高等学校计算机专业规划教材

# 数值分析

陈增荣 高卫国 编



電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

高等学校计算机专业规划教材

# 数值分析

陈增荣　高卫国　编

電子工業出版社
Publishing House of Electronics Industry
北京·BEIJING

## 内 容 简 介

本书全面系统地介绍了数值分析的重要算法并简单介绍了浮点计算、舍入误差分析、范数、迭代法的收敛性和收敛速度、微分方程数值解法的基本思想和途径。这些基本原理在数值分析中具有重要地位。在介绍算法时尽量深入浅出，有意忽略复杂繁琐的理论证明和推导。介绍的算法主要包括以下六个方面：函数方程求解，线性代数方程组求解的直接法和迭代法，函数插值、曲线拟合和函数逼近，数值积分，常微分方程的数值解，FFT 变换及其在图像压缩中的应用。最后介绍了并行计算的基本概念和程序设计基础。

本书适合于计算机科学和电子工程等非计算数学专业的高年级本科生和低年级研究生作为教材或自学参考书。



**图书在版编目(CIP)数据**

数值分析/陈增荣，高卫国编．—北京：电子工业出版社，2002.5
高等学校计算机专业规划教材
ISBN 7-5053-7575-X

Ⅰ.数… Ⅱ.①陈…②高… Ⅲ.计算方法-高等学校-教材 Ⅳ.0241

中国版本图书馆 CIP 数据核字(2002)第 024735 号

责任编辑：陈晓莉
印 刷 者：北京牛山世兴印刷厂
出版发行：电子工业出版社 http://www.phei.com.cn
北京市海淀区万寿路 173 信箱 邮编 100036
经 销：各地新华书店
开 本：787×1092 1/16 印张：8 字数：205 千字
版 次：2002 年 5 月第 1 版 2002 年 5 月第 1 次印刷
印 数：5 000 册 定价：12.00 元

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系。
联系电话：(010)68279077

# 前　言

本书是作者在复旦大学多年的教学实践基础上写成的，适合于计算机科学专业和电子工程等非计算数学专业高年级本科生和低年级研究生作为教材或自学参考书使用。我们假定本书的读者已经具备高等数学的基础知识。

教学改革是“十五”规划的一个重要议题。为了适应当前多元化的教改方向，在本书编写过程中，作者查阅了大量的国内外资料和相关教材，在内容上注意新旧结合，既保留了传统的经典算法，又在一定程度上加入了一些较新的研究方向和成果。考虑到非数学专业学生的特点，我们有意忽略了部分复杂而且繁琐的理论证明，而更多地注重介绍算法及其最新进展信息。

全书按如下形式组织：第 1 章介绍了计算机中数的表示和算法的基本概念；第 2 章讲述了函数方程求根的方法；第 3 章比较全面地叙述了线性方程组求解的直接法和迭代法；第 4 和第 5 章分别介绍了数值插值、函数逼近和数值积分的基本算法；第 6 章列举了常微分方程数值求解的过程和实现算法；第 7 章系统介绍了 FFT 变换及其在图像压缩算法中的应用；第 8 章以 ScaLapack 为例描述了并行计算的一些基本概念和程序设计基础。

最后，我们特别要感谢曹志浩教授和陈晓嵘工程师。复旦大学的曹志浩教授在百忙中认真审阅了本书初稿，并提出了许多建设性的意见。另外，陈晓嵘工程师在本书的编写、排版过程中也给予了诸多建议和技术支持，他的无私帮助才使此书得以成稿。一并在此表示感谢。

本书的第 1 至 6 章由陈增荣编写，第 7、8 两章由高卫国编写。

由于作者水平有限，书中的不当之处，恳请读者批评指正。

作　者

2001 年 12 月

# 目　录

# 第 1 章　数的表示、浮点算术和数值算法的误差

要用计算机求解实际问题，关键是选择算法。选择算法时要考虑的主要因素是算法的计算复杂性。对数值计算问题，由于实数在计算机上不能精确表示，观测数据也可能与实际数据有差异，这就产生了误差。对这些数据的计算机运算还会引起误差积累。有些算法对误差不敏感，这些算法在有初始误差的情况下，仍能得到基本准确的计算解。但也有一些算法，初始数据的微小差异也许会引起最后计算解的严重失真。所以，在选择数值计算问题的求解算法时还得考虑误差对算法的影响，即算法是否数值稳定的。

## 1.1　数的表示和误差

在计算机上整数与浮点数的表示截然不同。整数与它的机内表示完全等价，但浮点数与它在机内的表示则往往会有差异。

### 1.1.1　整数的机内表示和范围

在任何计算机上，整数都应与它的机内表示完全等价。通常使用的整数形式有十进制、八进制和十六进制三种。在计算机上通常使用二进制。它的机内表示有三种形式：原码、反码和补码。它们都由符号位 $x_0$ 和数值位 $x_1\ x_2\cdots x_n$ 两部分组成。$x_0=0$ 表示正数，$x_0=1$ 表示负数。

1. 原码表示法

在原码表示法中数值位部分给出该整数的绝对值。计算机上整数大多使用固定长度表示。如 C 语言中整型(int)为 16 位，短整型(short int)为 8 位，长整型(long int)为 32 位。以短整型为例，0000 1101 表示 +13，1000 1110 表示 −14。

由于使用固定长度，机内表示给出的整数范围有限。如 8 位原码表示(只有 7 位数值位)的整数范围为 [−127，127]。

2. 反码表示法

各种高级程序设计语言中的整数都使用原码表示法。但原码进行加减运算时符号位不能与数值位一样参加运算。运算规则复杂，给硬件和指令系统的设计加重负担。因此计算机硬件通常只使用反码表示法或补码表示法。反码表示法中正数的反码表示与原码相同，但当 $x_0=1$ 时该整数的数值位取相反码，0 改为 1，1 改为 0。如 5 位反码，10100 表示 −1011，即 −11。01001 表示 +9。可以看出，反码表示的整数范围与原码表示范围相同。

3. 补码表示法

由于补码的加减运算的运算规则简单,因此计算机硬件大多采用补码表示法。正数的补码表示与原码相同。负数的补码是在反码的末位加 1。

注意,在原码表示法和反码表示法中 0 有 +0 和 -0 两种形式。两种表示法的 -0 分别是 1000 0000 和 1111 1111。它们不同于 +0 的表示形式 0000 0000。但在补码表示法中,+0 和 -0 的表示形式是相同的,都是 0000 0000。于是 1000 0000 可用来表示 -128,从而 8 位补码表示的整数范围为 [ -128,127 ]。

## 1.1.2 浮点数的机内表示和范围

为了能用有限的位数表示更大的范围,现代的计算机都提供浮点数这一形式。顾名思义,浮点数就是小数点不固定(浮动)的数。对十进制,这就是常见的科学表示法。如 C 语言中 23E3 表示 $23\times10^3$, -12E4 表示 $-12\times10^4$,3E-2 表示 $3\times10^{-2}$。计算机硬件使用二进制:$N=2^E*S$。$E$ 称为浮点数 $N$ 的阶码,$S$ 称为 $N$ 的尾数。阶码和尾数都有各自的符号位。阶码大多使用补码表示法,尾数大多使用原码表示法。浮点数的格式如下:

$$E_0E_1\cdots E_n \quad S_0S_1\cdots S_m$$

其中 $E_0$ 是阶符,$S_0$ 是尾数的符号位,它也是整个浮点数的符号位。尾数中的小数点位置固定,通常取在 $S_1$ 之前。如 $-2^3\times0.1011101$ 的浮点形式为 (设阶码 4 位,尾数 8 位):

阶码 0011 尾数 1101 1101

浮点数的小数点可以浮动。若阶码加 1,则尾数右移一位。若阶码减 1,则尾数左移一位。如上面的浮点数也可表示为

阶码 0100 尾数 1010 1110(1)

由于阶码和尾数的长度都固定,阶码加 1,尾数右移一位,原 $S_1$ 移至 $S_2$,新 $S_1$ 补充为 0,原 $S_m$ 则移到固定长度之外,被丢失,产生误差。

为了使数据尽可能准确,我们希望 $S_1$ 必须为 1。若 $S_1$ 为 0,则尾数左移一位,同时阶码减 1。若新的 $S_1$ 仍为 0,则重复此过程,直至 $S_1=1$。这一过程称为规格化。每次浮点数运算后,都要将运算结果规格化。浮点数的运算通常都会产生误差。如 C 和 C++ 语言中输入回显的浮点数,通常与再输出结果不同。譬如输入回显 -34.6,输出显示为 -34.599998。这是因为数据输入存放到变量中,再从变量输出,这期间经过了 10→2,2→10 的两次转换,其中包括多次浮点数运算,产生了误差。所以在编程时,不能对浮点数进行相等比较。

浮点数表示范围很大,如 32 位的整数原码表示范围为 $[-(2^{31}-1),\ 2^{31}-1]$。若浮点数采用 8 位阶码,24 位尾数,则 32 位浮点数的表示范围为 $[-2^{127}(1-2^{-23}),\ 2^{127}(1-2^{-23})]$,远远超过上述整数原码表示范围。而且浮点数还可表示小数。绝对值最小的小数是 $\pm2^{-128}\times1/2$ (尾数中仅 $S_1=1$。由于浮点数必须规格化,尾数最小的绝对值为 1/2)。浮点数的具体表示范围不仅与长度有关,还与表示格式有关。阶码越长,则表示范围越大。但尾数短了,数据的精度减小了。所以两者必须兼顾。在 C++ 中 32 位的 float 类型的表示范围上面已给出,相

当于 $[-3.4\times10^{38}, 3.4\times10^{38}]$。64 位的 double 类型的表示范围则为 $[-1.7\times10^{308}, 1.7\times10^{308}]$。而 80 位的 long double 类型的表示范围竟可达到 $[-1.1\times10^{4932}, 1.1\times10^{4932}]$。

高级语言中的十进制形式的小数和十进制指数形式的浮点数,在计算机硬件中都转化成如上所述的二进制形式的浮点数。

### 1.1.3 相对误差、绝对误差和精度

1. 误差和绝对误差

设 $x$ 的近似值为 $x'$,则 $x'-x$ 称为近似值 $x'$ 关于精确值 $x$ 的误差。误差的绝对值 $|x'-x|$ 称为绝对误差。

误差虽不可避免,但人们总希望计算结果能足够准确。这就需要估计误差,设法给出绝对误差的上界 $|x'-x|<\varepsilon$。此上界 $\varepsilon$ 称为近似值 $x'$ 的绝对误差限,它反映了近似值 $x'$ 的精确程度。

2. 有效数字和精度

将一个位数很多的数表示成一定的位数,通常采用四舍五入方式。如 $\pi=3.14159265\cdots$ 可表示成 3.1416 或 3.14 等。如

$$|\Delta x|=|x'-x|\leqslant 0.5\times10^{-k}$$

则称近似值 $x'$ 准确到第 $k$ 位,$x'$ 的误差限是第 $k$ 位的半个单位,$x'$ 中第一个非零数字开始到第 $k$ 位的所有数字都称为有效数字。设 $x'=\pm0.a_1a_2\ldots a_n\times10^m(a_1\neq0)$,$a_I(I=1,\cdots,n)$ 是 0~9 之间的自然数且 $-k=m-h$,$1\leqslant h\leqslant n$,则 $x'$ 有 $h$ 位有效数字,或称 $x'$ 有 $h$ 位精度。如 $|3.1416-\pi|\leqslant0.5\times10^{-4}$,$x'=0.31416\times10^1$,即 $k=4$,$m=1$,于是 $h=m+k=5$,故 $\pi$ 的近似值 3.1416 有 5 位精度。

3. 相对误差

近似值 $x'$ 的绝对误差 $|x'-x'|$ 不足以刻画它的精度。因为测量 1000 米和 1 米时虽然都发生了 1 厘米的误差,但两者的精度是大有区别的。这就需要引进相对误差的概念。它是绝对误差除以精确值 $|x'-x|/|x|$。若 $|x'-x|/x\leqslant\varepsilon$,则称 $\varepsilon$ 为近似值 $x'$ 的相对误差限。容易得到相对误差与有效数字之间的对应关系。

## 1.2 浮点算术

在数值计算过程中,不可避免地会产生各种误差。主要有以下两种。

1. 截断误差

许多数学公式包含极限过程,而计算机只能执行有限次运算,由此产生的误差称为截断误差。如 $e^x$ 可展开为幂级数形式:

$$e^x = 1 + x + x^2/(2!) + \cdots + x^n/(n!) + \cdots$$

但用计算机求值时，只能截取有限项

$$s_n(x) = 1 + x + x^2/(2!) + \cdots + x^n/(n!)$$

用部分和 $s_n(x)$作为 $e^x$ 的近似值，由 Taylor 余项定理，其截断误差为

$$e^x - s_n(x) = (x^{n+1}/(n+1)!)e^{\theta x}(0 < \theta < 1)$$

2. 舍入误差

实数通常包含无限小数。受计算机字长的限制，用计算机表示的数据必须舍入成给定的位数，这就引起舍入误差。每一步的舍入误差也许微不足道，但经过计算过程的传播和积累，舍入误差甚至可能会“淹没”真正的解。

浮点算术和算法的数值稳定性只讨论舍入误差的影响。

## 1.2.1 浮点四则运算的舍入误差分析

科学计算和工程计算大多采用浮点数计算，计算机中的非零浮点数总可表示为：

$$\pm 2^J \times W \quad \{0.5 \leqslant W < 1\}$$

式中，$J$ 为阶码，$W$ 为尾数。在有些计算机中，基数不是 2 而是 4 或 16，这时误差分析可类似进行。设尾数的长度为 $t$，则 $W = 0.d_1d_2\cdots d_t$，其中 $d_i(i = 1,2,\cdots,t)$为 0 或 1。通常要求 $d_1 = 1$。

由于 $t$ 总是有限的，因此一般实数 $x$ 在计算机中只能近似地表示。记 $x$ 的计算机表示为 $x_r$。若 $x = 0$，则 $x_r = 0$(一般是机器零，即尾数为 0，阶码为最小负数)；若 $x \neq 0$，则在运算器有附加位(即 $t+1$ 位)的计算机中舍入 $t+1$ 位得到 $|x - x_r| \leqslant 2^{J-t-1}$。因 $|x| \geqslant 2^{J-1}$，$|x_r| \geqslant 2^{J-1}$，故

$$|x - x_r| \leqslant |x| \times 2^{-t}, |x - x_r| \leqslant |x_r| \times 2^{-t},$$

它们可表示为：

$$x_r = x(1+\delta) \text{ 或 } x_r = x/(1+\delta) \quad (|\delta| \leqslant 2^{-t}) \tag{1.1}$$

设“$\oplus$”表示四则运算，则 $x_r \oplus y_r$ 一般不是规格化浮点数，所以舍入不一定是在 $t+1$ 位进行的。在有乘法指令和除法指令的计算机中，舍入在 $t+1$ 位进行，所以对乘法和除法成立关系式：

$$fl(x \oplus y) = (x \oplus y)(1+\delta)$$

或

$$fl(x \oplus y) = (x \oplus y)/(1+\delta) \quad (|\delta| \leqslant 2^{-t}) \tag{1.2}$$

式中 $fl$ 表示浮点计算。

对加减法，舍入一般不在 $t+1$ 位进行，所以上式不一定成立。如在无附加位(即切断)的情形：

$$2^1 \times 0.10\cdots0 - 2^0 \times 0.11\cdots1$$

先对阶，将 $0.11\cdots1$ 右移 1 位，丢弃 $t+1$ 位变成 $2^1 \times 0.01\cdots1$，相减后得

$$2^1 \times 0.0\cdots01 = 2^{2-t} \times 0.10\cdots0,$$

即为 $2^{1-t}$，而精确值应为 $1-(1-2^{-t})=2^{-t}$。若记

$$fl(x \pm y) = (x \pm y)(1+\delta)$$

则这里的 $\delta$ 不小于 $2^{-t}$，而是 1。不过在任何情况下总成立

$$fl(x \pm y) = x(1+\delta_1) \pm y(1+\delta_2) \quad (|\delta_i| < 2^{-t}, i=1,2) \tag{1.3}$$

对有附加位的情形，不失一般性，考察两同号数的加法和减法。记

$$x = 2^{J_1} \times W_1, \quad y = 2^{J_2} \times W_2$$

则 $x+y$ 在 $t+1$ 位舍入，所以对⊕为 + 的情形，(1.2)式成立。对减法，若 $J_1 - J_2 \geqslant 2$（$J_2 - J_1 \geqslant 2$ 的情形类似），则 $|x-y| \geqslant 2^{J_1-1}$，从而

$$|fl(x-y)-(x-y)| \leqslant 2^{J_1} \times 2^{-t-1} \leqslant |x-y| \times 2^{-t}$$

即

$$fl(x-y) = (x-y)(1+\delta) \quad [|\delta| \leqslant 2^{-t}]$$

若 $|J2 - J_1| \leqslant 1$，则 $fl(x-y) \equiv x-y$。由此可知，(1.2)式对减法也成立。

综上所说，在有附加位（舍入）的计算机中对任一四则运算⊕总成立(1.2)式。在无附加位（切断）的计算机中(1.2)式仅对乘除法成立，对加减法只满足(1.3)式。在大型机和巨型机中往往没有除法指令，这时需按 1.2.2 节的分析方法对除法算法进行误差分析。

## 1.2.2 常用浮点运算的舍入误差分析

由于在误差估计式中经常遇到形式为 $\prod_i (1+\delta_i)$ 的乘积，这里先对它作出估计。

**定理 1.1** 若 $|\delta_i| \leqslant 2^{-t}(i=1,\cdots,n)$，$n \times 2^{-t} \leqslant 0.01$，则当 $n>2$ 时成立：

$$1 - n \times 2^{-t} \leqslant \prod_{i=1}^{n}(1+\delta_i) \leqslant 1 + 1.01n \times 2^{-t} \tag{1.4a}$$

$$\prod_{i=1}^{n}(1+\delta_i) = 1 + 1.01n \times \theta \times 2^{-t} \quad (|\theta| \leqslant 1) \tag{1.4b}$$

证明　由假设 $|\delta_i| \leqslant 2^{-t}$，易得

$$(1-2^{-t})^n \leqslant \prod_{i=1}^{n}(1+\delta_i) \leqslant (1+2^{-t})^n \tag{1.5}$$

由 Taylor 展开知：

$$(1-x)^n = 1 - nx + n(n-1)(1-\theta x)^{n-2}x^2/2 \geqslant 1 - nx$$

$$(1-2^{-t})^n \geqslant 1 - n \times 2^{-t} \tag{1.6}$$

又当 $0 \leqslant x \leqslant 0.01$ 时利用 $e^x < 2$ 得

$$1 + x \leqslant e^x = 1 + x + (1 + x/3 + 2x^2/4! + \cdots)x^2/2$$

$$\leqslant 1 + x + 0.01x \times e^x/2 \leqslant 1 + 1.01x$$

令 $x=2^{-t}$,由上式的左端不等式 $1+x\leqslant e^x$ 得 $(1+2^{-t})^n\leqslant e^{n\times 2^{-t}}$,令 $x=n\times 2^{-t}$,由上式的右端不等式得 $e^{n2^{-t}}\leqslant 1+1.01n\times 2^{-t}$

$$(1+2^{-t})^n\leqslant 1+1.01n\times 2^{-t} \tag{1.7}$$

由(1.5)~(1.7)式立即得到定理的结论。

(1) 考察乘积 $p_n=fl(x_1x_2\cdots x_n)$,这里乘法次序与书写次序一致.容易看到:

$$p_1=x_1$$

$$p_r=fl(p_{r-1}x_r)\equiv p_{r-1}x_r(1+\delta_r)\cdot$$

$$=x_1\prod_{i=2}^{r}x_i(1+\delta_i)=(1+E_r)\prod_{i=1}^{r}x_i\quad(r=1,2,\cdots,n)$$

$$1-(r-1)2^{-t}\leqslant(1-2^{-t})^{r-1}$$

$$\leqslant 1+E_r\leqslant(1+2^{-t})^{r-1}\leqslant 1+1.01(r-1)2^{-t}$$

同理

$$fl(x_1x_2\cdots x_m/(y_1y_2\cdots y_n))\equiv(1+E)x_1x_2\cdots x_m/(y_1y_2\cdots y_n)$$

$$1-(m+n-1)2^{-t}\leqslant 1+E\leqslant 1+1.01(m+n-1)2^{-t}$$

只要 $m+n$ 不太大,相对误差总是小的。

(2) 考察求和 $s_n=fl(x_1+x_2+\cdots+x_n)$,这里相加次序也与书写次序相同。由(1.2)式易知

$$s_1=x_1$$

$$s_r=fl(s_{r-1}+x_r)\equiv(s_{r-1}+x_r)(1+\delta_r)$$

于是

$$s_n=\sum_{i=1}^{n}x_i(1+\eta_i),\quad 1+\eta_1=\prod_{j=2}^{n}(1+\delta_j)$$

$$1+\eta_i=\prod_{j=i}^{n}(1+\delta_j),\quad(i=2,3,\cdots,n)$$

与上面的乘积估计类似,也能得到估计式:

$$|\eta_i|\leqslant 1.01(n+1-i)2^{-t}\quad(i=2,3,\cdots,n)$$

因而 $s_n$ 有小的绝对误差 $O(n)2^{-t}\sum_{i=1}^{n}|x_i|$。若 $|x_i|<1$,则容易看到 $s_n$ 的绝对误差为 $O(n^2)2^{-t}$。但应注意,$s_n$ 的相对误差不一定很小。如果 $s_n$ 接近于 0,可以有很大的相对误差。对切断即无附加位的情形,可利用(1.3)式得到类似的结果。

上面两种特殊情形的误差估计方法容易推广到一般的数值计算,并由此建立各种数值算法的误差分析。

## 1.3 算法复杂性和数值算法的误差

数值分析的核心是对各类问题导出求解的算法,并对求解算法估计它的误差。求解算法的截断误差与各类具体问题有关。这里只讨论舍入误差对求解算法的影响,它适用于任何类型的问题。

### 1.3.1 什么是算法

算法可定义为对一类问题的有限的机械的判定(计算)过程。以下对此定义作进一步的说明。

先引进问题类的直观概念。如一元二次方程:

$$ax^2 + bx + c = 0 \quad (a \neq 0) \tag{1.8}$$

的求根。易知

$$x_{1,2} = (-b \pm (b^2 - 4ac)^{1/2})/(2a)$$

不论 $a(\neq 0)$、$b$ 和 $c$ 为何值,这一算法对(1.8)式总给出正确的解。这里给出的问题类是:一元两次方程(1.8)式的求解。而对具体给出的 $a(\neq 0)$、$b$ 和 $c$,则给出了这个类中的一个具体的问题。

上面给出的是非常简单的问题类。大多数问题类与规模 $N$ 有关。如方程

$$a_N x^N + a_{N-1} x^{n-1} + \cdots + a_1 x + a_0 = 0 \ (a_n \neq 0)$$

的求根就是规模为 $N$ 的问题类。另一个经常遇到的问题类是线代数方程组 $\boldsymbol{Ax} = \boldsymbol{b}$ 的求解,即要求解:

$$\begin{cases} a_{11}x_1 + \cdots + a_{1N}x_N = b_1 \\ a_{21}x_1 + \cdots + a_{2N}x_N = b_2 \\ \qquad\qquad \vdots \\ a_{N1}x_1 + \cdots + a_{NN}x_N = b_N \end{cases}$$

如果 $\det \boldsymbol{A} \neq 0$,则 $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$。这里线代数方程组的求解就是规模为 $N$ 的问题类。而一组特定的 $\boldsymbol{A}$ 和 $\boldsymbol{b}$,则给出了这个类中的一个具体的问题(实例)。这里给出的求解算法是针对问题类的。如 Gauss 列主元消去法就是用于"线代数方程组求解"这个问题类的。

一般在考察一个问题类时,人们感兴趣的是:有没有一种方法(一套规则),借助于这个方法(这套规则),可以对这个类中的任意一个问题在有限步内获得解答。假如有一种方法(一套规则)存在,那么称这方法(规则)为对于该问题类的一个算法。如果某问题类存在求解的算法,则此问题类称为可解的。

不少人常认为,详细的方法就是算法。其实不然,算法一定要对任何情形都能在有限步内给出解答。数学软件包(或标准程序)中必须使用算法。如对迭代法,必须加上迭代终止标准、迭代次数限制等,否则有可能出现死循环。

对任意一个线代数方程组,Gauss 列主元消去法能在有限步内判定无解、有无限个解或给出满足方程组的解,所以它是一个算法。下面是问题类和算法的另一些例子。

**【例 1.1】** 问题类:自然数 $m$ 与 $n$ 互质吗?

算法:辗转相除法。

**【例 1.2】** 问题类:自然数 $n$ 是质数吗?

算法:筛法。

不是所有的问题类都是有算法的。有些问题类是半可解的,即对此类中的某些问题,算法可能在有限步内终止,但对此类中的另一些问题则无法求解,即在任意有限步内得不到解答。

除上面两个问题类外，还有(绝对)不可解的问题类，即在计算机上无论用多长时间也无法精确求解。如 Hilbert 第十个问题："任意给定的整系数代数方程有整数解吗?"就是(绝对)不可解的问题类。半可解和不可解等课题属于可计算性的范畴，这里就不作进一步的讨论。

### 1.3.2 算法的计算复杂性

算法可以用各种标准来评价。但是，人们关心的主要是算法与它能求解的问题规模之间的关系，即随着问题的规模越来越大，此算法运行所需的时间和空间究竟会怎样变化? 这里将算法所需的时间看成问题规模的函数，并称之为算法的时间复杂性或称为时间复杂度(time complexity)。当规模 $N$ 趋于无穷大时，就得到渐近时间复杂性。一般，人们关心 $N$ 较大时的时间复杂性。这时它与渐近时间复杂性相差不多。有时，人们对这两者就不予区分。若一算法处理 $N$ 个输入需时间 $CN^2$，$C$ 是某常数，则称算法的时间复杂性为 $O(N^2)$，或称它的阶为 2。类似地可以定义空间复杂性和渐近空间复杂性。

正是算法的渐近复杂性完全确定了算法能求解的问题规模。为了使人有深刻的印象，这里观察五个算法，它们的时间复杂性分别为 $N$，$N\log_2 N$，$N^2$，$N^3$ 和 $2^N$。假设时间单位是 1 毫秒。各个算法随着 $N$ 的变化，所需时间的增长有巨大的差别。这从下面的表 1.1 可以看出。复杂性的阶越高，当 $N$ 增加时所需时间增长越快。复杂性的阶或多或少反映出一个计算问题的困难程度。前四个算法对大的 $N$，所需的时间都是可以满足的。因此它们在计算机上是易处理的。但对复杂性为 $2^N$ 的算法，$N=50$ 时已需要几万年的计算机时间，这显然是不允许的。所以它在计算机上是很难处理的。

**表 1.1 时间复杂性的影响**

| $N \setminus T \setminus$ 复杂性 | $N$ | $N\log_2 N$ | $N^2$ | $N^3$ | $2^N$ |
|---|---|---|---|---|---|
| 10 | 0.01 秒 | 0.033 秒 | 0.1 秒 | 1 秒 | 1.024 秒 |
| 50 | 0.05 秒 | 0.282 秒 | 2.5 秒 | 125 秒 | $3.57\times10^4$ 年 |
| 100 | 0.1 秒 | 0.664 秒 | 10 秒 | 16.67 分 | $4.0197\times10^{12}$ 年 |
| 500 | 0.5 秒 | 4.483 秒 | 4.167 分 | 34.72 小时 | … |
| 1000 | 1 秒 | 9.966 秒 | 16.67 分 | 277.8 小时 | … |

表中第 1 行第 1 列的 $N$ 指输入规模 $N$ 给出在第 1 列，复杂性表示各算法的时间复杂性给出在第 1 行，$T$ 表示表中其他单元给出的对应算法和给定 $N$ 所需的计算时间。一般认为，如果一个算法所需时间的上界为 $N$ 的多项式，则此算法求解的问题类是易处理的；否则就是不易处理的。时间界不能表示为 $N$ 的多项式的计算问题称为 $N$-$P$ 完全问题(nondeterministic polynomial-time complete problems)。在这类问题中包括组合数学的许多经典问题，如巡回售货员问题、Hamilton 回路问题、整数线性规划等。

如果更仔细地考察算法，那就不仅要注意复杂性的数量级(即阶的大小)，而且不能忽略主项系数(即具有最高阶的那一项的系数)，特别当 $N$ 较小时是如此。如果一个算法的时间复杂性阶高，但它有小的主项系数，此算法对小的 $N$ 还是很好的，甚至对人们感兴趣的问题规模都是如此。如时间复杂性分别为 $1000N$，$100N\log_2 N$，$10N^2$，$N^3$ 和 $2^N$ 的五个算法 $A_1 \sim A_5$，对 $2\leqslant N$

$\leqslant 9$，算法 $A_5$ 最好；仅当 $N>1024$ 时算法 $A_1$ 最好。

一个问题类可能有多个算法，这时应寻找计算复杂性最小（或较小）的算法。如 Fourier 变换的常规算法的时间复杂性为 $O(N^2)$，但若用快速 Fourier 变换 FFT，则其时间复杂性降为 $O(N\log_2 N)$。

### 1.3.3 向后误差分析和算法的数值稳定性

算法总是在计算机上运行的，而计算机的浮点计算一般总是不精确的。一个数学上精确的算法，经过大量的计算机计算，最后是否能得到基本正确的结果呢？在 J.H.Wilkinson 建立向后误差分析理论之前，这方面的结论是悲观的。正因为如此，在 20 世纪 50 年代对直接法的评价是否定的。只是在采用向后误差分析并引进病态和良态的概念之后，这一问题才获得解决。现在，向后误差分析已成为所有数值计算方法最常用的误差分析手段。

早期的误差分析方法是向前误差分析。设 $a_1,a_2,\cdots,a_n$ 是已知的量，它们或者是原始数据，或者是已计算好的量。对 $a_1,a_2,\cdots,a_n$ 施行四则运算（计算机上的算法只包含四则运算或逻辑运算）得到的结果为 $x$。现在将这种计算看成数学函数，即

$$x=g(a_1,a_2,\cdots,a_n) \tag{1.9}$$

向前误差分析关心的是计算机的计算值 $x'$ 与精确值 $x$ 之差，即估计

$$|x'-g(a_1,a_2,\cdots,a_n)|$$

的界。这通常是困难的，得到的界也是很差的。

向后误差分析并不关心每一步上计算值与精确值之差。恰恰相反，它指出由(1.9)式给出的计算值 $x'$ 正好等于 $a_i$ 扰动后 $g$ 的精确值，并给出这些扰动 $\varepsilon_i$ 的界。显然 $\varepsilon_i$ 不是惟一的，于是

$$x'\equiv g(a_1+\varepsilon_1,a_2+\varepsilon_2,\cdots,a_n+\varepsilon_n)=fl(g(a_1,a_2,\cdots,a_n)) \tag{1.10}$$

不过，向后误差分析是不完整的，因为人们最终关心的是问题的计算解与精确解之间的差异。所以在向后误差分析之后，人们还必须估计这个差。以线代数方程组求解为例，用列主元消去法求解 $\boldsymbol{Ax}=\boldsymbol{b}$，$\boldsymbol{x}$ 的计算解 $\boldsymbol{x}'$ 将是 $(\boldsymbol{A}+\delta\boldsymbol{A})\boldsymbol{x}'=\boldsymbol{b}$ 的精确解。这里扰动矩阵 $\delta\boldsymbol{A}$ 的元素值相对很小。对良态矩阵 $\boldsymbol{A}$，$\boldsymbol{x}$ 与 $\boldsymbol{x}'$ 差别很小。但当 $\boldsymbol{A}$ 为病态矩阵时，尽管 $\delta\boldsymbol{A}$ 的元素绝对值很小，但 $\boldsymbol{x}$ 与 $\boldsymbol{x}'$ 的差别有可能非常大。差异大是问题本身造成的，而不是算法引起的。只要计算结果是初始数据小扰动后的精确解，人们就称算法是数值稳定的。显然对良态问题，数值稳定算法能得到很好的结果，但对病态问题则不然。

通常，如果一个问题类的算法不增加奇点或奇点的阶，则此算法是数值稳定的。譬如不选主元的消去法，如果 $a_{11}=0$，则算法（不选主元的消去法）无法进行，$a_{11}=0$ 成了该算法的奇点。但 $a_{11}=0$ 不是问题类的奇点，所以此算法对问题类而言增加了奇点。由此，该算法数值不稳定。这也是为什么 Gauss 消去法要选主元的根本原因。是否增加奇点或奇性是判断一个算法是否数值稳定的最简捷最有效的手段。

迭代法的误差分析超出本书范围，这里不予讨论。