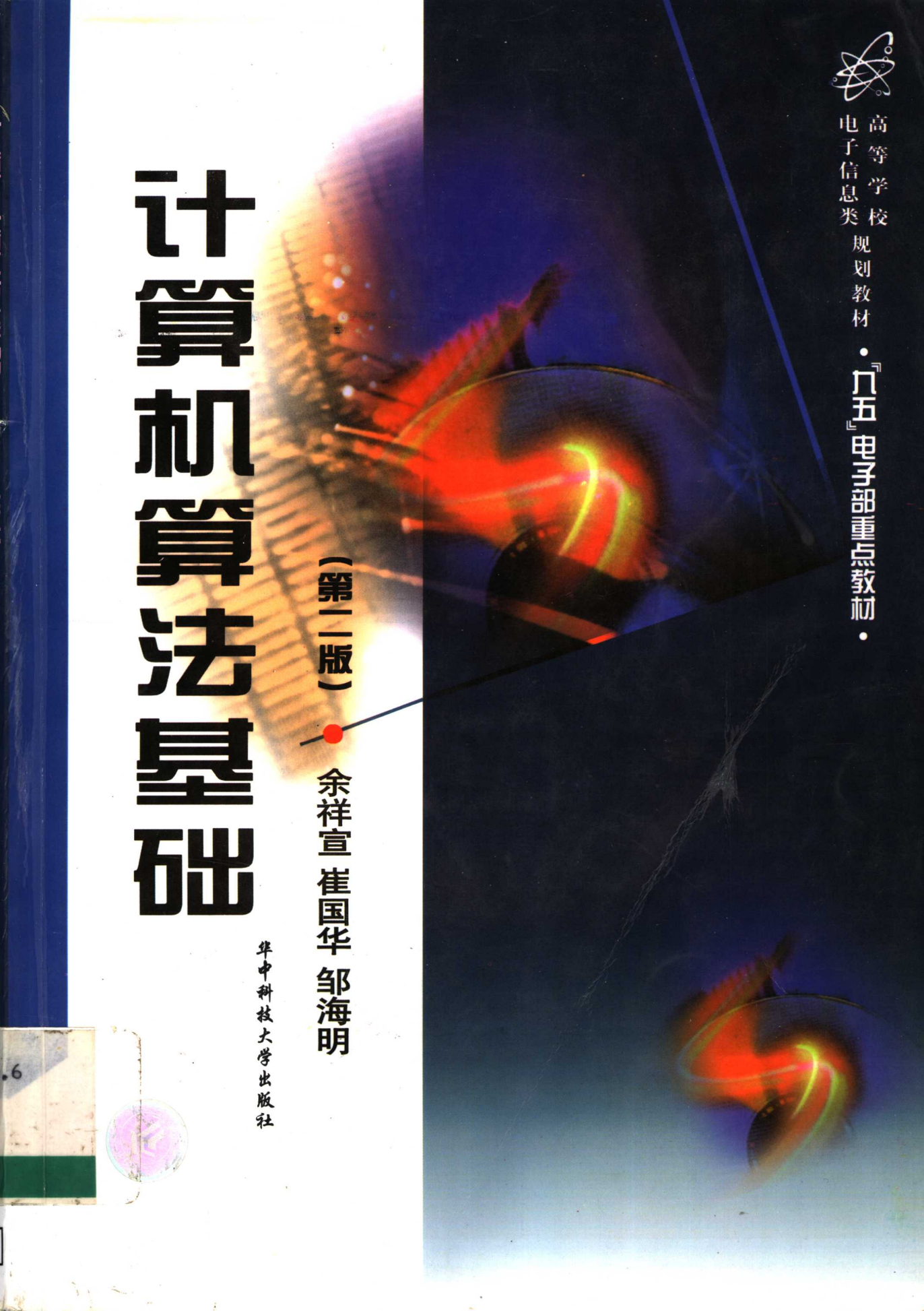

高等学校电子信息类规划教材·『九五』电子部重点教材·

计算机算法基础

（第二版）

余祥宣 崔国华 邹海明

华中科技大学出版社

计算机算法基础

（第二版）

余祥宣　崔国华　邹海明

华中科技大学出版社
（华中理工大学出版社）

内容简介

计算机算法是计算机科学和计算机应用的核心。无论是计算机系统、系统软件的设计，还是为解决计算机的各种应用课题做的设计都可归结为算法的设计。

本书围绕算法设计的基本方法，对计算机领域中许多常用的非数值算法作了精辟的描述，并分析了这些算法所需的时间和空间。全书共分九章，前七章介绍了分治法、贪心法、动态规划、基本检索与周游方法、回溯法以及分枝-限界法等基本设计方法，第八章对当今计算机科学的前沿课题——$P \stackrel{?}{=} NP$ 问题的有关知识作了初步介绍，第九章则对日益兴起的并行算法的基本设计方法作了介绍。

本书可作为高等院校与计算机有关的各专业的教学用书，也可作为从事计算机科学、工程和应用的工作人员的自学教材和参考书。

出 版 说 明

为做好全国电子信息类专业“九五”教材的规划和出版工作，根据国家教委《关于“九五”期间普通高等教育教材建设与改革的意见》和《普通高等教育“九五”国家级重点教材立项、管理办法》，我们组织各有关高等学校、中等专业学校、出版社，各专业教学指导委员会，在总结前四轮规划教材编审、出版工作的基础上，根据当代电子信息科学技术的发展和面向21世纪教学内容和课程体系改革的要求，编制了《1996～2000年全国电子信息类专业教材编审出版规划》。

本轮规划教材是由个人申报，经各学校、出版社推荐，由各专业教学指导委员会评选，并由我部教材办商各专指委、出版社后，审核确定的。本轮规划教材的编制，注意了将教学改革力度较大、有创新精神、特色风格的教材和质量较高、教学适用性较好、需要修订的教材以及教学急需，尚无正式教材的选题优先列入规划。在重点规划本科、专科和中专教材的同时，选择了一批对学科发展具有重要意义，反映学科前沿的选修课、研究生课教材列入规划，以适应高层次专门人才培养的需要。

限于我们的水平和经验，这批教材的编审、出版工作还可能存在不少缺点和不足，希望使用教材的学校、教师、同学和广大读者积极提出批评和建议，以不断提高教材的编写、出版质量，共同为电子信息类专业教材建设服务。

原电子工业部教材办公室

序

凡是学习了一种语言(不论是初级的还是高级的)程序设计课程并能编写一些实用程序的读者,也许都有这样一种体会,学会编程容易,但要想编出好程序难,因而很想学点如何设计良好算法的知识。另外,一些著名计算机科学家在有关计算机科学教育的论述中认为,计算机科学是一种创造性思维活动的科学,其教育必须面向设计。计算机算法设计与分析正是面向设计的、处于核心地位的教育课程。

基于上述的认识,自80年代初,作者对计算机科学专业的学生一直坚持开设算法设计与分析课程。在这门课程的教学过程中,我们查阅了国外流行的数种教材,发现多数教材在面向设计方面不是重视不够就是处理不甚恰当,只有E. Horowitz和S. Sahni合著的"Fundamentals of Computer Algorithms"一书比较集中地反映了以上观点。不过要将该书作为一个学期的教材则嫌内容太多。为了解决国内当前对此门课程教学的急需,在选用此书作为主要素材并参考、吸取了其它书籍的某些长处的基础上,根据计算机各专业学生目前需要形成的知识结构和在教学实践中的体会,于1985年编写了这本《计算机算法基础》,希望能对从事计算机算法教学和想设计出良好算法的人有所裨益。

本书出版以来,受到许多学校的普遍欢迎,并于1992年荣获机电部电子教材类一等奖。

计算机科学技术发展很快,特别是近几年来并行处理技术渐趋成熟,高性能并行计算机已投入使用,因此,算法课程除了讲授串行算法的设计方法外,必须增加并行算法的有关内容,以便将学生培养成为面向21世纪的计算机高级技术人才。鉴于上述原因,对本书进行了修订再版。再版时,删去了某些过时的内容,增加了新的内容,如第九章并行算法。

全书共分九章。首先,在第一章中介绍了算法的基本概念,并对计算复杂度、算法的描述工具和本书用到的基本数据结构作了简要的阐述;然后,围绕设计算法时常用的一些基本设计策略组织了第二至第七章的内容。其中每一章都先介绍一种设计算法的基本方法,然后从解决计算机科学和应用中出现的实际问题入手,由简到繁地描述几个经典的精巧算法,同时对每个算法所需的时间和空间作出数量级的分析,使读者既能学到一些常用的精巧算法,又能通过对这些设计策略的反复应用,牢固掌握这些基本设计方法,收到融会贯通之效。细心的读者从目录中就可立即发现,一个问题往往可以用多种设计策略求解。要指出的是,随着本书内容的逐步展开,读者将会体会到综合应用多种设计策略可以更有效地解决问题。第八章对当今计算机科学的前沿课题——P $\stackrel{?}{=}$ NP问题的有关知识作了初步介绍,目的是希望读者在学习了前七章内容之后,在理论上能提高到一个新的高度,激发某些读者对此课题的研究兴趣。第九章讨论了各种通用并行计算模型上的并行算法设计和分析方法。它以并行计算模型为线索,讲述了并行算法的基础知识及其基本设计技术,着重介绍了各种常用的和典型的并行算法。

本书的内容是自成体系的,凡具有大学二年级数学基础和有使用过一种高级程序设计语言编程序经验的人,都能学懂本书的内容。对于已学过数据结构课程的读者,可以跳过1.4节,而对于没学过数据结构课程的读者,则必须认真学习并掌握此节的全部内容。各章后面都附有一定数量的习题,其中有些题目最好是在写出算法后,用一种语言写成程序并到机器上去运

行，以检验所设计的算法的有效性。

讲授这门课程一般 70 学时左右即可完成。根据数年来对大学本科生、研究生讲授这门课程的经验，建议对本科生讲授第一至第七章以及第九章的全部或大部内容，对第八章只作简要的介绍。研究生则需认真学习第八章，其余章节的学习与本科生的相同。这门课程既可采用课堂讲授方式，也可采用讲授、自学和讨论相结合的方式。

本书第二版经全国计算机专业教学指导委员会讨论确定为电子部计算机专业“九五”规划教材，并列为部级重点教材。本书由福州大学王晓东教授担任责任编委，由国防科技大学陈火旺教授担任主审。

本书第一章到第七章由余祥宣编写，第八章由邹海明编写，第九章由崔国华编写。值本书再版之际，谨向在数年前就向我们建议本书应增写并行算法的中国科学技术大学唐策善教授致以诚挚的感谢。

编　　者

1998 年 6 月

目　录

第一章　导引与基本数据结构

1.1 算　　法

凡是使用数字计算机解决过数值计算或非数值计算问题的人对于算法(algorithm)一词都不陌生，因为他们都学习和编制过一些这样或那样的算法。但是，如果要给算法下一个定义或者作稍许准确一点的描述，那么至少其中的大多数人都会感到是件相当棘手的事情。的确，算法和数字、计算等基本概念一样，要给它下一个严格的定义是不容易的，只能笼统地把算法定义成解一确定类问题的任意一种特殊的方法，而在计算机科学中，算法已逐渐成了用计算机解一类问题的精确、有效方法的代名词。如果对算法作稍许详细一点的非形式描述，则**算法**就是一组有穷的规则，它规定了解决某一特定类型问题的一系列运算。此外，算法还具有以下五个重要特性：

1. 确定性

算法的每一种运算必须要有确切的定义，即每一种运算应该执行何种动作必须是相当清楚的、无二义性的。在算法中不允许有诸如“计算 5/0”或“将 6 或 7 与 x 相加”之类的运算，因为前者的结果是什么不清楚，而后者对于两种可能的运算应做哪一种也不知道。

2. 能行性

一个算法是能行的指的是算法中有待实现的运算都是基本的运算，每种运算至少在原理上能由人用纸和笔在有限的时间内完成。整数算术运算是能行运算的一个例子，而实数算术运算则不是能行的，因为某些实数值只能由无限长的十进制数展开式来表示，像这样的两个数相加就违背能行性这一特性。

3. 输入

一个算法有 0 个或多个输入，这些输入是在算法开始之前给出的量，这些输入取自特定的对象集合。

4. 输出

一个算法产生一个或多个输出，这些输出是同输入有某种特定关系的量。

5. 有穷性

一个算法总是在执行了有穷步的运算之后终止。

凡是算法，都必须满足以上五条特性。只满足前四条特性的一组规则不能称为算法，只能叫做**计算过程**。操作系统就是计算过程的一个重要例子。设计操作系统的目的是为了控制作业的运行，当没有作业时，这一计算过程并不终止，而是处于等待状态，一直等到一个新的作业进入。尽管计算过程包括这样一类重要的例子，我们还是将本书的讨论范围限制在那些总是终止的计算过程上。

由于研究计算机算法的目的最终是为了有效地求出问题的解，那就需要将算法投入到计算机上运行，因此对算法的讨论不能只研究到它能在有穷步内终止就结束，而应对有穷性作进

一步的研究，即对算法的效率要作出分析。例如，在象棋比赛中，对任意给定的一种棋局，都可以设计出一种算法来判断这一棋局是否可以导致赢局。这样的一个算法需要从开局起对所有棋子可能进行的移动以及相应的对策作逐一的检查。为了作出应走哪些棋着的决策，其计算步骤虽是有穷的，但实际上即使在最先进的计算机上运算也要千千万万年。由此可知，只应把那些在**相当**有穷步内就终止的算法投入计算机中运行，而对不能在相当有穷步内终止的算法则不必在计算机上运行，免得无益耗费计算机的宝贵资源。对这类问题的求解应另辟蹊径。

要制定一个算法，一般要经过设计、确认、分析、编码、检查、调试、计时等阶段，因此学习计算机算法必须涉及这些方面的内容。在这些内容中有许多都是现今重要而活跃的研究领域。为便于区别，把算法学习的内容分成五个不同的方面：

1. 如何设计算法

设计算法的工作是不可能完全自动化的。本书是要使读者学会已被实践证明是有用的一些基本设计策略。这些策略不仅在计算机科学，而且在运筹学、电气工程等多个领域都是非常有用的，利用它们已经设计出了很多精致有效的好算法。读者们一旦掌握了这些策略，也一定会设计出更多新的、有用的算法。

2. 如何表示算法

语言是交流思想的工具，设计的算法也要用语言恰当地表示出来。本书基本采用结构程序设计的方式，选择了一种名为 SPARKS 的程序设计语言来简单明了地表示算法。至于结构程序设计的内容本书并不打算具体介绍，而是将所能收集到的那些主要结构用于本书所给出的算法之中，只是在本章的最后一节详细讨论了一种非常重要的结构——递归。

3. 如何确认算法

一旦设计出了算法，就应证明它对所有可能的合法输入都能算出正确的答案，这一工作称为**算法确认**(algorithm validation)。要指出的是，用 SPARKS 所描述的算法还不足以是一个可以立即投入机器运行的程序(关于这一点在学完了 1.2,1.3 节之后就会更清晰)。确认的目的在于使我们确信这一算法将能正确无误地工作，而与写出这一算法所用的程序语言无关。一旦证明了算法的正确性，就可将其写成程序，在将程序放到机器上运行之前，实际上还应证明程序是正确的，即证明程序对所有可能的合法输入都能得到正确的结果，这一工作称为**程序证明**(program proving)。这一领域是当前很多计算机科学工作者集中研究的对象，还处于相当初期的阶段。在这一领域的工作还没取得突破性的进展之前，为了增强对所编制程序的置信度，只能用对程序的测试来权宜代替。

4. 如何分析算法

在前面对有穷性的讨论中，曾提及只应把能在相当有穷步内终止的算法实际投入计算机运行。细心的读者可能当时就会觉得“相当”一词用得非常含糊，能否对有穷步给出一个数量界限呢？这实际上是我们要在这里回答的问题。执行一个算法，要使用计算机的中央处理器(CPU)完成各种运算，要用存储器来存放程序和数据。**算法分析**(analysis of algorithms)是对一个算法需要多少计算时间和存储空间作定量的分析。分析算法不仅可以预计所设计的算法能在什么样的环境中有效地运行，而且可以知道在最好、最坏和平均情况下执行得怎么样，还可以使读者对解决同一问题不同算法的有效性作出比较判断。关于算法分析更确切的表征将在下一节讨论。

5. 如何测试程序

测试程序实际上由调试和作时空分布图两部分组成。**调试**(debugging)程序是在抽象数据

集上执行程序，以确定是否会产生错误的结果，若有，则修改程序。但是，这一工作正如著名计算机科学家迪伊克斯特拉(E. Dijkstra)所说的那样，“调试只能指出有错误，而不能指出它们不存在错误。”尽管如此，在程序正确性证明还没取得突破性进展的今天，调试仍是不可缺少且必须认真进行的一项重要工作。**作时空分布图**是用各种给定的数据执行调试认为是正确的程序，并测定为计算出结果所花去的时间和空间，以印证以前所作的分析是否正确和指出实现最优化的有效逻辑位置。

上述五个方面基本概括了学习算法所应涉及的内容。由于篇幅关系不可能面面俱到地讨论所有课题，而是将精力集中于设计和分析，对其它部分只适当述及。

最后要指出的是，本书中所介绍的算法，其绝大部分属于求解非数值计算范畴内的问题。

1.2 分析算法

分析算法是一种有趣的智力工作，它可以充分发挥人的聪明才智。更重要的是，从经济观点来看，由分析算法可以知道为完成一项任务所设计的算法的好坏，从而促使设计出一些更好的算法，以收到少花钱多办事、办好事的经济效果。

在讨论怎样分析算法之前，需要对算法运行的计算机类型作出假定。这对解出问题的速度影响甚大。尽管引进机器的形式模型(例如 Turing 机或随机存取机器)会使整个讨论建立在严密的理论基础之上，但对于大多数没学过这方面知识的读者来说，为此而花费较多精力去学习它也是没有必要的。就本书的绝大部分内容而言，现假定使用一台“通用”计算机就足够了。这台“通用”机就是平时所使用的顺序处理机：它每次执行程序中的一条指令；带有容量足够的随机存取存储器，在固定的时间内可以把一个数从任一单元取出或存入。

要分析一个算法，首先就要确定使用哪些运算以及执行这些运算所用的时间。一般将这些运算分为两类。一类是基本运算，它既包括加、减、乘、除这四种基本的整数算术运算，也包括浮点算术、比较、对变量赋值和过程调用等。这些运算所用时间虽然不同，但一般都只花费一个固定量的时间，因此，称它们为其时间是囿界于常数的运算。另一类运算则不然，它们可能是由一些更基本的任意长序列的运算所组成。例如，两个字符串的比较运算可以是一系列字符比较指令，而字符比较指令又可使用移位和位比较指令。如果说比较一个字符的时间囿界于常数，那么比较两个字符串的时间总量就取决于它们的长度。

第二件要做的事是确定能反映出算法在各种情况下工作的数据集，即是要求我们编造出能产生最好、最坏和有代表性情况的数据配置，通过使用这些数据配置来运行算法，以了解算法的性能。这部分工作是算法分析最重要和最富于创造性的工作之一。有关这方面更深入的叙述将在以后讨论一些具体算法时再进行。

对一个算法要作出全面的分析可分成两个阶段来进行，即**事前分析**(a priori analysis)和**事后测试**(a posteriori testing)。由事前分析，求出该算法的一个时间限界函数(它是一些有关参数的函数)。而由事后测试收集此算法的执行时间和实际占用空间的统计资料。假设在程序中的某个地方，有语句 x←x+y。在给出某种初始数据作为输入的情况下，如果要确定执行这条语句的时间总量，这基本上要求两项信息，该语句的**频率计数**(frequency count)(即，该语句执行的次数)和每执行一次该语句所需要的时间。这两个数的乘积就是时间总量。由于每次执行都依赖于所用的机器和程序设计语言以及它的编译程序，因此事前分析只限于确定每条语句的频率计数。频率计数与所用的机器无关，而且独立于写这算法的程序设计语言，从而可由

算法直接确定。

例如，考虑下面(a)、(b)、(c)三个程序段：

```
                                                    for i←1 to n do
                                    for i←1 to n do     for j←1 to n do
x←x+y                                   x←x+y               x←x+y
                                    repeat              repeat
                                                    repeat
```

(a)　　(b)　　(c)

对于每个程序段，假定语句 x←x+y 仅包含在当前可以看得见的循环之中，那末，在程序段(a)，此语句的频率计数为 1，在程序段(b)是 n，在程序段(c)是 n^2。这些频率计数显然具有不同的数量级。就算法分析而言，一条语句的数量级指的是执行它的频率，而一个算法的数量级则指的是它所有语句执行频率的和。假定解同一个问题的三个算法其数量级分别为 n，n^2 和 n^3。比较起来，自然更乐意采用第一个算法，因为它比其余两个算法要快。例如，若 n=10，在假定所有基本运算都具有相等的工作时间的情况下，这些算法则需分别执行 10，100 和 1000 个单位时间。由以上分析可知，确定一个算法的数量级是十分重要的，它在本质上反映了一个算法所需要的计算时间。

在实际的算法分析中，往往要分析出一个算法的计算时间或频率总数，并用某种函数来表示，譬如说用一个多项式来表示。但是，算法本身可能相当复杂以及其它许多因素，使得在事前分析阶段根本就写不出这个多项式的完整形式，甚至连最高次项的系数都不能写出，而只能写出该项的次数并判断出这个多项式与最高次项的关系。尽管这是件令人非常遗憾的事，但幸运的是这种关系仍反映了算法在计算时间上的基本特性，关于这一点，稍后即可看出，因此在算法的事前分析阶段，一般都致力于确定这种关系。下面给出这种关系的数学描述。

1.2.1 计算时间的渐近表示

假设某算法的计算时间是 f(n)，其中变量 n 可以是输入或输出量，也可以是两者之和，还可以是它们之一的某种测度(例如，数组的维数，图的边数等等)。g(n)是在事前分析中确定的某个形式很简单的函数，例如，n^m，$\log n^*$，2^n，n! 等。它是独立于机器和语言的函数，而 f(n)则与机器和语言有关。

定义 1.1　如果存在两个正常数 c 和 n_0，对于所有的 $n \geqslant n_0$，有

$$|f(n)| \leqslant c|g(n)|$$

则记作 f(n)=O(g(n))。

因此，当说**一个算法具有 O(g(n))的计算时间**时，指的是如果此算法用 n 值不变的同一类数据在某台机器上运行时，所用的时间总是小于|g(n)|的一个常数倍。所以 g(n)是计算时间 f(n)的一个上界函数，f(n)的数量级就是 g(n)。当然，在确定 f(n)的数量级时总是试图求出最小的 g(n)，使得 f(n)=O(g(n))。现在来证明一个很有用的定理。

定理 1.1　若 $A(n)=a_m n^m+\cdots+a_1 n+a_0$ 是一个 m 次多项式，则 $A(n)=O(n^m)$。

证明　取 $n_0=1$，当 $n \geqslant n_0$ 时，利用 A(n)的定义和一个简单的不等式，有

$$|A(n)| \leqslant |a_m| n^m + \cdots + |a_1| n + |a_0|$$

---

* 除非另有声明，本书所用的对数均以 2 为底。

$$\leqslant (|a_m| + |a_{m-1}|/n + \cdots + |a_0|/n^m)n^m$$
$$\leqslant (|a_m| + \cdots + |a_0|)n^m$$

选取 $c=|a_m|+\cdots+|a_0|$，定理立即得证。

事实上，只要将 n_0 取得足够地大，可以证明只要 c 是比 $|a_m|$ 大的任意一个常数，此定理都成立。

这个定理表明，变量 n 的固定阶数为 m 的任一多项式，与此多项式的最高阶 n^m 同阶。因此计算时间为 m 阶的多项式的算法，其时间都可用 $O(n^m)$ 来表示。例如，若一个算法有数量级为 $c_1n^{m_1}, c_2n^{m_2}, \cdots, c_kn^{m_k}$ 的 k 个语句，则此算法的数量级就是 $c_1n^{m_1}+c_2n^{m_2}+\cdots+c_kn^{m_k}$。由定理 1.1，它等于 $O(n^m)$，其中 $m=\max\{m_i|1\leqslant i\leqslant k\}$。

为了说明数量级的改进对算法有效性的影响，下面举一例子：假设有解决同一个问题的两个算法，它们都有 n 个输入，分别要求 n^2 和 nlogn 次运算，那末，当 n＝1024 时，它们需要 1048576 和 10240 次运算。如果每执行一次运算的时间是 1μs，则在输入相同的情况下，第一个算法大约需时 1.05s，第二个算法需要 0.01s。如果将 n 增加到 2048，则运算次数就变成 4194304 和 22528，即大约需要 4.2s 和 0.02s。这表明在 n 加倍的情况下，一个 $O(n^2)$ 的算法要用 4 倍长的时间来完成，而一个 O(nlogn) 的算法则只要两倍多一点的时间即可完成。在一般情况下，n 值为数千是很常见的，因此，数量级的大小对算法有效性的影响是决定性的。

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则称为**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(即，零次多项式)来限界，而一个时间为 $O(n^2)$ 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是最为常见的，其关系为

$$O(1)<O(\log n)<O(n)<O(n\log n)<O(n^2)<O(n^3)$$

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

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

其中，最常见的是时间为 $O(2^n)$ 的算法。当 n 取得很大时，指数时间算法和多项式时间算法在所需时间上非常悬殊，因为根本就找不到一个这样的 m，使得 2^n 囿界于 n^m。换言之，对于任意的 $m\geqslant 0$，总可以找到 n_0，当 $n\geqslant n_0$ 时，有 $2^n>n^m$。因此，只要有人能将现有指数时间算法中的任何一个算法化简为多项式时间算法，那就取得了一个伟大的成就。

图 1.1 和表 1.1 显示了当常数取为 1 时的六种典型的计算时间函数的增长情况。从中可以看出，O(logn)、O(n) 和 O(nlogn) 比另外三种时间函数的增长率慢得多。

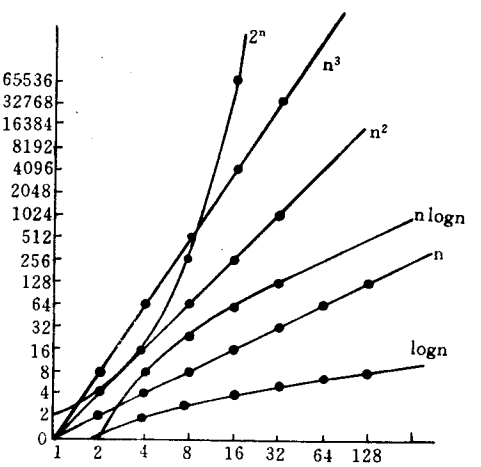

图 1.1　一般计算时间函数的曲线

由这些结果可看出，当数据集的规模(即 n 的取值)很大时，要在现代计算机上运行具有比 O(nlogn) 复杂度还高的算法往往是很困难的。尤其是指数时间算法，它只有在 n 值取得非常小时才实用。要想在顺序处理机上扩大所处理问题的规模，有效

的途径是降低算法计算复杂度的数量级，而不是提高计算机的速度。

表 1.1 计算时间函数值

logn	n	nlogn	n^2	n	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

符号 O 作为算法性能描述的工具，它表示计算时间的上界函数。为了进一步刻划算法的性能特性，有时也希望确定时间的下界函数，为此，引进另一个数学符号。

定义 1.2 如果存在两个正常数 c 和 n_0，对于所有的 $n>n_0$，有

$$|f(n)|\geqslant c|g(n)|$$

则记为 $f(n)=\Omega(g(n))$。

在某些情况下，某算法的计算时间既有 $f(n)=\Omega(g(n))$又有 $f(n)=O(g(n))$，即 g(n)既是 f(n)的上界又是它的下界。为简便起见，引进另一个数学符号来表示这种情况。

定义 1.3 如果存在正常数 c_1，c_2 和 n_0，对于所有的 $n>n_0$，有

$$c_1|g(n)|\leqslant|f(n)|\leqslant c_2|g(n)|$$

则记为 $f(n)=\Theta(g(n))$。

一个算法的 $f(n)=\Theta(g(n))$意味着该算法在最好和最坏情况下的计算时间就一个常因子范围内而言是相同的。这几种数学符号要经常使用，希望大家在此就能明确它们各自的含义。

上面仅对算法的计算时间特性作了较详细的介绍，算法计算空间的分析也可作完全类似的讨论，故在此从略。

1.2.2 常用的整数求和公式

在算法分析中，在确定语句的频率时，经常会遇到以下形式的表达式：

$$\sum_{g(n)\leqslant i\leqslant h(n)} f(i) \tag{1.1}$$

其中，f(i)是一个带有有理数系数且以 i 为变量的多项式。这表达式最常用到的是以下几种形式：

$$\sum_{1\leqslant i\leqslant n}1 \quad \sum_{1\leqslant i\leqslant n}i \quad \sum_{1\leqslant i\leqslant n}i^2 \tag{1.2}$$

由于它们都是有限求和，因此可列出它们的求和公式。由此可以容易地看出，第一个多项式的和数为 n。为以后使用方便，下面直接写出其余多项式的求和公式：

$$\sum_{1\leqslant i\leqslant n} i = n(n+1)/2 = \Theta(n^2) \tag{1.3}$$

$$\sum_{1\leqslant i\leqslant n} i^2 = n(n+1)(2n+1)/6 = \Theta(n^3) \tag{1.4}$$

通式是

$$\sum_{1\leqslant i\leqslant n} i^k = \frac{n^{k+1}}{k+1} + \frac{n^k}{2} + \text{低次项} = \Theta(n^{k+1}) \tag{1.5}$$

1.2.3 作时空性能分布图

事后测试是在对算法进行设计、确认、事前分析、编码和调试之后要做的工作，以确定程序所耗费的精确时间和空间，即作时空性能分布图。由于事后测试与所用计算机密切相关，故在此只对这一阶段所要进行的基本工作和若干注意之点概略地作些介绍。

以作时间分布图为例，要确定算法精确的计算时间，首先必须在所用计算机上配置一台能读出时间的时钟。有了时钟还必须了解它的精确程度以及计算机所用操作系统的工作方式，因为前者随所用计算机的不同而有相当大的差异，如果在一台时钟精确度不高的计算机上运行需时很少（譬如说比时钟的误差值还小）的程序，那末，所得的计时图只不过是一些“噪声”，时间分布性能完全被淹没在这些“噪声”之中。如果后者是以多道程序或分时方式工作的操作系统，则在取得算法工作的可靠时间上会出现困难，尤其对于那些时钟计时中包含了换出磁盘上的用户程序要用的那部分时间的操作系统而言。由于时间随当前记入系统的用户数而变化，因此无法确定算法本身所花去的时间。

为解决因时钟误差而引起的“噪声”问题，下面推荐两种可供选用的方法：一种是增加输入规模，直至得到算法所需的可靠的时间总量；第二种方法是取足够大的 r，将此算法反复执行 r 次，然后用 r 去除总的时间。

在解决了计时方面的具体技术问题之后，就可考虑如何作出时间性能分布图。对于事前分析为 $\Theta(g(n))$ 时间的算法，应选择按输入不断增大其规模的数据集。用这些数据集在计算机上运行程序，从而得到使用这些数据集情况下算法所耗费的时间，并画出这一数量级的时间曲线。如果这曲线与事前分析所得曲线形状基本吻合，则印证了早先分析的结论。而对于事前分析为 $O(g(n))$ 时间的算法，则应在各种规模的范围内分别按最好、最坏和平均情况的那些数据集独立运行程序，作出各种情况的时间曲线，并由这些曲线来分析最优的有效逻辑位置。

另外，如果为了解决某一个问题，分别设计了几种具有同一数量级的不同算法，或者为加快某种算法速度，作了在同一数量级情况下的一些改进，那末，只要在输入相同数据集的情况下作出它们的时间分布图就可比较出哪一个算法更快些。

1.3 用 SPARKS 语言写算法

为了便于表达算法所具有的特性，最好能用程序设计语言将算法写出来。选用语言最起码的一条要求是，由该语言所写出的每一个合法的句子都必须具有唯一的含义。程序就是用程序设计语言所表示的算法。本书中所出现的过程、子程序这样一些术语，有时也作为程序的同义词。选用何种语言来写算法呢？由于关心的是算法本身的基本思想和基本步骤，同时希望选用的语言简明、够用，写出的算法便于阅读并能容易地用人工或机器翻译成其它实际使用的程序设计语言，因此，这里选用的是 SPARKS 语言。它与 ALGOL 语言和 PASCAL 语言的形式很接近，凡是掌握了一门高级程序设计语言的人都能很快看懂并掌握 SPARKS 语言。

SPARKS 语言的基本数据类型是整型、实型、布尔型和字符型。变量只能存放单一类型的值，它可以用下述形式来说明其类型：

integer x,y　　**boolean** a,b　　**char** c,d

在 SPARKS 中，有特殊含义的标识符作为保留字来考虑，用黑体字表示。给变量命名的规则是，以字母起头，不允许使用特殊字符，且不要太长；不允许与任何保留字重复。一行可以有