“东北林业大学优秀教材及学术专著出版与奖励专项资金”资助出版

计算机算法设计与分析

JISUANJI SUANFA SHEJI YU FENXI

主 编 李艳娟 陈 冬 边继龙

東北林業大學出版社
Northeast Forestry University Press

“东北林业大学优秀教材及学术专著出版与奖励专项资金”资助出版

计算机算法设计与分析

主　编　李艳娟　陈　冬　边继龙

東北林業大學出版社
Northeast Forestry University Press
·哈尔滨·

版权专有　侵权必究

举报电话：0451-82113295

图书在版编目（CIP）数据

计算机算法设计与分析 / 李艳娟，陈冬，边继龙主编. —哈尔滨：东北林业大学出版社，2018.7

ISBN 978－7－5674－1493－8

Ⅰ. ①计…　Ⅱ. ①李…　②陈…　③边…　Ⅲ. ①电子计算机-算法设计②电子计算机-算法分析　Ⅳ. ①TP301.6

中国版本图书馆 CIP 数据核字（2018）第 182003 号

责任编辑：潘　琦

责任校对：任兴华

封面设计：乔鑫鑫

出版发行：东北林业大学出版社

（哈尔滨市香坊区哈平六道街 6 号　邮编：150040）

印　　装：哈尔滨市石桥印务有限公司

规　　格：185 mm×260 mm　16 开

印　　张：11

字　　数：254 千字

版　　次：2018 年 7 月第 1 版

印　　次：2018 年 7 月第 1 次印刷

定　　价：28.00 元

如发现印装质量问题，请与出版社联系调换。（电话：0451-82113296　82191620）

此为试读，需要完整PDF请访问：www.ertongbook.com

前　言

作为问题求解和程序设计的重要基础，《计算机算法设计与分析》在计算机科学与技术专业的课程体系中是一门重要的必修课。学习该课程，不但为学习其他专业课程奠定了扎实的基础，也对培养学生的逻辑思维和创造性有着不可替代的作用。教育部高等学校计算机科学与技术专业教学指导委员会提出的《计算机科学与技术专业规范》把该课程列入本专业的核心课程之一。纵观计算机学科数年来的发展历史，算法与计算复杂性理论一直是计算机科学研究的热点和活跃领域，也是获得图灵奖最多的研究领域之一。面对计算机应用领域的大量问题，最重要的是根据问题的性质选择正确的求解思路，即找到一个好的算法。特别是在复杂的、海量的信息处理中，一个好的算法往往起着决定性的作用。

目前已经出版了许多关于计算机算法的教材，各有特色。我们在多年从事算法设计分析及计算复杂性理论的教学和研究的基础上，精心选材，完成了本书的写作。本书的主要特点包括以下几方面。

(1) 每章都给出每种算法策略的基本思路、设计步骤及适用的问题。

(2) 每章都给出大量的具体实例，并对每个实例进行充分的讲解，让读者通过实例理解算法策略。

(3) 本书的素材来自多年的教学积累，选材适当，组织合理，先引入基本概念和数学基础知识，然后进入算法设计与分析的核心内容。在叙述中不但注意理论的严谨性，也精选了大量生动有趣的例子。每章都配有难度适当的习题，适合教学使用。

全书共 7 章。第 1 章是基础知识，介绍了算法设计与分析有关的基本概念、符号、数学知识和算法分析与问题的计算复杂度；第 2~7 章分别阐述递归与分治策略、动态规划、贪心算法、回溯法、分支限界法、概率算法等算法设计技术。

本书既可以作为本科生教材，也可以作为研究生教材。此外，对于从事实际问题求解的研究工作者，本书也可以作为一本算法设计与分析的入门参考书。在编写过程中，作者参考了国内外多种版本的算法设计与分析以及计算复杂性方面的教材、论文和专著，从中吸取了一些好的思路和素材，在此一并向这些作者致谢。感谢东北林业大学出版社对本书出版的大力支持。我们期待着广大读者，特别是使用本书的教师和学生对本书的批评、指正和建议。

编　者

2018 年 3 月

目　录

1　算法概述 …… (1)
1.1　算法在计算机科学中的地位 …… (1)
1.2　算法的概念 …… (1)
1.3　算法复杂性分析 …… (2)
习题 …… (8)
2　递归与分治策略 …… (9)
2.1　递归的概念 …… (9)
2.2　分治策略的基本思想 …… (12)
2.3　求最值 …… (14)
2.4　二分查找 …… (16)
2.5　大整数乘法 …… (17)
2.6　Strassen 矩阵乘法 …… (27)
2.7　合并排序 …… (30)
2.8　棋盘覆盖 …… (32)
2.9　循环赛日程表 …… (36)
2.10　快速排序 …… (38)
习题 …… (40)
3　动态规划 …… (41)
3.1　概述 …… (41)
3.2　矩阵连乘问题 …… (43)
3.3　动态规划算法的基本要素 …… (48)
3.4　最长公共子序列 …… (51)
3.5　最大子段和 …… (56)
3.6　凸多边形最优三角剖分 …… (61)
3.7　0-1 背包问题 …… (63)
习题 …… (66)
4　贪心算法 …… (70)
4.1　概述 …… (70)
4.2　TSP 问题 …… (72)
4.3　图着色问题 …… (75)
4.4　最小生成树问题 …… (76)

4.5 0-1 背包问题 …… (82)
4.6 活动安排问题 …… (86)
4.7 多机调度问题 …… (88)
习题…… (90)
5 回溯法 …… (92)
5.1 回溯法的算法框架 …… (92)
5.2 装载问题 …… (98)
5.3 批处理作业调度 …… (101)
5.4 符号三角形问题 …… (103)
5.5 n-皇后问题 …… (106)
5.6 0-1 背包问题 …… (110)
5.7 最大团问题 …… (115)
5.8 图着色问题 …… (119)
5.9 旅行商问题 …… (122)
习题…… (125)
6 分支限界法 …… (127)
6.1 分支限界法的基本思想 …… (127)
6.2 装载问题 …… (129)
6.3 布线问题 …… (138)
6.4 0-1 背包问题 …… (142)
6.5 最大团问题 …… (147)
习题…… (151)
7 概率算法 …… (153)
7.1 概述 …… (153)
7.2 数值概率算法 …… (153)
7.3 舍伍德概率算法 …… (156)
7.4 拉斯维加斯型概率算法 …… (160)
7.5 蒙特卡罗型概率算法 …… (164)
习题…… (167)
参考文献…… (169)

1 算法概述

1.1 算法在计算机科学中的地位

算法是计算机科学的重要主题。20 世纪 70 年代前，计算机科学基础的主题没有被明确地认清；20 世纪 70 年代 Knuth 出版 *The Art of Computer Programmin*，以算法研究为主线确立了算法为计算机科学基础的重要主题；20 世纪 70 年代后，算法作为计算机科学核心推动了计算机科学技术的飞速发展。程序设计的目标就是要编出一套让计算机按照人的旨意进行操作的指令。算法是处理问题的策略，数据结构是问题的数学模型。

1.2 算法的概念

算法（Algorithm）是一系列解决问题的清晰指令，也就是说，能够对一定规范的输入，在有限时间内获得所要求的输出。

算法可以理解为由基本运算及规定的运算顺序所构成的完整的解题步骤，或者看成按照要求设计好的有限的确切的计算序列，并且这样的步骤和序列可以解决一类问题。

一个算法具有以下五个重要的特征。

(1) 有穷性。一个算法必须保证执行有限步之后结束。

(2) 确切性。算法的每一个步骤必须有确切的定义。

(3) 输入。一个算法可以没有或有多个输入，以刻画运算对象的初始情况，所谓没有输入是指算法本身给出了初始条件。

(4) 输出。一个算法有一个或多个输出，以反映对输入数据加工后的结果。没有输出的算法是毫无意义的。

(5) 可行性。算法原则上应该能够精确地运行，而且人们用笔和纸做有限次运算后即可完成。

例 1.1 以下两段算法是否满足算法的特性，如不满足，违反了哪些特性?

```
void exam1 ( )
{
  int n=2;
  while (n%2==0)
   {
```

```
        n=n+2;
        printf ("%d\n", n);
        }
    }
    void exam2 ()
    {
    int y=0;
    int x=5/y;
    printf ("%d,%d", x, y);
    }
```

这两段算法都不满足算法的特性。第 1 个程序是死循环，违反了算法的有穷性；第 2 个程序包含除零错误，违反了算法的可行性。

程序与算法不同。程序是算法用某种程序设计语言的具体实现。程序可以不满足算法的性质。例如操作系统，是一个在无限循环中执行的程序，因而不是一个算法。

程序与算法的区别大体表现在以下三个方面。

(1) 一个程序不一定满足有穷性。例如操作系统，只要整个系统不遭破坏，它将永远不会停止，即使没有作业需要处理，它仍处于动态等待中。因此，操作系统不是一个算法。

(2) 程序中的指令必须是机器可执行的，而算法中的指令则无此限制。

(3) 算法是对问题的解，而程序则是算法在计算机上的特定实现。一个算法若用程序设计语言来描述，则它就是一个程序。

算法可采用多种描述语言来描述，例如，自然语言、计算机语言或某些伪语言。各种描述语言在对问题的描述能力方面存在一定的差异。例如，自然语言较为灵活，但不够严谨；而计算机语言虽然严谨，但由于语法方面的限制，导致其灵活性不足。因此，许多教材中采用的是以一种计算机语言为基础，适当添加某些功能或放宽某些限制而得到的一种类语言。这类语言既具有计算机语言的严谨性，又具有灵活性，同时也容易上机实现，因而被广泛接受。本教材代码采用类 C 语言实现。

1.3 算法复杂性分析

1.3.1 算法复杂性的概念

一个算法的复杂性体现在运行该算法所需要的计算机资源上，所需要的资源越多，则表示该算法的复杂性越高；反之，所需要的资源越少，则表示该算法的复杂性越低。计算机的资源，最重要的是时间和空间（即存储器）资源。因此，算法的复杂性有时间复杂性和空间复杂性之分。不言而喻，对于任意给定的问题，设计出复杂性尽可能低的算法是设计算法时追求的一个重要目标。另一方面，当给定的问题已有多种算法时，选择其中复杂

性最低者，是选用算法应遵循的一个重要原则。因此，算法复杂性分析对算法的设计或选用有重要的指导意义和实用价值。确切地说，算法的复杂性是运行算法所需要的计算机资源的量。需要的时间资源的量称为时间复杂性；需要的空间资源的量称为空间复杂性。这个量应该集中反映算法的效率，而从运行该算法的实际计算机中抽象出来，换句话说，这个量应该是只依赖于算法要解的问题的规模和算法的输入函数。如果分别用 n 和 I 表示算法要解的问题的规模和算法的输入，而且用 C 表示复杂性，那么，应该将算法复杂性表示为C（n，I）。如果把时间复杂性和空间复杂性分开，并分别用 T 和 S 来表示，那么应该有：$T=T$（n，I）和 $S=S$（n，I）。由于时间复杂性与空间复杂性概念雷同，计量方法相似，且空间复杂性分析相对简单些，所以本书将主要讨论时间复杂性。现在的问题是如何将复杂性函数具体化，即对于给定的 n 和 I，如何导出 $T=T$（n，I）和 $S=S$（n，I）的数学表达式，给出计算 $T=T$（n，I）和 $S=S$（n，I）的法则。下面以 $T=T$（n，I）为例，将复杂性函数具体化。

根据 $T=T$（n，I）的概念，它应该是算法在一台抽象的计算机上运行所需要的时间。设此抽象的计算机所提供的元运算有 k 种，分别记为 O_1，O_2，…，O_k。又设每执行一次这些元运算所需要的时间分别为 t_1，t_2，…，t_k。对于给定的算法 A，设经统计，用到元运算 O_i的次数为 e_i（$i=1$，2，…，k）。很显然，对于每一个 i（$1\leqslant i\leqslant k$），e_i是 n 和 I 的函数，即 $e_i=e_i$（n，I）。那么有 $T(n,\ I)=\sum\limits_{i=1}^{k}t_ie_i(n,\ I)$，其中 t_i（$i=1$，2，…，k）是与 n 和 I 无关的常数。

显然，不可能对规模为 n 的每一种合法的输入 I 都统计 $e_i=e_i$（n，I），$i=1$，2，…，k。因此 T（n，I）的表达式还得进一步简化，或者说，只能在规模为 n 的某些或某类有代表性的合法输入中统计相应的 e_i，并评价时间复杂性。

通常考虑 3 种情况下的时间复杂性，即最坏情况、最好情况和平均情况下的时间复杂性，并分别记为 $T_{\max}$（n）、$T_{\min}$（n）和 T_{avg}（n）。在数学上有

$$T_{\max}(n)=\max_{I\in D_n}T(n,\ I)=\max_{I\in D_n}\sum_{i=1}^{k}t_ie_i(n,\ I)=\sum_{i=1}^{k}t_ie_i(n,\ I^*)=T(n,\ I^*)$$

$$T_{\min}(n)=\min_{I\in D_n}T(n,\ I)=\min_{I\in D_n}\sum_{i=1}^{k}t_ie_i(n,\ I)=\sum_{i=1}^{k}t_ie_i(n,\ \bar{I})=T(n,\ \bar{I})$$

$$T_{\text{avg}}(n)=\sum_{I\in D_n}P(I)T(n,\ I)=\sum_{I\in D_n}P(I)\sum_{i=1}^{k}t_ie_i(n,\ I)$$

其中，D_n是规模为 n 的合法输入的集合；I^* 是 D_n中的一个使 T（n，I^*）达到$T_{\max}$（n）的合法输入；$\bar{I}$是 D_n 中的一个使 T（n，$\bar{I}$）达到 $T_{\min}$（n）的合法输入；而P（I）是在算法的应用中出现输入 I 的概率。

以上 3 种情况下的时间复杂性各从某一个角度来反映算法的效率，各有各的局限性，也各有各的用处。实践表明，可操作性最好且最有实际价值的是最坏情况下的时间复杂性。本教材对算法时间复杂性的分析主要采用最坏情况下时间复杂性分析。

1.3.2 渐进符号

要精确地表示算法的运行时间函数常常是很困难的，即使能够给出，也可能是个相当

复杂的函数，函数的求解本身也是相当复杂的。考虑到算法分析的主要目的在于比较求解同一个问题的不同算法的效率，为了客观地反映一个算法的运行时间，可以用算法中基本语句的执行次数来度量算法的工作量。基本语句是执行次数与整个算法的执行次数呈正比的语句，基本语句对算法运行时间的贡献最大，是算法中最重要的操作。这种衡量效率的方法得出的不是时间量，而是一种增长趋势的度量。换言之，只考察当问题规模充分大时算法中基本语句的执行次数在渐近意义下的阶，通常使用大 O、大 Ω 和 θ 这 3 种渐进符号表示。

1.3.2.1 大 O 符号

定义 1.1 若存在两个正的常数 c 和 n_0，对于任意 $n \geqslant n_0$，都有 $T(n) \leqslant c \times f(n)$，则称 $T(n) = O(f(n))$ [或称算法在 $O(f(n))$ 中]。

大 O 符号用来描述增长率的上限，表示 $T(n)$ 的增长最多像 $f(n)$ 增长得那样快，也就是说，当输入规模为 n 时，算法消耗时间的最大值，这个上限的阶越低，结果就越有价值。

大 O 符号的含义如图 1-1 所示，为了说明这个定义，将问题规模 n 扩展为实数。

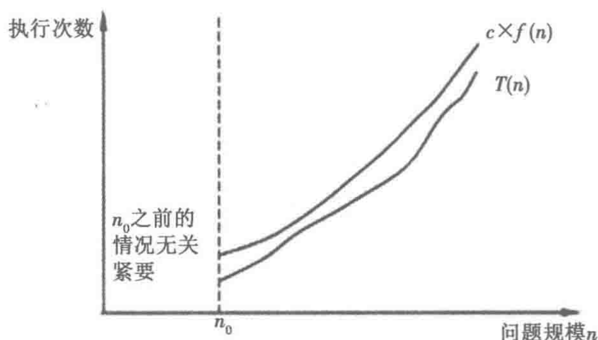

图 1-1 大 O 符号的含义

应该注意的是，定义 1.1 给了很大的自由度来选择常量 c 和 n_0 的特定值，例如，下列推导都是合理的：

$$100n+5 \leqslant 100n+n \ (\text{当 } n \geqslant 5) = 101n = O(n) \ (c=101,\ n_0=5)$$

$$100n+5 \leqslant 100n+5n \ (\text{当 } n \geqslant 1) = 105n = O(n) \ (c=105,\ n_0=1)$$

1.3.2.2 大 Ω 符号

定义 1.2 若存在两个正的常数 c 和 n_0，对于任意 $n \geqslant n_0$，都有 $T(n) \geqslant c \times g(n)$，则称 $T(n) = \Omega(g(n))$ [或称算法在 $\Omega(g(n))$ 中]。

大 Ω 符号用来描述增长率的下限，也就是说，当输入规模为 n 时，算法消耗时间的最小值。与大 O 符号对称，这个下限的阶越高，结果就越有价值。

大 Ω 符号的含义如图 1-2 所示。

大 Ω 符号常用来分析某个问题或某类算法的时间下界。例如，矩阵乘法问题的时间下界为 $\Omega(n^2)$，是指任何两个 $n \times n$ 矩阵相乘的算法的时间复杂性不会小于 n^2，基于比较的排序算法的时间下界为 $\Omega(n\log_2 n)$，是指无法设计出基于比较的排序算法，其时间复杂性小于 $n\log_2 n$。

此为试读，需要完整PDF请访问：www.ertongbook.com

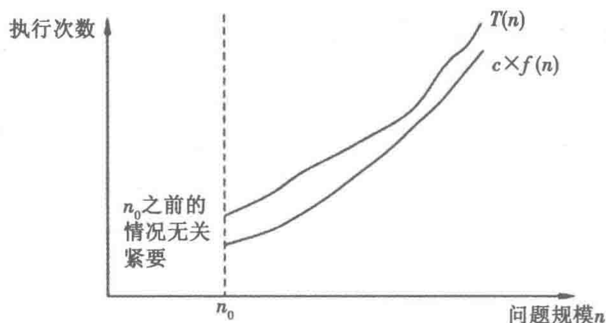

图 1-2 大 Ω 符号的含义

大 Ω 符号常常与大 O 符号配合以证明某问题的一个特定算法是该问题的最优算法，或是该问题中的某算法类中的最优算法。

1.3.2.3 θ 符号

定义 1.3 若存在 3 个正的常数 c_1，c_2 和 n_0，对于任意 $n \geqslant n_0$，都有 $c_1 \times f(n) \geqslant T(n) \geqslant c_2 \times f(n)$，则称 $T(n)=\theta(f(n))$。

θ 符号意味着 $T(n)$ 与 $f(n)$ 同阶，用来表示算法的精确阶。θ 符号的含义如图 1-3 所示。

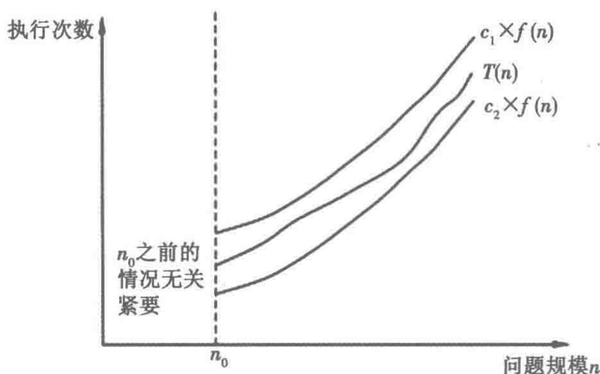

图 1-3 θ 符号的含义

下面举例说明大 O、大 Ω 和 θ 这 3 种渐进符号的使用。

例 1.2 $T(n)=3n-1$。

解：当 $n \geqslant 1$ 时，$3n-1 \leqslant 3n=O(n)$；

当 $n \geqslant 1$ 时，$3n-1 \geqslant 3n-n=2n=\Omega(n)$；

当 $n \geqslant 1$ 时，$3n \geqslant 3n-1 \geqslant 2n$，则 $3n-1=\theta(n)$。

例 1.3 $T(n)=5n^2+8n+1$

解：当 $n \geqslant 1$ 时，$5n^2+8n+1 \leqslant 5n^2+8n+n=5n^2+9n \leqslant 5n^2+9n^2 \leqslant 14n^2=O(n^2)$；

当 $n \geqslant 1$ 时，$5n^2+8n+1 \geqslant 5n^2=\Omega(n^2)$；

当 $n \geqslant 1$ 时，$14n^2 \geqslant 5n^2+8n+1 \geqslant 5n^2$，则 $5n^2+8n+1=\theta(n^2)$。

1.3.3 算法时间复杂性分析/计算方法

1.3.3.1 非递归算法

从算法是否递归调用的角度来说，可以将算法分为非递归算法和递归算法。对非递归算法时间复杂性的分析，关键是建立一个代表算法运行时间的求和表达式，然后用渐进符号表示这个求和表达式。

例 1.4 在一个整型数组中查找最小值元素，具体代码如下：

```
int ArrayMin (inta [], intn)
{
min=a [0];
for (i=1; i<n; i++)
if (a [i] <min) min=a [i];
return min;
}
```

在例 1.4 的代码中，问题规模显然是数组中的元素个数，执行最频繁的操作是在 for 循环中，循环体中包含两条语句：比较和赋值，应该把哪一个语句作为基本语句呢？由于每做一次循环都会进行一次比较，而赋值语句却不一定执行，所以，应该把比较运算作为该算法的基本语句。接下来考虑基本语句的执行次数，由于每执行一次循环就会做一次比较，而循环变量 i 从 1 到 n-1 之间的每个值都会做一次循环，可得到如下求和表达式：

$$T(n) = \sum_{i=1}^{n-1} i$$

用渐进符号表示这个求和表达式：

$$T(n) = \sum_{i=1}^{n-1} i = n - 1 = O(n)$$

非递归算法分析的一般步骤如下所述。

(1) 决定用哪个（或哪些）参数作为算法问题规模的度量。在大多数情况下，问题规模是很容易确定的，可以从问题的描述中得到。

(2) 找出算法中的基本语句。算法中执行次数最多的语句就是基本语句，通常是最内层循环的循环体。

(3) 检查基本语句的执行次数是否只依赖于问题规模。如果基本语句的执行次数还依赖于其他一些特性（如数据的初始分布），则最好情况、最坏情况和平均情况的效率需要分别研究。

(4) 建立基本语句执行次数的求和表达式。计算基本语句执行的次数，建立一个代表算法运行时间的求和表达式。

(5) 用渐进符号表示这个求和表达式。计算基本语句执行次数的数量级，用大 O 符号来描述算法增长率的上限。

1.3.3.2 递归算法

计算递归算法的时间复杂度主要有两种方法。

（1）Iteration 法。Iteration 法求递归算法的时间复杂度的基本过程为：首先循环地展开递归方程；然后把递归方程转化为和式；最后使用求和技术求解。

例 1.5 设 $n=2^k$，$T(1)=1$，求 $T(n)=2T(n/2)+2$ 的渐进阶。

解：
$$
\begin{aligned}
T(n)&=2T(n/2)+2\\
&=2(2T(n/4)+2)+2\\
&=4T(n/4)+4+2\\
&=4(2T(n/8)+2)+4+2\\
&=8T(n/8)+8+4+2\\
&=\cdots=2^kT(n/2^k)+2^k+2^{k-1}+\cdots+2\\
&=2^k+2^{k+1}-2=3n-2=O(n)
\end{aligned}
$$

（2）Master 定理。通过 Master 定理也可以求递归算法的时间复杂度，Master 定理的具体内容如下：

设 $a\geqslant 1$ 和 $b>1$ 是常数，$f(n)$ 是一个函数，$T(n)$ 是定义在非负整数集上的函数 $T(n)=aT(n/b)+f(n)$，$T(n)$ 可以如下求解：

①若 $f(n)=O(n^{(\log_b a)-\varepsilon})$，$\varepsilon>0$ 是常数，则 $T(n)=\theta(n^{\log_b a})$；

②若 $f(n)=\theta(n^{\log_b a})$，则 $T(n)=\theta(n^{\log_b a}\lg n)$；

③若 $f(n)=\Omega(n^{(\log_b a)+\varepsilon})$，$\varepsilon>0$ 是常数，且对于所有充分大的 n，$af(n/b)\leqslant cf(n)$，$c<1$ 是常数，则 $T(n)=\theta(f(n))$。

直观地，我们用 $f(n)$ 与 $n^{\log_b a}$ 比较：

①若 $n^{\log_b a}$ 大，则 $T(n)=\theta(n^{\log_b a})$；

②若 $f(n)$ 大，则 $T(n)=\theta(f(n))$；

③若 $f(n)$ 与 $n^{\log_b a}$ 同阶，则 $T(n)=\theta(n^{\log_b a}\lg n)$。

具体如图 1-4 所示。

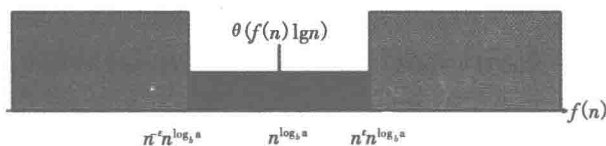

图 1-4 Master 定理示意图

例 1.6 求解 $T(n)=9T(n/3)+n$。

解：$a=9$，$b=3$，$f(n)=n$，$n^{\log_b a}=\theta(n^2)$

$\because f(n)=n=O(n^{(\log_b a)-\varepsilon})$，$\varepsilon=1$

$\therefore T(n)=\theta(n^{\log_b a})=\theta(n^2)$

例 1.7 求解 $T(n)=T(2n/3)+1$。

解：$a=1$，$b=3/2$，$f(n)=1$

$n^{\log_b a}=n^{\log_{3/2}1}=n^0=1$

$f(n)=1=\theta(1)=\theta(n^{\log_b a})$

$T(n)=\theta(n^{\log_b a}\lg n)=\theta(\lg n)$

例 1.8　求解 $T(n)=3T(n/4)+n\lg n$。

解：$a=3,b=4,f(n)=n\lg n$，$n^{\log_b a}=n^{\log_4 3}=O(n^{0.793})$

$\because f(n)=n\lg n \geqslant n=n^{(\log_b a)+\varepsilon},\varepsilon\approx 0.2$

对所有 $n,af(n/b)=3(n/4)\lg(n/4)\leqslant 3n/4\lg n$

$=cf(n)\qquad c=3/4$

$\therefore T(n)=\theta(f(n))=\theta(n\lg n)$

习题

（1）求下列表达式的时间复杂度：

①$T(n)=5T(n/2)+\theta(n^2)$；

②$T(n)=2T(n/2)+n$；

③$T(n)=5T(n/2)+\theta(n^3)$。

（2）相传，国际象棋是很久以前由印度人 Shashi 发明的，当他把该发明献给国王时，国王很高兴，就许诺可以给这个发明人任何他想要的奖赏。Shashi 要求以这种方式给他一些粮食：棋盘的第 1 个方格内只放 1 粒麦粒，第 2 格 2 粒，第 3 格 4 粒，第 4 格 8 粒，以此类推，直到 64 个方格全部放满。这个奖赏的最终结果会是什么样的呢？

（3）有 4 个人打算过桥，这座桥每次最多只能有两个人同时通过。他们都在桥的某一端，并且是在晚上，过桥需要一只手电筒，而他们只有一只手电筒。这就意味着两个人过桥后必须有一个人将手电筒带回来。每个人走路的速度是不同的：甲过桥要用 1 分钟，乙过桥要用 2 分钟，丙过桥要用 5 分钟，丁过桥要用 10 分钟，显然，两个人走路的速度等于其中较慢那个人的速度，问题是他们全部过桥最少要用多长时间？

（4）欧几里得游戏：开始的时候，白板上有两个不相等的正整数，两个玩家交替行动，每次行动时，当前玩家都必须在白板上写出任意两个已经出现在板上的数字的差，而且这个数字必须是新的，也就是说，和白板上的任何一个已有的数字都不相同，当一方再也写不出新数字时，他就输了。请问，你是选择先行动还是后行动？为什么？

2 递归与分治策略

用计算机求解问题时，所需的计算时间与问题的规模有关。问题的规模越大，解题所需的计算时间往往也越长，而且较难处理，反之亦然。例如，求 n 个元素中的最大值，当 $n=1$ 时，不需任何比较；$n=2$ 时，只要做一次比较即可找到最大值；$n=3$ 时只要做两次比较即可……而当 n 较大时，问题就不那么容易处理了，这里 n 就是上面所说的问题规模。

对于一个规模较大的问题，有时直接解决是相当困难的。分治策略的设计思想是，将一个难以直接解决的大问题分割成一些规模较小的相同的子问题，这些子问题较容易解决，然后依次解决这些子问题，进而解决大规模问题。如果原问题可分割成 t 个子问题，$1<t\leqslant n$，且这些子问题都可以直接解决，并且可以利用这些子问题的解得到原问题的解，那么这种分治策略就是可行的。由分治策略产生的子问题通常是原问题的较小模式，因此分治策略通常采用递归技术实现。分治与递归就像一对孪生兄弟，经常同时应用在算法设计中，并由此产生了许多高效算法。

2.1 递归的概念

递归算法是指直接或间接地调用自身的算法，而递归函数是用函数自身给出定义的函数。递归技术在算法设计中是十分有用的。使用递归技术通常使函数的定义和算法的描述简洁且易于理解。有些数据结构，如二叉树等，由于其本身固有的递归特性，特别适合用递归的形式来描述。另外，还有一些问题，虽然其本身并没有明显的递归结构，但用递归技术来求解会使设计出的算法简洁易懂且易于分析。

递归算法设计步骤如下所述。

（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。和其他算法一样设计函数体中的操作及相关参数。

下面给出几个递归函数、递归结构和可以用递归方式解决问题的实例。

例 2.1　阶乘函数可递归地定义为

$$n! = \begin{cases} 1 & n=0 \lor n=1 \\ n(n-1)! & n>1 \end{cases}$$

阶乘函数的自变量 n 的定义域是非负整数，递归式的第一式给出了这个函数的初始值，是非递归定义的，也就是递归算法的边界，每个递归函数都必须有非递归定义的初始

值，否则，递归函数就无法计算。递归式的第二式是用较小自变量的函数值来表达较大自变量的函数值的方式来定义 n 的阶乘，也就是递归算法设计的递归关系。定义式的左右两边都引用了阶乘记号，它是递归定义式，写成递归算法如下：

```
int fac (int n)
{
if (n==0| |n==1)    return 1;
else
return n * fac (n-1);
}
```

例 2.2　排列问题。

设 $R=\{r_1, r_2, \cdots, r_n\}$ 是要进行排列的 n 个元素，求 R 的全排列 Perm（R）。

分析：设（r_i）Perm（X）表示在全排列 Perm（X）的每一个排列前加上前缀得到的排列。

（1）递归关系：Perm（R）由（r_1）Perm（R_1），（r_2）Perm（R_2），…，（r_n）Perm（R_n）构成，其中 $R_i=R-\{r_i\}$。

（2）终止条件：$n=1$ 时，Perm（R）$=r$，r 是 R 中的唯一元素。

（3）参数设置：待排序数组 List，开始下标 s，终止下标 t。

```
void Perm (int list [], int s, int t)
{
        if (s==t)
         {
  for (int i=1; i<=m; i++) printf ("%d", list [i]);
printf ("\n");
          }
          else
          for (int i=s; i<=t; i++)
             {
                swap (list [s], list [i]);
                Perm (list s+1, t);
                swap (list [s], list [i]);
              }
}
```

例 2.3　求二叉树叶子节点数。

二叉树是每个节点最多有两个孩子，且其子树有左右之分的有序树。非空二叉树包含一个根节点、一个左子树、一个右子树，并且左右子树也是二叉树。二叉树的结构是递归的，这种递归结构适合用递归算法求解。求二叉树叶子节点数的代码如下：

```
typedef struct node
{
```

```
char data;
  struct node *lchild, *rchild;
} BNode, *BiTree;
int LeafCount (BiTree T)
{
if (! T)    return 0;
  else
     if (T->lchild==NULL&&T->rchild==NULL)    return 1;
         else   return LeafCount (T->lchild) +LeafCount (T->rchild);
}
```

例 2.4 Hanoi 塔问题。

Hanoi 塔问题起源于一个传说，在 Hanoi 这个地方有一个寺庙，这里有 3 根柱子和 64 个大小不同的金碟子。每个碟子有一个孔可以穿过。所有的碟子都放在第一个柱子上，而且按照从上到下碟子的大小依次增大的顺序摆设，如图 2-1 所示。

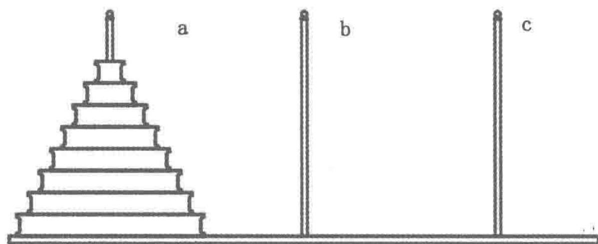

图 2-1 Hanoi 塔问题

现在，假定寺庙里的僧侣要移动这些碟子，将它们从柱子 a 移动到柱子 b 上。移动的规则如下。

（1）每次只能从一个柱子的最上面移动一个碟子到另外一个柱子上。

（2）不能将大碟子放到小碟子的上面。

按照前面这个规则，我们该怎么去移动这些碟子呢?

解决方案如下。

（1）递归关系：将 n-1 个碟子借助 b 移到 c，将最后一个移到 b，再将 n-1 个碟子借助 a 从 c 移到 b。

（2）终止条件：n=0，没有碟子需要移动了。

（3）确定参数：n，a，b，c。

```
void hanoi (int n, char a, char b, char c) //将 n 个碟子从 a 移到 b
{
     if (n==0)
     return;
     else
      {
```

此为试读，需要完整PDF请访问：www.ertongbook.com