+\cdots+n}$	$\frac{3}{3}$	$\frac{5}{3}$	$\frac{7}{3}$	$\frac{9}{3}$	$\frac{11}{3}$	$\frac{13}{3}$……	

最后必须指出，通过精心观察而提出的归纳假设得不到证实或者最后证明是错的情况也是常有的。

1.3.3 递推

所谓递推，是指从已知的初始条件出发，逐次推出所要求的各中间结果及最后结果。其中初始条件或是问题本身已经给定，或是通过对问题的分析与化简后确定。递推本质上也属于归纳法。工程上许多递推公式，实际上是通过对实际问题的分析与归纳而得到的，因此，递推算法是归纳的结果。

例 1.3 设 $\varphi=\frac{1}{2}(\sqrt{5}-1)$，求 $\varphi^n(n=0,1,\cdots)$的值。

显然，对于每一个 n，直接计算 φ^n是愚蠢的。本例中的计算具有明显的递推性质。

设 $\varphi_n=\varphi^n(n=0,1,2,\cdots)$，则有

$$\begin{cases}\varphi_0 = 1 \\ \varphi_n = \varphi \cdot \varphi_{n-1}, \quad n = 1,2,\cdots\end{cases}$$

即从初始条件 $\varphi^0=1$ 出发，逐步进行递推，在 φ^{n-1} 的基础上乘以 $\varphi=\frac{1}{2}(\sqrt{5}-1)$ 本身就可得到 φ^n。在这个递推公式中，每递推一步只需作一次乘法。

实际上，还可以验证，序列 $\varphi^n(n=0,1,\cdots)$ 满足下列三项递推关系：

$$\varphi^{n+1} = \varphi^{n-1} - \varphi^n$$

即计算序列 $\varphi_n=\varphi^n$ 还可以按下列递推公式来计算：

$$\begin{cases}\varphi_0 = 1,\ \varphi_1 = \frac{1}{2}(\sqrt{5} - 1) \\ \varphi_{n+1} = \varphi_{n-1} - \varphi_n,\ n = 1,2,\cdots\end{cases}$$

在这个递推公式中，只要从两个初值 φ_0 与 φ_1 出发，就可以逐步推出各 φ^n 的值，并且每递推一步只需作一次减法运算。

递推算法在数值计算中极为常见。数学中的许多特殊函数一般都有一些递推关系成立。但必须注意，在一些数值型的递推公式中，其数值计算有可能是不稳定的，在实际使用时要作适当的处理。例如，上面例 1.3 中的后一个三项递推关系，在真正用于计算时却是不稳定的，即实际不能直接用于递推计算。关于递推算法的稳定性问题将在第 2 章 2.2 节中作详细的讨论。

递推算法最主要的优点是算法结构比较简单，最适合于用计算机来处理。但稳定性问题不容忽视。

1.3.4 递归

在工程实际中，有许多概念是用递归来定义的，数学中的许多函数也是用递归来定义的。递归函数在可计算性理论与算法设计中都占有很重要的地位。

描述递归定义的函数或求解递归问题的过程称为递归算法。一个递归算法，本质上是将较复杂的处理归结为较简单的处理，直到最简单的处理。因此，递归的基础也是归纳。

例 1.4 求解 Hanoi 塔问题

设有三个塔座分别为 X,Y,Z。现有 n 个直径各不相同的圆盘，且按直径从小到大用自然数编号为 $1,2,\cdots,n$。开始时，此 n 个圆盘依下大上小的顺序放在塔座 X 上，如图 1.1 所示。现要根据下列原则将 X 塔座上的这 n 个圆盘移到 Z 塔座上：

(1) 每次只允许移动一个圆盘(从一个塔座到另一个塔座)；

(2) 移动时可用 Y 塔座作为中间塔座；

(3) 在移动过程中，任何一个塔座上均不允许大压小的情况发生。

只要对这个问题稍作分析，就可以得到如下的求解步骤：

(1) 如果 $n=1$，则可直接将这一个圆盘移动到目的柱上，过程结束。如果 $n>1$，则进行下一步。

(2) 设法将起始柱的上面 $n-1$ 个圆盘(编号 1 至 $n-1$)按移动原则移到中间柱上。

(3) 将起始柱上的最后一个圆盘(编号为 n)移到目的柱上。

(4) 设法将中间柱上的 $n-1$ 个圆盘按移动原则移到目的柱上。

试读结束：需要全本请在线购买： www.ertongbook.com

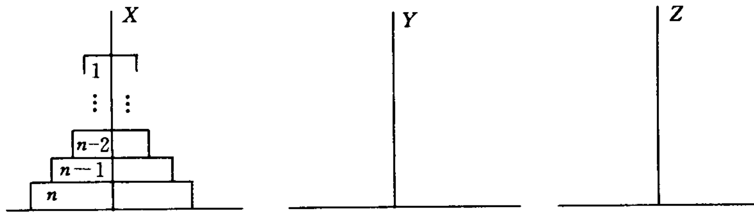

图 1.1 Hanoi 塔问题

在上述步骤中，(2)与(4)实际上还是 Hanoi 塔问题，只不过其规模小一些而已。如果最原始的问题为 n 阶 Hanoi 塔问题，且表示为

Hanoi (n,X,Y,Z)

则(2)与(4)为 $n-1$ 阶 Hanoi 塔问题，分别表示为

Hanoi $(n-1,X,Z,Y)$

Hanoi $(n-1,Y,X,Z)$

其中第一个参数表示问题的阶数(即需要移动的圆盘个数)，第二、三、四个参数分别表示起始柱、中间柱与目的柱。如果再用过程

move (X,n,Y)

表示将 X 塔座上的 n 号圆盘移到 Y 塔座上，则可得到求解 n 阶 Hanoi 塔的算法如下。

算法 1.3 求解 n 阶 Hanoi 塔问题。

```
PROCEDURE  AHanoi  (n,X,Y,Z)
IF  n=1  THEN  move (X,1,Z)
ELSE
  { AHanoi (n-1,X,Z,Y)
    move (X,n,Z)
    AHanoi (n-1,Y,X,Z)
  }
RETURN
```

算法 1.3 是一个典型的递归算法——自己调用自己。如果一个算法显式地调用自己，则称为直接递归。算法 1.3 是一个直接递归的算法。如果算法 P 调用另一个算法 Q，而算法 Q 又调用算法 P，则称算法 P 为间接递归。

有些实际问题，既可以归纳为递推算法，又可以归纳为递归算法。但递推与递归的实现方法是大不一样的。递推是从初始条件出发，逐次推出所需求的结果；而递归则是从算法本身到达递归边界。通常，递归算法要比递推算法更清晰易读，其结构也比较简练。特别是在许多比较复杂的问题中，很难找到从初始条件推出所需结果的全过程，此时，设计递归算法要比递推算法容易得多。但递归算法也有一个致命的缺点，即执行的效率比较低。通常，递归最适用于写算法。

1.3.5 减半递推

通常，解决实际问题的复杂程度与问题的规模有密切的关系。因此，如果将实际问题

的规模逐渐减小，则可以明显地降低解决问题的复杂程度。算法设计的这种方法称为分治法，即对问题分而治之。在例 1.4 中所采用的方法实际上也是一种分而治之的方法。移动 n 个圆盘比较复杂，将它化为 $n-1$ 个圆盘与 1 个圆盘的问题，1 个圆盘可以直接移动，而 $n-1$ 个圆盘的移动又可以化为 $n-2$ 个圆盘与 1 个圆盘的问题，以此类推，直到都变为 1 个圆盘移动的问题为止，问题就解决了。但像例 1.4 那种对问题的分裂过程，其速度太慢，因为经过一次分裂，只将问题的规模减小了 1。在工程上，常用的分治方法是采用减半递推技术。这个技术在快速算法的研究中有很重要的实用价值。

所谓"减半"，是指通过一次分裂将问题的规模减半，但问题的性质不变。假设原问题的规模为 n，则可以用与问题有关的特定方法将原问题化为 C(C 是与规模无关而只与问题性质有关的常数)个规模减半的问题，然后通过研究规模为 $n/2$ 的问题(显然与原问题的性质相同，只是规模不同而已)来解决原问题。由于问题的规模减小了，会给解决问题带来不少的方便。但是，在规模减半的过程中，势必要增加某些辅助工作，在分析算法工作量时必须予以考虑。

所谓"递推"，是指重复上述"减半"的过程。因为规模为 $n/2$ 的问题同样又可以化为 C 个规模为 $n/4$ 的相同性质的问题。以此类推，直到使问题的规模减到最小，以致于能很方便地解决为止。

例 1.5 有序数组的对分查找。

设一个有序数组 A 的长度为 n，其中的元素值是按非递减顺序排序的，现要在该数组中查找值为 x 的元素下标 j。对分查找的过程如下：

将被查找的值 x 与数组的中间一个元素进行比较，有以下三种可能：

(1) 若 x 等于数组中间这个元素值，则查找到，过程结束。

(2) 若 x 大于数组中间这个元素值，则从数组的后半部分中去查找。

(3) 若 x 小于数组中间这个元素值，则从数组的前半部分中去查找。

这个过程一直进行到查找成功或数组长度被减到 0 为止。

这个方法实际上是根据每次的试探结果将有序数组的长度减半，并有规则地划分出可能包含被查值 x 的部分。

下面是对有序数组对分查找的算法。

算法 1.4 有序数组的对分查找。

输入：有序数组 $A(1:n)$，被查项 x。

输出：若 x 在数组 A 中，则输出 $A(j)=x$ 的 j，否则输出 $j=0$。

```
PROCEDURE  ABSRCH (A,n,x,j)
k←1; m←n
WHILE  k≤m DO
  { j←INT [(k+m)/2]
    IF  x=A(j)  THEN  EXIT
    IF  x<A(j)  THEN  m←j-1
    ELSE  k←j+1
  }
IF  k>m  THEN  j←0
OUTPUT  j
```

RETURN

对于有些实际问题，可以设计出直观的减半递推算法，经过逐次的减半递推，直接得到所需求的结果，如例 1.5 那样。而对于另外一些问题，根据减半递推技术设计出的算法是递归算法，在执行过程中只能靠算法本身到达递归边界，在后面各章中我们将会遇到这样的算法。

1.3.6 回溯法

前面讨论的递推与递归算法本质上是对实际问题进行归纳的结果，而减半递推技术也是归纳法的一个分支。但有些实际问题却很难归纳出一组简单的递推公式或直观的求解步骤，并且也不能进行无限的列举。对于这类问题，一种有效的方法是"试"。通过对问题的分析，找出解决问题的一个线索，然后沿着这个线索往前试探，若试探成功，就得到解；若试探失败，就逐步往回退，换别的路线再往前试探。这种方法称为回溯法。回溯法在处理复杂数据结构方面，应用很广泛。

下面举一个例子说明回溯法的应用。

例 1.6 求解皇后问题

由 n^2 个方块排成 n 行 n 列的正方形，称之为"n 元棋盘"。如果两个皇后位于 n 元棋盘上的同一行、或同一列、或同一对角线上，则称它们在互相攻击。现要求找出使 n 元棋盘上的 n 个皇后互不攻击的布局。

n 个皇后在 n 元棋盘上的布局共有 n^n 种，为了从中找出互不攻击的布局，需要对此 n^n 种方案逐个进行检查，将有攻击的布局删除掉。这是一种列举法。但这种方法对于较大的 n，其工作量会急剧增加。而实际上，从互不攻击的布局的要求可分析出逐一列举也是没有必要的。例如，每一行上只能放一个皇后。又例如，如果第一行上的皇后已经在某一列，则其它行上的皇后就不可能在该列。下面用回溯法来设计求解这个问题的算法。

首先，将问题归结为在 n 元棋盘的每一行上安置一个皇后，并假设第 i 个皇后被安置在第 i 行上。定义一个一维数组 $A(1:n)$，其中每一个元素 $A(i)$ $(i=1,2,\cdots,n)$ 用于在安置皇后过程中随时记录第 i 行上的皇后所在的列号。由此可知，在这种情况下，实际上已经剔除了两个皇后在同一行上的可能性。因此，在安置每一行上的皇后时，只需考虑每两个皇后不能在同一列或同一对角线上即可。容易验证，第 i 行上的皇后与第 k 行上的皇后正好在同一列上的充要条件为

$$A(i) - A(k) = 0$$

正好同在某一对角线上的充要条件为

$$|A(i) - A(k)| - |i - k| = 0$$

初始时，将每一行的皇后均放在第一列，即初始状态为 $A(i)=1$，$i=1,2,\cdots,n$。

然后，从第一行(即 $i=1$)开始进行以下过程。

设前 $i-1$ 行上的皇后已经布局好，即它们互不攻击。现考虑安排第 i 行上的皇后位置，使之与前 $i-1$ 行上的皇后也都互不攻击。为了实现这一点，可以从第 i 行皇后的当前位置开始向右搜索：

(1) 若 $A(i) \leqslant n$，则需检查第 i 行皇后与前 $i-1$ 行的皇后是否互不攻击。若无攻击，

则考虑安排下一行皇后的位置；若有攻击，则将第 i 行皇后右移一个位置，重新进行这个过程。

(2) 若 $A(i)>n$，则说明在前 $i-1$ 行皇后的当前布局下，第 i 行皇后已无法安置。此时，将第 i 行皇后重新放在第一列，且回退一行，再考虑第 $i-1$ 行皇后与前 $i-2$ 行皇后均互不攻击的下一个位置。在这种情况下，如果已退到第 0 行(n 元棋盘不存在第 0 行)，则过程结束。

(3) 若当前安置好的皇后是在最后一行(即第 n 行)，则说明已找到了 n 个皇后互不攻击的一个布局，将这个布局打印输出。然后将第 n 行的皇后右移一个位置，重新进行这个过程，以便进一步找出另外的布局。

综合以上过程，可以形象地概括成一句话："向前走，碰壁回头"。这种方法也称为深度优先搜索 DFS(Depth First Search)技术。下面是求解皇后问题的算法。

算法 1.5 求解皇后问题

输入：n。

输出：n 个皇后互不攻击的各种布局，即数组元素 $A(i)$ $(i=1,2,\cdots,n)$。

```
PROCEDURE  AQUEEN(n)
定义数组 A(1:n)
FOR  i=1  TO  n  DO  A(i)←1
i←1
loop:
IF  A(i)≤n  THEN
  {  k←1
     WHILE  (k≤i-1) and [(A(i)-A(k))*(|A(i)-A(k)|-|i-k|)]≠0
     DO  k←k+1
     IF  k≤i-1  THEN
       { A(i)←A(i)+1;  GOTO  loop}
     i←i+1
     IF  i≤n  THEN  GOTO  loop
     OUTPUT  A(i) (i=1,2,…,n)
     A(n)←A(n)+1; i←n;  GOTO  loop
  }
ELSE
  {  A(i)←1; i←i-1
     IF  i<1  THEN  RETURN
     A(i)←A(i)+1;  GOTO  loop
  }
```

1.3.7 数字模拟法

数字模拟也称数字仿真。

在自然界和日常生活中，许多现象带有不确定的性质，有些问题甚至很难建立数学模型。对这类实际问题很难建立列举、递推、递归或回溯等算法，通常采用模拟法。通过编制程序，利用数字计算机进行模拟，称为数字模拟。在对实际问题中的随机现象进行数字模拟时，通常用计算机产生的随机数来表示。由于计算机产生的随机数总是要受有效数字位

数的限制，其随机的意义要比实际问题中真实的随机变量差一些。因此，计算机产生的随机数称为伪随机数。

产生随机数最常用的方法是在区间[0,1]上生成均匀分布的随机数，然后利用这种分布再产生模拟其它分布(如正态分布、指数分布等)的随机数。而产生均匀分布的随机数通常用线性同余法，其计算公式为

$$x = \mathrm{mod}(j * x + k, m)$$

其中 x 取值于 $0\sim(m-1)$之间。m 越大，其随机性质越好。且 x 取不同的初值，就可以产生不同的随机数序列。如果要产生 0～1 之间的随机数，则只要令 $y=x/m$ 即可。例如，在上述公式中的参数可取 $j=2053, k=13849, m=2^{16}$。下面是产生 0～1 之间均匀分布的随机数的算法。

算法 1.6 产生 0～1 之间均匀分布的随机数。

输入：x 为随机数种子。第一次调用时由调用过程给出初值，以后由本过程自动返回循环使用。

输出：0～1 之间的一个随机数 y；循环使用的种子值 x。

```
PROCEDURE ARND(x,y)
x←mod(2053 * x+13849,2^16)
y←x/2^16
RETURN
```

下面的例子是利用随机数计算定积分

例 1.7 用蒙特卡洛(Monte Carlo)法计算定积分

$$S = \int_a^b f(x)\mathrm{d}x$$

不失一般性，假设 $a<b, 0<f(x)<d$，其中 $d \geqslant \max[f(x)], x\in[a,b]$，如图 1.2 所示。

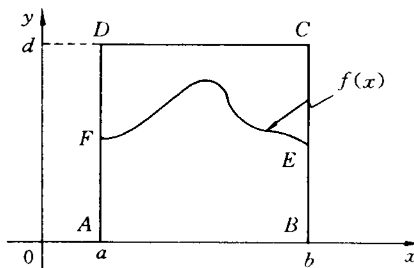

图 1.2 蒙特卡洛法计算定积分例

蒙特卡洛法计算定积分的基本思想是：产生 n(n 足够大)个均匀分布在长方形 $ABCD$ 上的随机数点(x_i, y_i)，其中 x_i 为均匀分布在区间$[a,b]$上的随机数，y_i 为均匀分布在区间$[0,d]$上的随机数。设其中落在曲边梯形 $ABEF$ 上的随机数点数为 m，则曲边梯形 $ABEF$ 的面积(即定积分值 S)为

$$S = \frac{m}{n}(b-a)d$$

其算法如下。

算法 1.7 蒙特卡洛法计算定积分。

输入：积分上、下限 b、a；$d \geqslant \max\limits_{x\in[a,b]}[f(x)]$；随机数点个数 n。

输出：定积分值 S。

```
PROCEDURE  AMONCAR (a,b,d,n,S)
```

```
m←0; t=1
FOR  i=1  TO  n  DO
  {  ARND(t,x); x←a+(b−a)*x
     ARND (t,y); y←d*y
     IF y<f(x)  THEN  m←m+1
  }
S←m*(b−a)*d/n
OUTPUT  S
RETURN
```

在算法 1.7 中，需要调用由算法 1.6 提供的产生 0～1 之间均匀分布的随机数的过程 ARND。只要 n 足够大，用蒙特卡洛法计算得到的定积分近似值的近似程度是比较好的。而且，这种方法特别适用于计算多重积分。

1.3.8 数值法

数值法又称非直接解法。

尽管许多实际问题的求解有现成的公式或固定的直接解法，例如：五次以下的多项式方程有公式解法，计算定积分有牛-莱公式等。但工程上的绝大部分实际问题用求解析解的方法并不适宜，甚至是很难实现的。例如，五次以上的多项式方程就没有公式解法，所有的超越方程更没有公式解；当被积函数的源函数不能用初等函数表示时，无法使用牛-莱公式来计算定积分。有的问题虽然有解析解，但由于函数关系太复杂，其实用价值也不大。因此，用数值法求解已成为计算机算法的一种重要方法，被大量地用于解决数值型的问题。本书的第 4 章到第 10 章主要讨论用数值法求解数值型的问题，在此不再详细叙述了。

习　题

1.01　已知 $\sqrt{1001}=31.64$，$\sqrt{1000}=31.62$。试分别计算 $\sqrt{1001}-\sqrt{1000}$ 与 $1/(\sqrt{1001}+\sqrt{1000})$ 的值，并比较它们的结果与真值 $\sqrt{1001}-\sqrt{1000}=1/(\sqrt{1001}+\sqrt{1000})=0.015808$（具有五位有效数字）的差。

1.02　利用 1.2 节介绍的算法描述语言，写出求 2 到 1000 之间所有素数的算法。

1.03　若 n 位数的各位数的 n 次方之和等于该数，则称该数为 armstrong 数。例如：

$$153=1^3+5^3+3^3$$

$$1634=1^4+6^4+3^4+4^4$$

试设计一个算法，求出 2，3，4 位数中的所有 armstrong 数。

1.04　三边长都是整数的直角三角形称为整数直角三角形。设 a、b、c 是整数直角三角形的三条边，且 $a\leqslant b<c$。试设计一个算法，当输入某个整数 b 时，求出构成整数直角三角形的所有 a 与 c 的可能值。

1.05　有一对刚出生的小兔子，一个月后长成大兔子，再过一个月以后，每月生一对小兔子。在没有死亡的情况下，问在第 n 个月后总共有多少对兔子。

试读结束：需要全本请在线购买： www.ertongbook.com