第1章 算 法

1.1 算法的基本概念

什么是算法呢？概括地说，所谓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，则称这个问题是算法可解的。但算法不等于程序，也不等于计算方法。当然，程序也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，程序的编制不可能优于算法的设计。

1.1.1 算法的基本特征

作为一个算法，一般应具有以下几个基本特征。

1. 能行性(effectiveness)

算法的能行性包括以下两个方面：

(1) 算法中的每一个步骤必须能够实现。如在算法中不允许执行分母为0的操作，在实数范围内不能求一个负数的平方根等。

(2) 算法执行的结果要能够达到预期的目的。

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。但一个算法又总是在某个特定的计算工具上执行的，因此，算法在执行过程中往往要受到计算工具的限制，使执行结果产生偏差。例如，在进行数值计算时，如果某计算工具具有7位有效数字(如程序设计语言中的单精度运算),则在计算下列三个量

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

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。因此，算法与计算公式是有差别的。在设计一个算法时，必须要考虑它的能行性，否则是不会得到满意结果的。

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。这一性质也反映了算法与数学公式的明显差别。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，此计算过程就不能适应了。

需要全本请访问 www.ertongbook.com

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做完，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，即计算无穷级数值的过程只能是有穷的。因此，一个数的无穷级数表示的只是一个计算公式，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。如果一个算法需要执行千万年，显然也就失去了实用价值。

4. 拥有足够的情报

一个算法是否有效，还取决于为算法所提供的情报是否足够。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。当输入不够或输入错误时，算法本身也就无法执行或导致执行有错。一般来说，当算法拥有足够的情报时，此算法才是有效的，而当提供的情报不够时，算法并不有效。

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、明确的,此顺序将在有限的次数下终止。

.2 算法的基本要素

一个算法通常由两种基本要素组成：一是对数据对象的运算和操作，二是算法的控制结构。

1. 算法中对数据的运算和操作

每个算法实际上是按解题要求从环境能进行的所有操作中选择合适的操作所组成的一组指令序列。因此，计算机算法就是计算机能处理的操作所组成的指令序列。

通常，计算机可以执行的基本操作是以指令的形式描述的。一个计算机系统能执行的所有指令的集合，称为该计算机系统的指令系统。计算机程序就是按解题要求从计算机指令系统中选择合适的指令所组成的指令序列。在一般的计算机系统中，基本的运算和操作有以下四类：

(1) 算术运算　主要包括加、减、乘、除等运算。

(2)逻辑运算　主要包括“与”、“或”、“非”等运算。

(3)关系运算　主要包括“大于”、“小于”、“等于”、“不等于”等运算。

(4) 数据传输　主要包括赋值、输入、输出等操作。

前面提到，计算机程序也可以作为算法的一种描述，但由于在编制计算机程序时通常要考虑很多与方法和分析无关的细节问题(如语法规则),因此,在设计算法的一开始,通常并不直接用计算机程序来描述算法，而是用别的描述工具（如流程图、专门的算法描述语言，甚至自然语言）来描述算法。但不管用哪种工具来描述算法，算法的设计一般都应从上述四种基本功能操作考虑，按解题要求从这些基本操作中选择合适的操作组成解题的操作序列。算法的主要特征着重于算法的动态执行，它区别于传统的着重于静态描述或按演绎方式求解问题的过程。传统的演绎数学是以公理系统为基础的，问题的求解过

程是通过有限次推演来完成的，每次推演都将对问题作进一步的描述，如此不断推演，直到直接将解描述出来为止。而计算机算法则是使用一些最基本的操作，通过对已知条件一步一步的加工和变换，从而实现解题目标。这两种方法的解题思路是不同的。

2. 算法的控制结构

一个算法的功能不仅取决于所选用的操作，而且还与各操作之间的执行顺序有关。算法中各操作之间的执行顺序称为算法的控制结构。

算法的控制结构给出了算法的基本框架，它不仅决定了算法中各操作的执行顺序，而且也直接反映了算法的设计是否符合结构化原则。描述算法的工具通常有传统流程图、N-S 结构化流程图、算法描述语言等。一个算法一般都可以用顺序、选择、循环三种基本控制结构组合而成。

1.2 算法描述语言

在本书中，对算法的描述，根据具体情况将采用两种描述方法：C 语言描述和简单的算法描述语言。对于没有学过 C 语言的读者，只要掌握了下面所介绍的算法描述语言，就可以很方便地将书中的算法转换成用读者所熟悉的某种程序设计语言来描述。

下面所介绍的算法描述语言只是一种表示算法的工具，它只面向读者，不能直接用于计算机，实际使用时还需将它转换成某种计算机语言来表示。

在用算法描述语言描述一个算法时，开始一般都要对输入和输出参数作必要的说明。下面简要叙述本书所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，主要用以表示变量名、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 = i + 1
```

在具体的算法描述中，变量或数组的数据类型一般可以从上下文中看出，因此，在算法中一般对变量或数组的数据类型不作说明，除非在特殊情况下才作必要的说明。

有时，为了使算法更清楚，算法中的某些指令或子过程直接用叙述的方式给出。例如：“设 x 是 A 中的最大项”(其中 A 是一个数组)；“将 x 插入到 L 之中”(其中 L 是某个表)，等等。

在算法中，算术运算符沿用数学中的表示法：

(1) 关系运算符用 =、≠、<、>、≤、≥等表示。

(2)逻辑运算符用 and(与)、or(或)、not(非) 等表示。

2. 赋值语句

赋值语句的形式为

```
a = e
```

其中 a 为变量名或数组元素，e 为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← →b

表示将 a 与 b 的内容进行交换；而用记号

a=b=e

表示将表达式 e 的计算结果同时赋给 a 与 b。

3. 控制转移语句

无条件转移语句的形式为

GOTO 标号

它导致转到具有指定标号的语句去执行。

条件转移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 是单一的语句或者是用一对括号{}括起来的语句组。如果 C 的值为“真”，则 S 或 S_1 被执行一次。如果 C 的值为“假”，则在第一种形式中，不执行 S；在第二种形式中，将执行 S_2 一次。在所有情况下，最后都将转到 IF 语句后面的语句去执行，除非在 S、S_1 或 S_2 中有 GOTO 语句将控制转到别的地方。

4. 循环语句

循环语句有两种形式：一是 WHILE 语句，二是 FOR 语句。

WHILE 语句的形式为

WHILE C DO S

其中 C 是逻辑表达式，S 是单一的语句或者是用一对括号{}括起来的语句组。如果 C 的值为“真”，则执行 S，且在每次执行 S 后都要重新检查 C 的值 如果 C 的值为“假”，则将转到紧跟在 WHILE 语句后面的语句去执行。WHILE 语句的功能等价于如下的 IF 语句：

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

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 都是算术表达式，S 是单一的语句或者是用一对括号{}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 循环语句中，如果 step = 1，则 BY step 可以省略，变为

```
FOR i = init TO limit DO S
```

5. 其他语句

在算法描述中，还可能要用到其他一些语句，读者在遇到时完全可以理解它们的含义。下面只列出几个常见的语句。

EXIT 语句主要用于退出某个循环，使控制转到包含 EXIT 语句的最内层的 WHILE 或 FOR 循环后面的一个语句去执行。

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，则可以省略 RETURN 语句。并且，在 RETURN 语句中允许使用用引号括起来的注释信息。

READ(或 INPUT)和 WRITE(或 PRINT,或 OUTPUT) 语句分别用于输入和输出。

最后还需要说明的是，算法中的注释总是用一对方括号 [] 括起来。几个短的语句可以写在同一行上，但彼此之间要用分号隔开。另外，复合语句总是用一对花括号 { } 括起来作为语句组。有时为了叙述方便，对算法中的每一行可加上行号。

1.3 算法设计基本方法

计算机解题的过程实际上是在实施某种算法，这种算法称为计算机算法。计算机算法不同于人工处理的方法。例如，为了计算定积分

$$S = \int_a^b f(x)\,dx$$

人工处理的步骤为

(1) 找出被积函数 f(x)的原函数 F(x)。

(2) 利用牛顿-莱布尼兹公式计算 S = F(b) - F(a)。

但用计算机计算这个积分不能按照上述步骤，因为通常很难用计算机程序来寻找被积函

需要全本请访问 www.ertongbook.com

数的原函数，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原函数（即原函数不能用初等函数表示）。因此，在实际问题中，利用计算机计算定积分通常采用数值积分的方法，根据实际被积函数的类型及精度要求选择相应的算法。

本节介绍工程上常用的几种算法设计方法。在实际应用时，各种方法之间往往存在着一定的联系。

1. 列举法

列举法的基本思想是：根据提出的问题，列举所有可能的情况，并用问题中给定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。

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，使方案优化，尽量减少运算工作量，是应该重点注意的。通常，在设计列举算法时，只要对实际问题进行详细分析，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，就可以大大减少列举量。

下面举例说明利用列举算法解决问题时如何对算法进行优化。

【例 1.1】 设每只母鸡值 3 元，每只公鸡值 2 元，每只小鸡值 0.5 元。现要用 100 元钱买 100 只鸡，设计买鸡方案。

假设买母鸡 I 只，公鸡 J 只，小鸡 K 只。根据题意，粗略的列举算法如下。

【算法 1.1】 求解百鸡问题。

```
PROCEDURE BAIJI
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=100)and(N=100)) THEN
           OUTPUT I,J,K
      }
RETURN
```

在算法 1.1 中，共嵌套有三层循环，每层循环各需要循环 101 次，因此，总循环次数为 101^3。但只要对问题进行分析，发现还可以对这个算法进行优化，减少大量不必要的循环次数。

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

其次，考虑到公鸡为 2 元一只，因此，公鸡最多只能买 50 只。又考虑到对公鸡的列举是在算法的第二层循环中，此时已经买了 I 只母鸡，且买一只母鸡的价钱相当于买 1.5 只公鸡。因此，由第一层循环已经确定买 I 只母鸡的前提下，公鸡最多只能买 50 - 1.5I 只，即第二层对 J 的循环只需从 0 到 50 - 1.5I 就可以了。

最后，考虑到买的总鸡数为 100,而由第一层循环已确定买 I 只母鸡,由第二层循环已确定买 J 只公鸡，因此，买小鸡的数量只能是 K = 100 - I - J，即第三层循环已经没有必要了。

经过以上分析，可以将算法 1.1 改写成如下算法。

【算法 1.2】求解百鸡问题。

```
PROCEDURE BAIJI
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(即循环次数)为

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

算法 1.2 的 C 语言描述如下:

```
#include "stdio.h"
main()
{ int i,j,k;
  for (i=0; i<=33; i++)
  for (j=0; j<=50-1.5*i; j++)
     { k=100-i-j;
       if (3*i+2*j+0.5*k==100.0)
          printf("%5d%5d%5d\n",i,j,k);
     }
}
```

运行结果如下:

```
 2   30   68
 5   25   70
 8   20   72
11   15   74
14   10   76
17    5   78
20    0   80
```

列举原理是计算机应用领域中十分重要的原理。事实上，对许多实际问题采用人工列举是不可想象的。由于计算机的运算速度快，擅长重复操作，因此可以很方便地进行大量列举。列举算法虽然是一种比较笨拙而原始的方法，其运算量比较大，但在有些实际问

题中(如寻找路径、查找、搜索等问题),局部使用列举法却是很有效的。因此,列举算法是计算机算法中的一个基础算法。

2. 归纳法

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,并且可以解决列举量为无限的问题。但是,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更为困难。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。

归纳是一种抽象,即从特殊现象中找出一般关系。但由于在归纳的过程中不可能对所有的情况进行列举,因此,最后由归纳得到的结论还只是一种猜测,还需要对这种猜测加以必要的证明。实际上,不能证明通过精心观察而得到的猜测,或最后证明猜测是错误的,也是常有的事。

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和最后结果。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而得到确定。递推本质上也属于归纳法,工程上许多递推关系式实际上是通过对实际问题的分析与归纳而得到的,因此,递推关系式往往是归纳的结果。

递推算法在数值计算中是极为常见的。但是,对于数值型的递推算法必须要注意数值计算的稳定性问题。下面通过一个具体的例子来说明递推算法的数值稳定性问题。

【例 1.2】 计算定积分

$$I_n = \int_0^1 \frac{x^n}{x+5}dx,\ n = 0,1,2,\cdots,20$$

利用牛顿-莱布尼兹公式分别计算这 21 个积分显然是很麻烦的。如果对这个积分稍作分析,可以发现相邻两个积分之间存在以下关系:

$$I_n + 5I_{n-1} = \frac{1}{n} \tag{1.1}$$

根据这个关系就可以得到如下递推公式:

$$I_n = \frac{1}{n} - 5I_{n-1} \tag{1.2}$$

由这个递推公式可以看出,只要知道 I_{n-1} 就可以算出 I_n,也就是说,只要知道了 I_0,就可以通过这个递推公式计算出所有的积分值 $I_n(n=1,2,\cdots,20)$。

利用牛顿-莱布尼兹公式可以计算出积分 I_0 的值为

$$I_0 = \int_0^1 \frac{1}{x+5}dx = \ln(6/5) \approx 0.182322$$

由以上的分析和归纳,可以得到计算本例中 21 个积分的递推算法如下:

$$\begin{cases} I_0 = 0.182322 \\ I_n = \dfrac{1}{n} - 5I_{n-1},\ n = 1,2,\cdots,20 \end{cases} \tag{1.3}$$

利用递推算法(1.3),用 C 语言编制程序,采用双精度运算,其计算结果如表　　所示。

表 1.1 计算结果(1)

n	I_n	n	I_n
0	1.82322e-01	11	-2.16268e+01
1	8.83900e-02	12	1.08218e+02
2	5.80500e-02	13	-5.41011e+02
3	4.30833e-02	14	2.70513e+03
4	3.45833e-02	15	-1.35256e+04
5	2.70833e-02	16	6.76279e+04
6	3.12500e-02	17	-3.38139e+05
7	-1.33929e-02	18	1.69070e+06
8	1.91964e-01	19	-8.45348e+06
9	-8.48710e-01	20	4.22674e+07
10	4.34355e+00		

如果对本例中的积分稍作分析，可以发现，相邻两个积分之间除了具有递推关系式(1.1)以外,还满足下列不等式

$$0 < I_n < I_{n-1} \tag{1.4}$$

由不等式(1.4)可知,I_{20}应比 I_0 小，且所有的积分值都不可能为负。因此，由递推算法(1.3)计算得到的结果（表 1.1)是不可靠的。

实际上，根据递推关系式(1.1)还可以得到另一个递推公式：

$$I_{n-1} = \frac{1}{5n} - \frac{1}{5}I_n \tag{1.5}$$

从而得到如下的递推算法：

$$\begin{cases} I_{30} = 1 \\ I_{n-1} = \dfrac{1}{5n} - \dfrac{1}{5}I_n, \quad n = 30,29,\cdots,1 \end{cases} \tag{1.6}$$

利用递推算法(1.6),用 C 语言编制程序，采用双精度运算，其计算结果（只取需要的21 个积分值)如表 1.2 所示。

表1.2 计算结果(2)

n	I_n	n	I_n
20	7.99762e-03	9	1.69265e-02
19	8.40048e-03	8	1.88369e-02
18	8.84622e-03	7	2.12326e-02
17	9.34187e-03	6	2.43249e-02
16	9.89633e-03	5	2.84684e-02
15	1.05207e-02	4	3.43063e-02
14	1.12292e-02	3	4.31387e-02
13	1.20399e-02	2	5.80389e-02
12	1.29766e-02	1	8.83922e-02
11	1.40713e-02	0	1.82322e-01
10	1.53676e-02		

显然,由递推算法(1.6)计算得到的结果(表 1.2)是正确的。由同一个递推关系式(1.1)得到的两个递推算法计算,为什么会有两种不同结果呢?

在递推算法(1.3)中,初值 I_0 是近似的,即实际上 I_0 存在一个误差;而在用递推算法(1.3)递推过程中,每递推计算一次,后一个值 I_n 的误差是前一个值 I_{n-1} 误差的 5 倍,因此,当计算到 I_{20} 时,其误差是初值 I_0 误差的 5^{20} 倍。

在递推算法(1.6)中,虽然初值 I_{30} 是近似的,而且其误差可能很大。但在用递推算法(1.6)递推过程中,每递推计算一次,后一个计算值 I_{n-1} 的误差是前一个计算值 I_n 误差的 1/5,因此,当计算到 I_{20} 时,其误差是初值 I_{30} 误差的 $1/5^{10}$,已经是微不足道了。当继续递推计算时,其误差还会越来越小,因此计算得到的 21 个积分值都是可靠的。

4. 递归

人们在解决一些复杂问题时,为了降低问题的复杂程度(如问题的规模等),一般总是将问题逐层分解,最后归结为一些最简单的问题。这种将问题逐层分解的过程,实际上并没有对问题进行求解,而只是当解决了最后那些最简单的问题后,再沿着原来分解的逆过程逐步进行综合,这就是递归的基本思想。由此可以看出,递归的基础也是归纳。在工程实际中,有许多问题就是用递归来定义的,数学中的许多函数也是用递归来定义的。递归在可计算性理论和算法设计中占有很重要的地位。

下面用一个简单的例子来说明递归的基本思想。

【例 1.3】编写一个过程,对于输入的参数 n,依次打印输出自然数 1 到 n。

这是一个很简单的问题,实际上不用递归就能解决,其算法如下:

【算法 1.3】输出自然数 1 到 n。

```
PROCEDURE WRT(n)
FOR k = 1 TO n DO OUTPUT k
RETURN
```

算法 1.3 用 C 语言描述如下:

```
#include "stdio.h"
wrt(int n)
{ int k;
  for (k = 1;k < = n;k + + ) printf("%d\n",k);
  return;
}
```

解决这个问题还可以用递归过程来实现,其算法如下:

【算法 1.4】输出自然数 1 到 n 的递归算法。

```
PROCEDURE WRT1(n)
IF (n≠0) THEN
   { WRT1(n - 1)
     OUTPUT n
   }
RETURN
```

在递归算法 WRT1 中，n 是形参。在开始执行算法 WRT1 时，首先要判断形参变量值(开始时为 n)是否不等于 0。如果不等于 0，则将形参值减 1(即 n－1)后作为新的实参再调用算法 WRT1。在调用函数 WRT1 时，又需判断形参值(此时已变为 n－1)是否不等于 0。如果不等于 0，则又将形参值减 1(即 n－2)后作为新的实参再次调用算法 WRT1……这个过程一直进行下去，直到算法 WRT1 的形参值等于 0 为止。此时，由于在先前各层的函数调用中，算法 WRT1 实际上没有执行完，即各层中的形参值还没有被打印输出，这就需要逐层返回，以便打印输出各层中的输入参数 1，2，…，n。为此，在递归算法的执行过程中，需要记忆各层调用中的参数，以便在逐层返回时恢复这些参数继续进行处理。具体来说，在算法 WRT1 开始执行后，随着各次的递归调用，逐次记忆各层调用中的输入参数 n，n－1，n－2，…，2，1 在逐层返回时，又依次(按记忆的相反次序)将这些参数打印输出。

算法 1.4 用 C 语言描述如下：

```
#include "stdio.h"
wrt1(int n)
{ if (n!=0)
    { wrt1(n-1); printf("%d\n",n);
  return;
}
```

在程序设计中，递归是一个很有用的工具。对于一些比较复杂的问题，设计成递归算法其结构清晰，可读性也强。

由上例可以看出，自己调用自己的过程称为递归调用过程。

递归分为直接递归与间接递归两种。如果一个算法 P 显式地调用自己则称为直接递归。例如算法 1.4 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，则称为间接递归调用。

递归是一种很重要的算法设计方法之一。实际上，递归过程能将一个复杂的问题归结为若干个较简单的问题，然后将这些较简单的每一个问题再归结为更简单的问题，这个过程可以一直做下去，直到最简单的问题为止。

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的。通常，递归算法要比递推算法清晰易读，其结构比较简练。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，但递归算法的执行效率比较低。

5. 减半递推技术

解决实际问题的复杂程度往往与问题的规模有着密切的关系。例如，两个 n 阶矩阵相乘，通常需要作 n^3 次乘法。两个二阶矩阵相乘需要作 8 次乘法。设两个二阶矩阵为

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

其乘积矩阵 C = AB 为

需要全本请访问 www.ertongbook.com

$$C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

能不能减少乘法次数呢？如果直接考虑一般的 n 阶矩阵相乘的问题，这是很困难的。但对于低阶的矩阵相乘问题，如二阶矩阵相乘，减少乘法次数是有可能的。实际上，若令

$$\begin{cases} x_1 = (a_{11} + a_{22})(b_{11} + b_{22}) \\ x_2 = (a_{21} + a_{22})b_{11} \\ x_3 = a_{11}(b_{12} - b_{22}) \\ x_4 = a_{22}(b_{21} - b_{11}) \\ x_5 = (a_{11} + a_{12})b_{22} \\ x_6 = (a_{21} - a_{11})(b_{11} + b_{12}) \\ x_7 = (a_{12} - a_{22})(b_{21} + b_{22}) \end{cases} \tag{1.7}$$

可以验证，乘积矩阵 C 中的各元素可用以上 7 个量的线性组合来表示，即

$$\begin{cases} c_{11} = x_1 + x_4 - x_5 + x_7 \\ c_{12} = x_3 + x_5 \\ c_{21} = x_2 + x_4 \\ c_{22} = x_1 + x_3 - x_2 + x_6 \end{cases} \tag{1.8}$$

由此可以看出，利用上述方法计算两个二阶矩阵相乘只需要 7 次乘法就够了，比通常的方法减少了 1 次乘法。由于在式(1.7)的计算中没有用到乘法的交换性（即矩阵 A 中的元素始终在前面，矩阵 B 中的元素始终在后面），因此，对于一般的 n 阶矩阵可以划分为 4 块 n/2 阶的矩阵，然后利用分块矩阵相乘，可以得到 7 个 n/2 阶矩阵相乘的问题。同样，对于每个 n/2 阶矩阵相乘的问题，又可以化为 7 个 n/4 阶矩阵相乘的问题。以此类推，最后可以归结为计算一阶矩阵相乘的问题。而一阶矩阵相乘只需要一次乘法。

根据以上分析，假设 $n=2^k$，且 $n=2^k$ 阶矩阵相乘所需要的乘法次数为 M(k)，则有

$$M(k) = 7M(k-1) = 7^2M(k-2) = \quad = 7^kM(0)$$

且 M(0) = 1。因此有

$$M(k) = 7^k = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.81}$$

显然，这个乘法次数比通常的 n^3 次要少得多，当 n 较大时更为显著。

算法设计的这种方法称为分治法，即对问题分而治之。工程上常用的分治法是减半递推技术。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’，是指将问题的规模减半，而问题的性质不变。所谓‘递推’，是指重复“减半”的过程。

下面举例说明利用减半递推技术设计算法的基本思想。

【例 1.4】设方程 f(x) = 0 在区间[a,b]上有实根，且 f(a)与 f(b) 异号。利用二分法求该方程在区间[a,b]上的一个实根。

二分法求方程实根的减半递推过程如下：

首先取给定区间的中点 c = (a + b)/2。

然后判断 f(c)是否为 0。若 f(c) = 0，则说明 c 即为所求的根，求解过程结束 如果

$f(c) \neq 0$,则根据以下原则将原区间减半：

若 $f(a)f(c) < 0$,则取原区间的前半部分 若 $f(b)f(c) < 0$,则取原区间的后半部分。

最后判断减半后的区间长度是否已经很小。若 $|a-b| < \varepsilon$,则过程结束,取 $(a+b)/2$ 为根的近似值 若 $|a-b| \geqslant \varepsilon$ ，则重复上述的减半过程。

其算法如下：

【算法 1.5】二分法求方程实根。

```
FUNCTION ROOT(a,b,eps,f)
f0 = f(a)
WHILE ( |a - b| ≥ε) DO
    { c = (a + b)/2
      f1 = f(c)
      IF (f1 = 0) THEN
         { ROOT = c
           RETURN
         }
      IF (f0 * f1 > 0) THEN a = c
      ELSE b = c
    }
c = (a + b)/2
ROOT = c
RETURN
```

算法 1.5 的 C 语言描述如下：

```
#include "stdio.h"
#include "math.h"
double root(a,b,eps,f)
double a,b,eps,( *f)();
{ double f0,f1,c;
  f0 = ( *f)(a);
  while (fabs(a - b) > = eps)
     { c = (a + b)/2; f1 = ( *f)(c);
       if (f1 = =0) return(c);
       if (f0 * f1 > 0) a = c;
       else b = c;
     }
  c = (a + b)/2;
  return(c);
```

6. 回溯法

前面讨论的递推和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。在工程上，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，找出一个解决问题的线索；然后沿着这个线索逐步试探。对于每一步的试探,若试探成功,就得到问题的解;若试探失败,就逐步回退,换别的路线再进行试探。这种方法称为回溯法。回溯法在处理复杂数据结构方面有着广泛的应用。

下面举例说明回溯法的基本思想。

【例 1.5】求解皇后问题。

由 n^2 个方块排成 n 行 n 列的正方形称为“n 元棋盘”。如果两个皇后位于棋盘上的同一行或同一列或同一对角线上，则称它们为互相攻击。现要求找使 n 元棋盘上的 n 个皇后互不攻击的所有布局。

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，将有攻击的布局剔除掉。这是一种列举法。但这种方法对于较大的 n，其运算量会急剧地增加。而在实际上，逐一列举也是没有必要的。例如，如果第一行上的皇后在第一列，则第二行上的皇后就不可能在第一列。

下面用回溯法来求解皇后问题。

假设棋盘上的每一行放置一个皇后，分别用自然数进行编号为 1,2,…,n。

首先定义一个长度为 n 的一维数组 q,其中每一个元素 $q[i](i=1,2,\cdots,n)$ 随时记录第 i 行上的皇后所在的列号。

初始时,先将各皇后放在各行的第 1 列。即数组 q 的初值为

$$q[i] = 1, i = 1,2,\cdots,n$$

第 i 行与第 j 行上的皇后在某一对角线上的条件为

$$|q[i] - q[j]| = |i - j|$$

而它们在同一列上的条件为

$$q[i] = q[j]$$

回溯法的步骤如下：

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，即它们均互不攻击。现在考虑安排第 i 行上的皇后的位置,使之与前 i-1 行上的皇后也都互不攻击。为了实现这一目的，可以从第 i 行皇后的当前位置 q[i] 开始按如下规则向右进行搜索：

(1) 若 q[i]=n+1,则说明第 i 个皇后暂时无法安排，将第 i 个皇后放在第 1 列(即置 q[i]=1),且回退一行,考虑重新安排第 i-1 行上的皇后与前 i-2 行上的皇后均互不攻击的下一个位置。此时如果已退到第 0 行(实际没有这一行),则过程结束。

(2) 若 q[i]≤n,则需检查第 i 行上的皇后与前 i-1 行的皇后是否互不攻击。若有攻击,则将第 i 行上的皇后右移一个位置（即置 q[i]=q[i]+1),重新进行这个过程;若无攻击,则考虑安排下一行上的皇后位置,即置 i=i+1。

(3) 若当前安排好的皇后是在最后一行（即第 n 行),则说明已经找到了 n 个皇后互不攻击的一个布局,将这个布局输出(即输出 q[i],i=1,2,…,n)。然后将第 n 行上的皇后右移一个位置（即置 q[n]=q[n]+1),重新进行这个过程,以便寻找下一个布局。

下面是用回溯法求解皇后问题的算法。

【算法 1.6】回溯法求解皇后问题。

```
PROCEDURE QUEEN(n)
定义长度为 n 的数组存储空间 q
FOR i = 1 TO n DO q[i] = 1
i = 1
loop:
IF( q[i] ≤ n) THEN
  { k = 1
    WHILE ((k < i) and((q[k] - q[i]) * ( |q[k] - q[i]| - |k - i|)) ≠ 0)
        DO k = k + 1
    IF (k < i) THEN q[i] = q[i] + 1
    ELSE
      { i = i + 1
        IF (i > n) THEN
          { OUTPUT q[i](i = 1,2,…,n)
            q[n] = q[n] + 1
            i = n
          }
      }
  }
ELSE
  { q[i] = 1
    i = i - 1
    IF (i < 1) THEN RETURN
    q[i] = q[i] + 1
  }
GOTO loop
RETURN
```

算法 1.6 的 C 语言描述如下(C 语言中的数组元素下标从 0 开始):

```
#include "stdio.h"
#include "math.h"
#include "stdlib.h"
void queen(int n)
{ int i,j,k,jt, *q;
  q = malloc(n * sizeof(int));
  for (i = 0; i < n; i + +) q[i] = 0;
  i = 0; jt = 1;
  printf("\n");
  printf("%d queen problem\n",n);
```

```
while(jt = =1)
  { if (q[i] < n)
      { k =0;
        while ((k < i)&&((q[k] - q[i]) *
              (fabs(q[k] - q[i]) - fabs(k - i)))! =0) k = k + 1;
        if (k < i) q[i] = q[i] + 1;
        else
          { i = i + 1;
            if (i > n - 1)
              { for (j =0; j < n; j + + ) printf("%5d",q[j] +1);
                printf("\n");
                q[n - 1] = q[n - 1] +1;
                i = n - 1;
              }
          }
      }
    else
      { q[i] =0; i = i - 1;
        if (i <0) { printf("\n"); free(q); return; }
        q[i] = q[i] +1;
      }
  }
```

以上介绍了几种工程上常用的算法设计的基本方法。实际上，算法设计的方法还有很多,如数字模拟法,用于数值近似的数值法等,在此不再一一介绍了,有兴趣的读者可参看有关算法方面的书籍。

1.4 算法的复杂度分析

算法的复杂度主要包括时间复杂度和空间复杂度。

1.4.1 算法的时间复杂度

所谓算法的时间复杂度，是指执行算法所需要的计算工作量。

为了能够比较客观地反映出一个算法的效率，在度量一个算法的工作量时，不仅应该与所使用的计算机、程序设计语言以及程序编制者无关，而且还应该与算法实现过程中的许多细节无关。为此，可以用算法在执行过程中所需基本运算的执行次数来度量算法的工作量。基本运算反映了算法运算的主要特征，因此，用基本运算的次数来度量算法工作量是客观的也是实际可行的，有利于比较同一问题的几种算法的优劣。例如，在考虑两个矩阵相乘时，可以将两个实数之间的乘法运算作为基本运算，而对于所用的加法（或减法）运算可以忽略不计。又如，当需要在一个表中进行查找时，可以将两个元素之间的比

较作为基本运算。

算法所执行的基本运算次数还与问题的规模有关。例如，两个 20 阶矩阵相乘与两个 10 阶矩阵相乘，所需要的基本运算(即两个实数的乘法)次数显然是不同的，前者需要更多的运算次数。因此，在分析算法的工作量时，还必须对问题的规模进行度量。

综上所述，算法的工作量用算法所执行的基本运算次数来度量，而算法所执行的基本运算次数是问题规模的函数，即

$$\text{算法的工作量} = f(n)$$

其中 n 是问题的规模。例如，两个 n 阶矩阵相乘所需要的基本运算（即两个实数的乘法）次数为 n^3，即计算工作量为 n^3，也就是时间复杂度为 n^3。

在具体分析一个算法的工作量时，还会存在这样的问题：对于一个固定的规模，算法所执行的基本运算次数还可能与特定的输入有关，而实际上又不可能将所有可能情况下算法所执行的基本运算次数都列举出来。例如“在长度为 n 的一维数组中查找值为 x 的元素”，若采用顺序搜索法，即从数组的第一个元素开始，逐个与被查值 x 进行比较。显然，如果第一个元素恰为 x，则只需要比较 1 次。但如果 x 为数组的最后一个元素，或者 x 不在数组中，则需要比较 n 次才能得到结果。因此，在这个问题的算法中，其基本运算(即比较)的次数与具体的被查值 x 有关。

在同一问题规模下，如果算法执行所需的基本运算次数取决于某一特定输入时，可以用以下两种方法来分析算法的工作量。

1. 平均性态(average behavior)

所谓平均性态分析，是指用各种特定输入下的基本运算次数的带权平均值来度量算法的工作量。

设 x 是所有可能输入中的某个特定输入，p(x)是 x 出现的概率（即输入为 x 的概率），t(x)是算法在输入为 x 时所执行的基本运算次数，则算法的平均性态定义为

$$A(n) = \sum_{x \in D_n} p(x)t(x)$$

其中 D_n 表示当规模为 n 时，算法执行时所有可能输入的集合。这个式子中的 t(x)可以通过分析算法来加以确定，而 p(x)必须由经验或用算法中有关的一些特定信息来加以确定，通常是不能解析地加以计算的。如果确定 p(x)比较困难，则会给平均性态的分析带来困难。

2. 最坏情况复杂性(worst-case complexity)

所谓最坏情况分析，是指在规模为 n 时，算法所执行的基本运算的最大次数。它定义为

$$W(n) = \max_{x \in D_n} \{t(x)\}$$

显然，W(n)的计算要比 A(n)的计算方便得多。由于 W(n)实际上是给出了算法工作量的一个上界，因此，它比 A(n)更具有实用价值。

下面通过一个例子来说明算法复杂度的平均性态分析与最坏情况分析。

【例 1.6】采用顺序搜索法，在长度为 n 的一维数组中查找值为 x 的元素。即从数组的第一个元素开始，逐个与被查值 x 进行比较。基本运算为 x 与数组元素的比较。

需要全本请访问 www.ertongbook.com