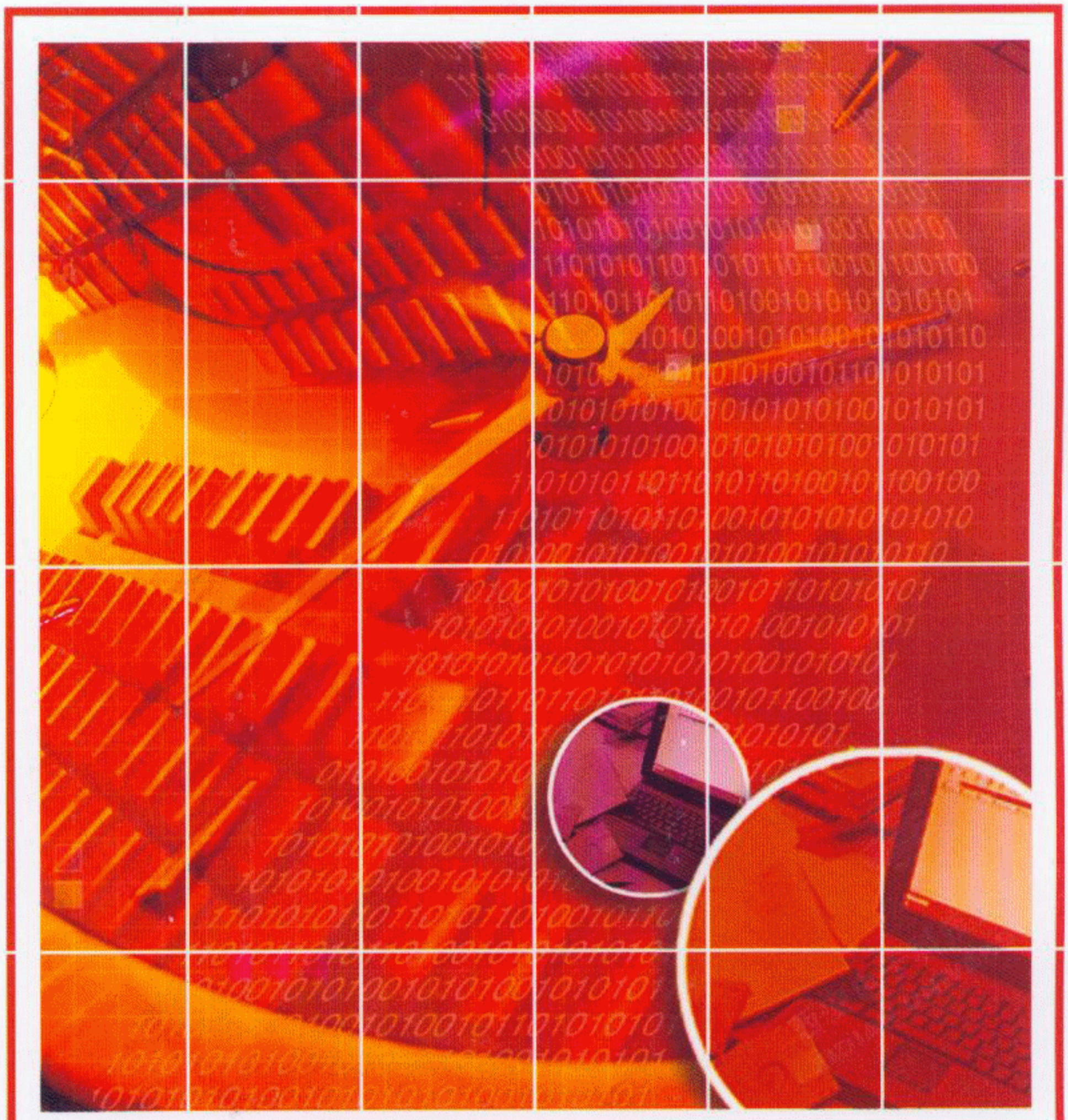
新世纪计算机类本科系列教材



算法设计与分析

霍红卫　编著

西安电子科技大学出版社
http://www.xduph.com

新世纪计算机类本科系列教材

算法设计与分析

霍红卫　编著

西安电子科技大学出版社

2005

内 容 简 介

本书系统地介绍了算法设计与分析的基本内容，并对讨论的算法进行了详尽分析。全书共7章，内容包括算法基础、基本算法设计和分析技术(递归和分治法、动态规划、贪心法、回溯法和分枝限界法)，以及NP完全性理论。书中以类高级程序设计语言对算法所做的简明描述，使得稍微具有程序设计语言知识的人即可读懂。此外，书中以大量图例说明每个算法的工作过程，使得算法更加易于理解和掌握。

本书可作为高等院校与计算机相关的各专业“算法设计”课程的教材，也可作为计算机领域的相关科研人员的参考书。此外，本书也可供参加ACM程序设计大赛的算法爱好者参考。

图书在版编目(CIP)数据

算法设计与分析/霍红卫编著．—西安：西安电子科技大学出版社，2005.2
(新世纪计算机类本科系列教材)
ISBN 7-5606-1492-2
Ⅰ.算… Ⅱ.霍… Ⅲ.①电子计算机-算法设计-高等学校-教材 ②电子计算机-算法分析-高等学校-教材 Ⅳ.TP301.6
中国版本图书馆CIP数据核字(2005)第004361号

策　　划　臧延新
责任编辑　龙　晖　臧延新
出版发行　西安电子科技大学出版社(西安市太白南路2号)
电　　话　(029)88242885　88201467　　邮　编　710071
http://www.xduph.com　　E-mail:xdupfxb@pub.xaonline.com
经　　销　新华书店
印刷单位　中铁一局印刷厂
版　　次　2005年2月第1版　2005年2月第1次印刷
开　　本　787毫米×1092毫米　1/16　印　张13.25
字　　数　310千字
印　　数　1～4000册
定　　价　15.00元
ISBN 7-5606-1492-2/TP·0794(课)
XDUP 1763001-1

前　言

算法研究是计算机科学研究的核心领域之一。在过去的半个世纪中，算法研究领域取得了大量重要的突破。这些突破引起了人们对算法研究的浓厚兴趣。同时，也使算法的应用领域不断扩大。从天体物理学中的 N 体问题的模拟到分子生物学中的序列分析，从排版系统到数据压缩，从数据库系统到 Internet 搜索引擎，算法在其中起着至关重要的作用，已经成为现代软件系统重要的组成部分。

全书分为三大部分：算法基础(第 1 章)、基本算法设计和分析技术(第 2～6 章)，以及 NP 完全性理论(第 7 章)。书中较全面地阐述了算法设计与分析方面的诸多理论和实践。本书内容安排如下：

· 第一部分介绍算法的基本概念和渐近表示，函数增长的数量级，证明算法正确性的循环不变式。

· 第二部分讨论递归和分治法、动态规划、贪心法、回溯法和分枝限界法。在分治法中，阐述了递归、递归方程和分治算法的关系，讨论了求解一般递归方程的三种方法。所给出的分治法应用实例包括经典问题(找最大值和最小值、矩阵相乘及整数相乘)、排序问题(归并排序、快速排序)、选择问题和最近点对问题。在动态规划算法中，分别介绍了自顶向下与自底向上的动态规划方法，深入地分析了设计一个动态规划算法时，问题自身所应具有的最优子结构和重叠子问题的性质，给出了动态规划算法的应用实例。在贪心算法中，分析了贪心算法所具有的基本元素，讨论了贪心算法在调度问题、文本压缩和网络算法中的应用。在回溯法和分枝限界法中，讨论了算法的设计思想及其在典型问题中的应用。

· 第三部分以深入浅出的方式，介绍了 NP 完全性理论，引入了 P 类问题和 NP 类问题的定义。通过网络路由器最优配置问题、网络服务器带宽优化问题和 Internet 网站多次抽签拍卖问题这些现实中的具体问题，来说明我们为什么要研究 NP 完全问题。同时，还给出了许多重要的 NP 完全问题的实例。

本书以类高级程序设计语言对算法进行简明描述，使得稍微具有程序设计语言知识的人即可读懂。另外，本书还以大量图例说明每个算法的工作过程，使得算法更加易于理解和掌握。

本书适合作为高等院校与计算机相关的各专业“算法设计”课程的教材，同时也可作为计算机领域的相关科研人员的参考书。此外，本书也可供参加 ACM 程序设计大赛的算法爱好者参考。

感谢西安电子科技大学出版社对于本书的出版给予的支持。

由于时间仓促及作者水平有限，书中难免有错误及不妥之处，希望读者批评指正。

作　者

2004 年 12 月

目　录

第1章 算法基础

1.1 算 法

算法(algorithm)可以被定义为一个良定的计算过程，它具有一个或者若干输入值，并产生一个或者若干输出值。因此，算法是由将输入转换成输出的计算步骤所组成的序列。

也可将算法看作是解决良定计算问题的工具。人们采用一般术语陈述问题，确定输入/输出关系，而算法则是描述这种输入/输出关系的特定计算过程。

例如，把 n 个元素排成非降序列，是实际中常见的一个问题。定义排序问题如下：

输入：n 个元素组成的序列$\langle a_1, a_2, \cdots, a_n \rangle$。

输出：重排输入序列之后，输出$\langle a_1', a_2', \cdots, a_n' \rangle$，其中元素满足 $a_1' \leqslant a_2' \leqslant \cdots \leqslant a_n'$。

给定一个输入序列$\langle 2, 10, 5, 4, 11 \rangle$，由排序算法返回的结果序列为$\langle 2, 4, 5, 10, 11 \rangle$。这样的输入序列称为问题的一个实例。一般而言，问题的实例由计算问题的一个解所需的输入组成。

若对每一个输入实例，算法都能终止并给出正确输出，则称这个算法是正确的。我们称这个正确算法解决了给定的计算问题。对于某些输入实例，一个不正确的算法可能根本不能终止，也可能终止，但输出不是问题的所需结果。与人们的想法相反，如果能够控制算法中的出错率，有时一个不正确的算法也是有用的。通常，我们只考虑正确的算法。

在本书中，我们用类 C 的伪代码表示算法。伪代码可以引用任何具有表达能力的方法来清晰、简洁地表达一个算法。因此，这里的伪代码不太考虑软件工程中的一些问题。为了更突出地表达算法自身的特性，在伪代码中常常忽略数据抽象、模块性、出错处理等问题。

1.1.1 冒泡排序

冒泡排序(bubble sort)属于基于交换思想的排序方法。它将相邻的两个元素加以比较，若左边元素值大于右边元素值，则将这两个元素交换；若左边元素值小于等于右边元素值，则这两个元素位置不变。右边元素继续和下一个元素进行比较，重复这个过程，直到比较到最后一个元素为止。

冒泡排序的伪代码用过程 BUBBLE-SORT 表示，其参数为包含 n 个待排序数的数组 $A[1..n]$。当过程 BUBBLE-SORT 结束时，数组 A 中包含已排序的序列。

```
BUBBLE-SORT(A)
1    for i ← 1 to length[A]
```

```
2        do for j ← length[A] downto i + 1
3            do if A[j] < A[j − 1]
4                then exchange A[j] ↔ A[j − 1]
```

图 1－1 说明了输入实例为 $A = \langle 5, 2, 4, 6, 1, 3 \rangle$ 时，算法 BUBBLE-SORT 的工作过程。对于外层 for 循环的每一次迭代，则在 $A[i]$ 位置产生当前元素比较范围 $A[i..n]$ 内的一个最小值。下标 i 从数组第一个元素开始，从左向右直至数组中最后一个元素。深色阴影部分表示数组元素 $A[1..i]$ 构成已排好的序列，浅色阴影部分表示外层循环开始时的下标 i。数组元素 $A[i+1..n]$ 表示当前正在处理的序列。

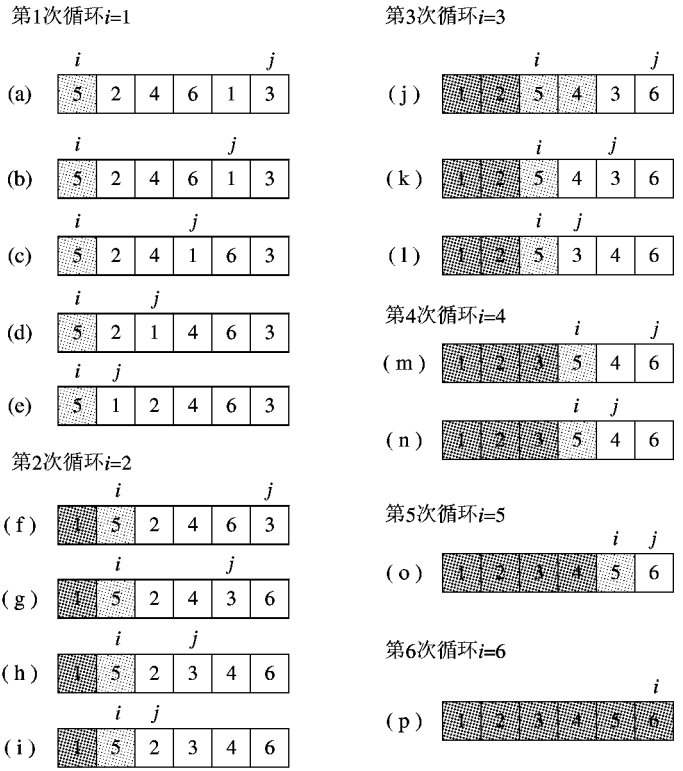


图 1－1　冒泡排序工作过程

1.1.2　循环不变式和冒泡排序算法的正确性

在内层 for 循环的每一次迭代的开始，$A[i]$ 是 $A[i..length[A]]$ 中的最小元素。在外层 for 循环每一次迭代的开始，子数组 $A[1..i-1]$ 中的元素有序。形式上我们称这些性质为循环不变式(loop invariant)。

我们可以利用循环不变式证明算法的正确性。循环不变式具有以下三个性质：

- 初始(initialization)：在循环的第一次迭代之前，循环不变式为真。
- 维持(maintenance)：如果在循环的某次迭代之前循环不变式为真，那么在下一次迭代之前，循环不变式仍然为真。
- 终止(termination)：当循环终止时，循环不变式给出有用性质，这个性质可以用于证明算法的正确性。

当循环不变式的前两个性质成立时，循环不变式在循环的每次迭代之前为真。循环不变式的证明与数学归纳法的证明类似，当证明某一条性质成立时，首先需证归纳基础，然后是归纳步。证明第一次迭代之前循环不变式成立，就像是证明归纳基础，证明从一次迭代到下一次迭代的不变式成立就像是归纳步。

循环不变式的第三个性质是最重要的性质，因为我们要利用循环不变式证明算法的正确性。同时，它与使用数学归纳法的不同之处在于，在数学归纳法中，可无限次利用归纳步。而在这里，当循环终止时，就停止“归纳”。

下面，我们考察这些性质是如何对冒泡排序成立的。首先证明内层 for 循环的不变式。

· 循环不变式：$A[j]$是 $A[j..length[A]]$中的最小元素。

· 初始：在内循环第一次开始迭代之前，$j = length[A]$，因此，子数组 $A[length[A]..length[A]]$ 中只包含一个元素，也即子数组中的最小元素，此时，循环不变式成立。

· 维持：假定在内循环的某次迭代之前循环不变式为真，即 $A[j]$是 $A[j..length[A]]$中的最小元素。在下一次迭代之前，若元素 $A[j]<A[j-1]$，则执行第 4 行语句，$A[j]$与$A[j-1]$ 交换，于是 $A[j-1]$是 $A[j-1..length[A]]$中的最小元素；若 $A[j]\geqslant A[j-1]$，那么第 4 行语句不执行，$A[j-1]$仍然是 $A[j-1..length[A]]$中的最小元素。无论哪一种情况，都能使循环不变式为真。

· 终止：对于冒泡排序，当 j 小于 $i+1$，即 $j=i$ 时，内层 for 循环结束。在内循环不变式中，用 i 代替 j，可得子数组 $A[i..length[A]]$，其中 $A[i]$是最小元素。

其次，证明外层 for 循环的不变式。

· 循环不变式：在 1～4 行外层 for 循环的每次迭代开始，子数组 $A[1..i-1]$中的元素有序。

· 初始：在外层 for 循环的第一次迭代之前，$i=1$，因此，$A[1..0]$为空，循环不变式成立。

· 维持：假定在外循环的某次迭代之前循环不变式为真，即子数组 $A[1..i-1]$中的元素有序。在下一次迭代之前，由内循环的不变式可得 $A[i-1]\leqslant A[i]$成立，因此，子数组 $A[1..i]$中的元素有序。

· 终止：当 i 大于 $length[A]$，即 $i = length[A] + 1$ 时，外层 for 循环结束。在外循环不变式中，用 $length[A]+1$ 代替 i，可得子数组 $A[1..length[A]]$。由循环不变式得，子数组有序，而这个子数组就是整个数组。因此，整个数组有序，这表明冒泡排序算法是正确的。

1.1.3 伪代码使用约定

在本书的伪代码(pseudocode)中使用以下约定：

(1) 缩进形式表示块结构。例如，BUBBLE-SORT 算法第 1 行开始的 for 循环体包括第 2～4 行。这种缩进风格也适用于 if-then-else 语句。用缩进形式代替传统块结构，如 begin 和 end 的表示形式，可大大减少代码的混乱，提高代码的清晰度。

(2) while，do-while，for 循环结构，以及 if-then-else 条件结构采用类似于高级语言中的相应表示。

(3) 符号“//”后面是注释部分。

(4) 多重赋值 $i \leftarrow j \leftarrow e$ 是将表达式 e 的值赋给变量 i 和变量 j，这种赋值与 $i \leftarrow e$ 和 $j \leftarrow e$ 等价。

(5) 变量如 i，j 和 key 是给定过程的局部变量。不经显式说明，不使用全局变量。

(6) 通过数组名后跟下标访问数组元素。例如，$A[i]$表示数组的第 i 个元素，符号“..”表示数组中元素值的范围。因此，$A[1..j]$表示 j 个元素 $A[1]$，$A[2]$，…，$A[j]$组成的子数组。

(7) 复合数据可以组织成由属性或域组成的对象。通过域名后跟方括号括住的对象名访问某个特定域。例如，可把数组看作属性为 $length$ 的对象，其中 $length$ 表示数组中包含的元素个数，记为 $length[A]$。尽管方括号既用作数组下标，又用作对象属性，但从上下文中就可了解其含义。

(8) 通过传值将参数传给一个过程。被调用的过程接收参数的一个复制，如果它对某个参数赋值，则调用过程是看不到这种变化的。当传递一个对象时，只是拷贝指向表示对象的数据的指针，不拷贝它的各个域。例如，x 是一个被调用过程的参数，在被调用过程内的赋值 $x \leftarrow y$ 对于调用过程而言是不可见的。赋值 $f[x] \leftarrow 3$ 是可见的。

(9) “and”和“or”是布尔运算符。当对表达式“x and y”求值时，首先计算 x 的值，如果值为 FALSE，则整个表达式的值为 FALSE，我们也无需计算 y 的值；如果 x 的值为 TRUE，则必须计算 y 的值，这样才能决定整个表达式的值。类似地，当对表达式“x or y”求值时，仅当 x 的值为 FALSE 时，才需计算表达式 y 的值。

(10) break 语句表示将控制转移到含有 break 的最内层循环语句后面的第一条语句。循环语句可以是约定(2)中所列的那些循环语句。

1.2 算法分析

算法分析是指对一个算法所需的计算资源进行预测。最重要的计算资源是时间和空间资源(存储器)，此外，还有通信带宽等，但我们常常最想要测量的是算法的计算时间。一般而言，通过分析问题的几个候选算法，可以确定一个最有效的算法。

在分析一个算法之前，必须建立实现技术所用的模型，包括该技术的资源及其开销的模型。在本书中，主要采用一个处理器的随机存取模型(RAM)作为计算模型。基于这种模型可以实现我们的技术，并将算法的实现理解为计算机程序。在 RAM 模型中，指令逐条执行，没有并发操作。

在算法分析中，即使分析 RAM 模型的一个简单算法也可能具有挑战性。所需的数学工具包括组合数学、概率论、代数学，此外，还需具有确定公式中最重要项的能力。由于算法的行为可能由于每次输入的不同而不同，因而，我们需要一些手段，能够用简单、易于理解的公式概述算法的这个行为。

即使选择了一种典型的机器模型分析给定算法，在决定如何表示分析时仍然会面临多种选择。我们希望找到一种方式，它易于书写、操纵，能够表达算法资源要求的一些重要特性，同时可略去一些繁琐细节。

1.2.1 冒泡排序算法分析

BUBBLE-SORT 过程的时间开销与输入有关：1000 个元素排序的时间要比 10 个元素的排序时间长。此外，即使对两个具有相同元素个数的序列排序，时间也可能不同，这取决于它们已排序的程度。一般而言，算法所需时间是与输入规模同步增长的，因而传统上将一个程序的运行时间表示为输入规模的函数。为此，需要更仔细地定义“输入规模”和“运行时间”的概念。

输入规模的概念取决于所研究的问题。对于许多问题，如查找、排序问题，最自然的度量是输入元素的个数，即待排序数组的大小 n。而对另外一些问题，如两个整数相乘，输入规模的最佳度量是用二进制表示的输入的总位数。有时，用两个数表示输入更合适。例如，某个算法的输入是一个图，则可用图中顶点数和边数表示输入。在以下研究的每个问题中，我们都将明确所用输入规模的度量标准。

一个算法的运行时间是指在某个输入时，算法执行基本操作的次数或者步数。我们尽可能独立于机器定义算法的基本操作，这样做便于进行算法分析。暂且采用以下观点：执行每行伪代码需要常量时间。尽管执行每一行代码所花费的时间可能不同，但我们假设第 i 行执行所花费的时间为常量 c_i。这个观点与 RAM 模型一致，它反映了伪代码是如何在实际的计算机上实现的。

在下面的讨论中，我们先给出 BUBBLE-SORT 过程中每一条语句的执行时间开销以及执行次数。设 $n=length[A]$，t_i 为第 4 行执行的次数。假定注释部分是不可执行的语句，不占运行时间。

BUBBLE-SORT(A)	开销	次数
1 **for** $i \leftarrow 1$ **to** $length[A]$	c_1	$n+1$
2 **do for** $j \leftarrow length[A]$ **downto** $i+1$	c_1	$\sum_{i=1}^{n}(n-i+1)$
3 **do if** $A[j] < A[j-1]$	c_2	$\sum_{i=1}^{n}(n-i)$
4 **then** exchange $A[j] \leftrightarrow A[j-1]$	c_3	t_i

该算法的总运行时间是每一条语句的执行时间之和，若执行一条语句的开销为 c_i，共执行了 n 次这条语句，则它在总的运行时间中占 $c_i n$。对每一对开销和执行时间乘积求和，可得算法的运行时间 $T(n)$为

$$\begin{aligned} T(n) &= c_1(n+1) + c_1\left(\sum_{i=1}^{n}(n-i+1)\right) + c_2\left(\sum_{i=1}^{n}(n-i)\right) + c_3 t_i \\ &= c_1(n+1) + c_1\frac{n(n+1)}{2} + c_2\frac{n(n-1)}{2} + c_3 t_i \\ &= \frac{(c_1+c_2)}{2}n^2 + \frac{(3c_1-c_2)}{2}n + c_1 + c_3 t_i \end{aligned} \tag{1.1}$$

即使给定问题规模，算法的运行时间也不能完全确定。例如，在冒泡排序算法中，当输入数组正好为升序时，出现最佳情况(best-case)。第 4 行语句不执行，即 $t_i=0$，式(1.1)最后一项为 0。此时，运行时间为

$$T(n)=\frac{(c_1+c_2)}{2}n^2+\frac{(3c_1-c_2)}{2}n+c_1$$

当输入数组为降序时，则出现最坏情况(worst-case)。此时，第 4 行语句执行次数为 $\sum_{i=1}^{n}(n-i)$，即 $t_i=n(n-1)/2$。此时，运行时间为

$$T(n)=\frac{(c_1+c_2+c_3)}{2}n^2+\frac{(3c_1-c_2-c_3)}{2}n+c_1$$

这时 $T(n)$也可写成 an^2+bn+c，常量 a、b 和 c 与 c_i 有关，这是 n 的一个二次函数。

1.2.2 最坏情况和平均情况分析

在冒泡排序算法中，我们分析了算法最佳情况和最坏情况下的性能。在本书的后续章节中，主要考虑算法最坏情况下的运行时间，即对于规模 n 的任何输入，算法运行最长的时间。之所以这样，是由于以下三个原因：

· 算法的最坏情况运行时间是任一输入运行时间的上界。由此可知，算法的运行时间不会比这更长。我们无需对算法的运行时间做出有根据的推测，并希望它不会变得更坏。

· 对于某些算法，最坏情况经常出现。例如，在数据库中搜索某个信息时，若搜索的信息不在数据库中，搜索算法就会出现最坏情况。在某些搜索应用中，搜索不存在的信息是经常有的。

· 算法的“平均情况”性能常常与最坏情况大致相同。上述冒泡排序算法的平均情况与最坏情况具有相同数量级。

在某些情况下，我们感兴趣的是算法的平均情况(average-case)或期望运行时间。当进行平均情况分析时，出现的问题是什么构成这个问题的平均输入。通常，我们假设已知规模的所有输入出现机率相等，实际中，可能会违背这个假定。有时，对于进行随机选择的随机算法，需进行概率分析。

1.2.3 增长的数量级

为了更容易地分析算法，需对公式进行简化。首先，忽略每条语句的实际开销，利用常量 c_i 表示这些开销。其次，公式 an^2+bn+c 中的常数给出了我们并不需要的详细信息，这些常数 a、b 和 c 依赖于 c_i。因此，我们可以忽略实际语句开销，也可以忽略抽象开销 c_i。

我们可以对公式做进一步简化，只对运行时间增长的数量级(order of growth)感兴趣。当 n 很大时，相对于公式中的首项(leading term，最高阶项)，如 an^2，低阶项可以忽略。另外，还可以忽略最高阶项的系数，因为对于较大规模的输入而言，与计算效率的增长相比，系数也是可以忽略的。因此，冒泡排序最坏情况下的运行时间为 $\Theta(n^2)$。(符号 Θ 的定义在下一节中给出。)

如果一个算法最坏情况运行时间的数量级比另一个算法的数量级要低，则认为该算法更有效。由于忽略了常数因子和低阶项，这样的评价对于输入规模较小的问题会产生错误。但对于足够大的输入规模 n，一个最坏情况下数量级为 $\Theta(n^2)$ 的算法要比数量级为 $\Theta(n^3)$ 的算法运行快。

1.3 算法的运行时间

1.3.1 函数增长

大部分算法都有一个主要参数 n，它是影响算法运行时间最主要的因素。这个参数可以是多项式的指数、待查找文件的大小或其他对于问题规模的抽象度量。为解决同一个问题所设计的各种算法在效率上会有很大差异，这些差异可能比个人微型计算机与巨型计算机之间的差异还大。例如，一台巨型机进行冒泡排序，另一台微型计算机进行归并排序，它们的输入都是一个规模为 100 万的有序数组。假设巨型机每秒执行 1 亿条指令，微型机每秒执行百万条指令。假如，一个优秀的程序员用机器代码在巨型机上实现冒泡排序，编出的程序需要执行 $2n^2$ 条指令来对 n 个数进行排序；另一个一般的程序员在微型机上用高级语言实现归并排序算法，产生的代码为 $50\ n\ \mathrm{lb}\ n$[①] 条指令。为排序 100 万个数，巨型机需要的时间为

$$\frac{2\times(10^6)^2\ \text{条指令}}{10^8\ \text{条指令}/\text{秒}}=20\ 000\ \text{秒}=5.56\ \text{小时}$$

微型机需要的时间为

$$\frac{50\times10^6\times\mathrm{lb}\ 10^6\ \text{条指令}}{10^6\ \text{条指令}/\text{秒}}\approx1000\ \text{秒}=16.67\ \text{分}$$

由上述比较可以看出，由于采用了更低阶的算法，即使用低效的编译器，微型机还是比巨型机快 20 倍。上述例子说明，数量级的改进可对算法效率产生重要影响。

表 1－1 列举了算法分析中一些常见函数的相对大小，说明快的算法比快的计算机更能帮助我们在可以忍受的时间内解决问题。例如，对于较大的 n，$n^{3/2}$ 的值比 $n(\mathrm{lb}\ n)^2$ 大，而当 n 取较小值时，可能 $n(\mathrm{lb}\ n)^2$ 反而要大一些。一个准确描述算法运行时间的函数可能是这样几个函数的线性组合。$\mathrm{lb}\ n$ 与 n、n 与 n^2 之间存在巨大差异，但要在几个快的算法中区分出哪个更快还需要仔细分析。

表 1－1 常见函数值

$\mathrm{lb}\ n$	$\sqrt{n}$	n	$n\ \mathrm{lb}\ n$	$n\ (\mathrm{lb}\ n)^2$	$n^{3/2}$	n^2
3	3	10	33	110	32	100
7	10	100	664	4414	1000	10 000
10	32	1000	9966	99 317	31 623	1 000 000
13	100	10 000	132 877	1 765 633	1 000 000	100 000 000
17	316	100 000	1 660 964	17 588 016	31 622 777	10 000 000 000
20	1000	1 000 000	19 931 569	397 267 426	1 000 000 000	1 000 000 000 000

对于很多的应用问题，当问题规模很大时，能够解决它们的惟一方法是找到一个有效算法。表 1－2 列出了当问题规模为 100 万和 10 亿时，在运算能力分别为每秒执行 100 万、

① 书中 $\mathrm{lb}x$ 表示 $\log_2 x$，即以 2 为底的对数；$\lg x$ 表示 $\log_{10} x$，即以 10 为底的对数。

10 亿、1 万亿条指令的计算机上，分别使用线性的、$n\ \mathrm{lb}\ n$ 和二次的算法解决问题所需的最小运行时间。一个快的算法能够使我们在较慢的机器上解决问题，而使用慢的算法，即使是在很快的机器上也仍要花费很长的时间。

表 1－2　解决大规模问题所需的时间

每秒操作数	问题规模为 100 万			问题规模为 10 亿		
	n	$n\ \mathrm{lb}\ n$	n^2	n	$n\ \mathrm{lb}\ n$	n^2
10^6	几秒	几秒	几周	几小时	几小时	几乎永不结束
10^9	瞬间	瞬间	几小时	几秒	几秒	几十年
10^{12}	瞬间	瞬间	几秒	瞬间	瞬间	几周

1.3.2　渐近表示

当我们进行算法分析时，可以利用数学技巧忽略一些细节，这些数学技巧就是 O 表示法(O-notation)、Ω 表示法(omega-notation)和 Θ 表示法(theta-notation)。这些表示法可以方便地表示算法最坏情况下的计算复杂度(computational complexity)。下面定义这些符号的含义。

定义 1.1　如果存在三个正常数 c_1，c_2，n_0，对于所有的 $n\geqslant n_0$，有 $0\leqslant c_1 g(n)\leqslant f(n)\leqslant c_2 g(n)$，则记作 $f(n)=\Theta(g(n))$。

图 1－2 给出了函数 $f(n)$ 和 $g(n)$ 的直观图示，其中 $f(n)=\Theta(g(n))$。对于所有位于 n_0 右边的 n 值，$f(n)$ 值落在 $c_1 g(n)$ 和 $c_2 g(n)$ 之间，即对所有 $n\geqslant n_0$，$f(n)$ 在一个常数因子内与 $g(n)$ 相等，其中 n_0 是最小可能的值。我们称 $g(n)$ 是 $f(n)$ 的一个渐近紧致界(asymptotically tight bound)。

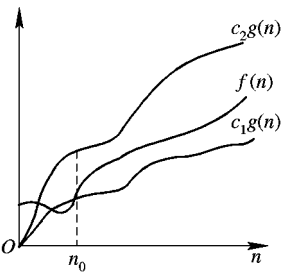


图 1－2　$f(n)=\Theta(g(n))$

Θ($g(n)$)的定义要求每个元素渐近非负，即当 n 充分大时 $f(n)$ 非负。这就要求函数 $g(n)$ 本身是非负的。

下面我们利用定义来证明 $\frac{1}{2}n^2-5n=\Theta(n^2)$。首先要确定常数 c_1，c_2 和 n_0，使得对于所有 $n\geqslant n_0$，有

$$c_1 n^2 \leqslant \frac{1}{2}n^2-5n \leqslant c_2 n^2$$

化简得

$$c_1 \leqslant \frac{1}{2}-\frac{5}{n} \leqslant c_2$$

右边的不等式在 $n\geqslant 1$，$c_2\geqslant 1/2$ 时成立。同样，左边的不等式在 $n\geqslant 11$，$c_1\geqslant 1/22$ 时成立。这样，通过选择 $c_1\geqslant 1/22$，$c_2\geqslant 1/2$ 以及 $n_0=11$，就能证明 $\frac{1}{2}n^2-5n=\Theta(n^2)$。当然，其中常数的选择不是惟一的。

定义 1.2　如果存在两个正常数 c，n_0，对于所有的 $n\geqslant n_0$，有 $0\leqslant f(n)\leqslant cg(n)$，则记作 $f(n)=\mathrm{O}(g(n))$。

图 1-3 给出了函数 $f(n)$ 和 $g(n)$ 的直观图示，其中 $f(n)=O(g(n))$。对于所有位于 n_0 右边的 n 值，$f(n)$ 的值总落在 $cg(n)$ 之下，即对所有 $n\geqslant n_0$，$g(n)$ 是计算时间 $f(n)$ 的一个上界(upper bound)函数。$f(n)$ 的数量级就是 $g(n)$。

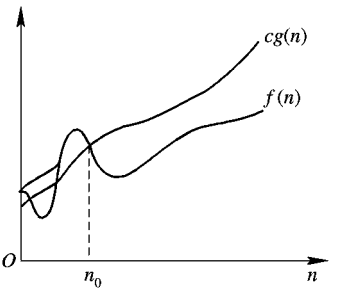


图 1-3 $f(n)=O(g(n))$

当我们用 O 表示算法的最坏情况运行时间时，同时也隐含地给出了对任意输入的运行时间的上界。例如，冒泡排序最坏情况运行时间为 $O(n^2)$，这隐含着该算法的运行时间为 $O(n^2)$。

下面我们利用定义来证明 $\frac{1}{3}n^2-3n=O(n^2)$。首先要确定常数 c 和 n_0，使得对于所有 $n\geqslant n_0$，有

$$\frac{1}{3}n^2-3n\leqslant cn^2$$

化简得

$$\frac{1}{3}-\frac{3}{n}\leqslant c$$

不等式在 $n\geqslant 1$，$c\geqslant 1/3$ 时成立。

定义 1.3 如果存在两个正常数 c 和 n_0，对于所有的 $n\geqslant n_0$，有 $0\leqslant cg(n)\leqslant f(n)$，则记作 $f(n)=\Omega(g(n))$。

图 1-4 给出了函数 $f(n)$ 和 $g(n)$ 的直观图示，其中 $f(n)=\Omega(g(n))$。对于所有位于 n_0 右边的 n 值，$f(n)$ 的值总落在 $cg(n)$ 之上，即对所有 $n\geqslant n_0$，$g(n)$ 是计算时间 $f(n)$ 的一个下界(lower bound)函数。

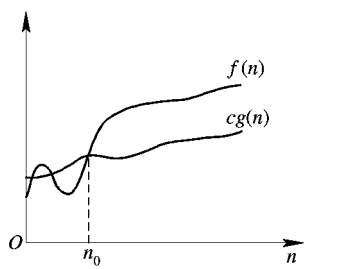


图 1-4 $f(n)=\Omega(g(n))$

当我们用 Ω 表示算法的最佳情况运行时间时，同时也隐含地给出对任意输入的运行时间的下界。例如，冒泡排序算法的最佳情况运行时间为 $\Omega(n^2)$，这隐含着该算法的运行时间为 $\Omega(n^2)$。因此可得，冒泡排序运行时间为 $\Omega(n^2)$。

下面我们利用定义来证明 $\frac{1}{3}n^2-3n=\Omega(n^2)$。首先要确定常数 c 和 n_0，使得对于所有 $n\geqslant n_0$，有

$$cn^2\leqslant\frac{1}{3}n^2-3n$$

化简得

$$c\leqslant\frac{1}{3}-\frac{3}{n}$$

不等式在 $n\geqslant 18$，$c\geqslant 1/6$ 时成立。

从计算时间上可以把算法分成两类，凡可用多项式来对其计算时间限界的算法，称为多项式时间算法(polynomial time algorithm)；而计算时间用指数函数限界的算法称为指数时间算法(exponential time algorithm)。例如，一个计算时间为 $O(1)$ 的算法，它的基本运算执行的次数是固定的，因此，总的时间由一个常数来限界；而一个计算时间为 $O(n^2)$

的算法则由一个二次多项式来限界。以下六种计算时间的多项式时间算法最为常见，其关系为

$$O(1) < O(\text{lb } n) < O(n) < O(n \text{ lb } n) < O(n^2) < O(n^3)$$

指数时间算法一般有 $O(2^n)$、$O(n!)$和 $O(n^n)$这几种，其关系为

$$O(2^n) < O(n!) < O(n^n)$$

其中最为常见的是计算时间为 $O(2^n)$的算法。

习　　题

1－1

(1) 冒泡排序在最坏情况下要做多少次元素比较？最坏情况下元素排列顺序是什么？

(2) 冒泡排序在最好情况下元素排列顺序是什么？在这种情况下，需要进行多少次元素比较？

1－2　利用 BUBBLE-SORT 排序算法，通过实例 $A=\langle 31, 41, 59, 26, 41, 58\rangle$说明该算法的运行过程。

```
BUBBLE-SORT(A)
1   for i ← 1 to length[A]
2       do for j ← length[A] downto i+1
3             do if A[j] < A[j−1]
4                   then exchange A[j] ↔ A[j−1]
```

1－3　重写冒泡排序，排成递减次序。

1－4　假设在同一台机器上实现插入排序和归并排序。对于输入规模 n，冒泡排序运行步数为 $8n^2$，归并排序运行步数为 $64n$ lb n。当 n 为多大值时，冒泡排序优于归并排序？

1－5　设有两个运行时间分别为 $100n^2$ 和 2^n 的算法，要使前者快于后者，n 最小为多少？

1－6　比较函数的运行时间。函数 $f(n)$和时间 t 如表 1－3 所示。请确定时间 t 内可解的最大问题规模 n。假设解问题算法需花费 $f(n)$微秒时间。

表 1－3　函数 $f(n)$与时间 t 的对应关系

$f(n)$ \ t	1 秒	1 分	1 小时	1 天	1 月	1 年	1 世纪(100 年)
lb n							
$\sqrt{n}$							
n							
n lb n							
n^2							
n^3							
2^n							
$n!$							

1-7　考虑查找问题。

输入：n 个数的序列 $A=\langle a_1, a_2, \cdots, a_n\rangle$ 和一个值 v。

输出：满足 $v=A[i]$ 的下标 i。如果 v 不在 A 中，则输出特殊值 NIL。

写出在序列 A 中线性查找 v 的伪代码。利用循环不变式，证明你的算法是正确的。确信你所构造的循环不变式履行了三个性质。

1-8　用 Θ 表示法表示函数 $n^3/1000-100n^2-100n+3$。

1-9　考虑对存储在 A 中的元素进行排序。首先，找出 A 中的最小元素，并将最小元素与 $A[1]$ 交换。然后，找出 A 中的次小元素，并将它与 $A[2]$ 交换。对 A 中的前 $n-1$ 个元素继续这一过程。这个算法称为选择排序。给出这个算法的循环不变式。为什么这个算法只运行前 $n-1$ 个元素，而不是所有 n 个元素？用 Θ 表示法表示该算法的最佳和最坏情况下的运行时间。

1-10　对于线性查找问题，假设查找每个元素的概率相等，则平均情况下要查找多少个元素？最坏情况下要查找多少个元素？(用 Θ 表示法)

1-11　给定 n 个元素的集合 S 和一个整数 x，描述一个 $\Theta(n \text{ lb } n)$ 的算法，确定 S 中是否存在两个元素其和正好为 x。

1-12　设 $f(n)$ 和 $g(n)$ 是渐近非负函数。利用 Θ 表示法，证明

$$\max(f(n), g(n)) = \Theta(f(n)+g(n))$$

1-13　$2^{n+1}=O(2^n)$？$2^{2n}=O(2^n)$？

1-14　对于两个函数 $f(n)$ 和 $g(n)$，$f(n)=\Theta(g(n))$ 当且仅当 $f(n)=O(g(n))$ 且 $f(n)=\Omega(g(n))$。

1-15　设

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

是阶为 d 的 n 的多项式，其中 $a_i>0$。设 k 是常数，利用渐近表示法的定义，证明以下性质：

(1) 如果 $k\geqslant d$，那么 $p(n)=O(n^k)$。

(2) 如果 $k\leqslant d$，那么 $p(n)=\Omega(n^k)$。

(3) 如果 $k=d$，那么 $p(n)=\Theta(n^k)$。

1-16　设 $f(n)$ 和 $g(n)$ 是渐近正函数，试证明以下猜测是否正确。

(1) $f(n)=O(g(n))$ 蕴含 $g(n)=O(f(n))$。

(2) $f(n)+g(n)=\Theta(\min(f(n), g(n)))$。

(3) $f(n)=O(g(n))$ 蕴含 $\text{lb}(f(n))=O(\text{lb}(g(n)))$，其中对于所有足够大的 n，$\text{lb}(g(n))\geqslant 1$ 且 $f(n)\geqslant 1$。

(4) $f(n)=O(g(n))$ 蕴含 $2^{f(n)}=O(2^{g(n)})$。

(5) $f(n)=O((f(n))^2)$。

(6) $f(n)=O(g(n))$ 蕴含 $g(n)=\Omega(f(n))$。

(7) $f(n)=\Theta(f(n/2))$。

1-17　lb $*$ 函数中使用的迭代算子“$*$”，可用于实数域上任一单调递增函数 $f(n)$ 中。对于给定的常数 $c\in\mathbf{R}$，迭代函数 f_c^* 定义如下：