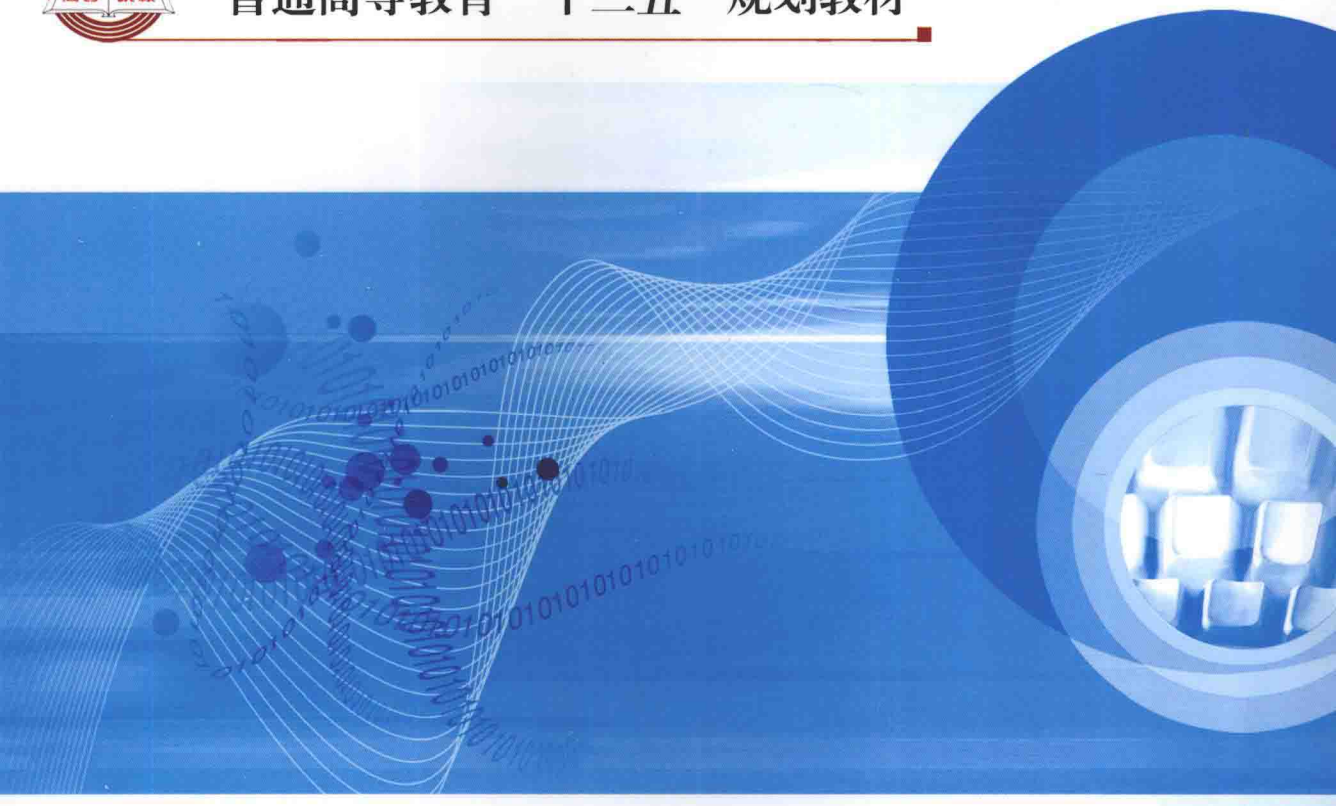
普通高等教育“十二五”规划教材



算法设计与分析

张威　葛琳琳　王军　主编

中国石化出版社
HTTP://WWW.SINOPEC-PRESS.COM

普通高等教育“十二五”规划教材

算法设计与分析

张威　葛琳琳　王军　主编

中国石化出版社

内 容 提 要

本书讲解算法设计与分析的相关知识，首先介绍了算法基本概念、算法基础知识及数学工具，然后介绍一些经典的算法，包括递归与分治法、贪心法、动态规划、回溯法、分支限界法、概率算法及 NP 问题。本书以算法设计策略为知识单元，结合数据结构中的实例，系统地介绍计算机算法的设计与分析技巧。另外，书中还配有大量的习题及上机试题，以便读者检验和强化所学的知识，起到事半功倍的效果。

本书内容丰富、结构清晰。采用 C/C++语言描述算法，可读性强；可以作为普通高校本科和研究生的教材，也适合广大工程技术人员在实际工作中学习参考。

图书在版编目(CIP)数据

算法设计与分析 / 张威，葛琳琳，王军主编. —北京：中国石化出版社，2015.8
普通高等教育“十二五”规划教材
ISBN 978-7-5114-3468-5

Ⅰ.①算… Ⅱ.①张… ②葛… ③王… 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.①TP301.6

中国版本图书馆 CIP 数据核字(2015)第 165184 号

中国石化出版社出版发行
地址：北京市东城区安定门外大街 58 号
邮编：100011　电话：(010)84271850
读者服务部电话：(010)84289974
http://www.sinopec-press.com
E-mail:press@sinopec.com
北京科信印刷有限公司印刷
全国各地新华书店经销
*
787×1092 毫米 16 开本 15.25 印张 357 千字
2015 年 8 月第 1 版　2015 年 8 月第 1 次印刷
定价：32.00 元

❖ 前　言 ❖

算法研究是计算机科学的核心问题之一。《算法设计与分析》是计算机软、硬件专业的基础课程，本课程的目的是通过对计算机领域中常见而有代表性算法的研究，理解和掌握算法设计的主要方法，培养对算法复杂性的分析能力，为独立地设计算法和对给定算法进行复杂性分析奠定坚实的知识基础。

计算机系统中的任何软件，都是按一个特定的算法来予以实现的。算法性能的好坏，直接决定了所实现软件性能的优劣。如何判定一个算法的性能？用什么方法来设计算法？所设计的算法需要多少运行时间？多少存储空间？在实现一个软件时，这些都是必须予以解决的。计算机的操作系统、语言编译系统、数据库管理系统以及各种各样的计算机应用系统软件，都离不开用具体的算法来实现。因此，算法设计与分析是计算机科学与技术的一个核心问题，也是大学计算机专业本科生及研究生重要的专业基础课程。通过《算法设计与分析》这门课程的学习，使读者能够掌握算法设计与分析的方法，利用这些方法去解决在计算机科学与技术中所遇到的各种问题，去设计计算机系统的各种软件中所可能遇到的算法，并对所设计的算法作出科学的评价。因此，《算法设计与分析》这门课程不仅对计算机专业的科学技术人员，而且对使用计算机的其他专业技术人员，都是非常重要的。

本书介绍算法的设计与分析的相关知识，全书共分9章。

第1章　算法概述：对分析算法的抽象表示、算法渐进复杂度以及如何对算法进行设计与分析作了简要的阐述。

第2章　常用的数学工具：对在算法分析处理过程中，需要的一些基本数学工具进行简单介绍。

第3章　递归与分治：递归技术是算法设计与分析的基础，学好递归技术为以后各章问题算法的求解打下坚实的基础；分治法作为一种算法设计策略是非常有价值的，可以求解许多算法问题。

第4章　贪心法：贪心法是一种重要的算法设计策略，它与动态规划算法的设计思想有一定的联系，但其效率更高。按贪心法设计出的许多算法能产生最优解。其中有许多典型问题和典型算法可供学习和使用。

第5章　动态规划：以具体实例详述动态规划算法的设计思想、适用性以及算法的设计要点。

第6章　回溯法：介绍其概念以及所要解决的问题。

第 7 章　分支限界法：介绍其概念以及所要解决的问题，它同第 6 章的回溯法适合于处理难解问题，其解题的思想各具特色，值得学习和掌握。

第 8 章　概率算法：介绍了概率算法的概念及其可以解决的一些常见的问题。

第 9 章　NP 问题：NP 问题是计算机科学理论中最重要的问题，本章简要介绍了 P 类与 NP 类问题以及 NP 完全问题。

本书不仅适合作为普通高校本科生和研究生的教材，也适合作为工程技术人员在实际工作过程中的技术参考书。

本书在编写过程中，借鉴了许多现行教材的宝贵经验，并得到了沈阳化工大学计算机科学与技术学院的王军院长和郭希旺博士的大力支持，在本书的资料收集整理过程中辽宁科技大学张卓老师等给予了很大的帮助，牟新宇和王梓敏同学在程序的编写和调试中做了大量的工作，在此一并表示感谢。

由于时间仓促，作者水平有限，书中错误或是不足之处敬请广大读者批评指正，使其能在使用中不断地得到改进，日臻完善。

❖ 目 录 ❖

1 算法概述

算法是计算机学科的一个重要分支，是计算机科学的基础，更是程序设计的基石。学习算法，一方面需要学习求解计算领域中典型问题的各种有效算法，在遇到问题时能灵活地应用所掌握的方法技巧；另一方面还要学习设计新算法和分析算法性能的方法，当没有现成可用的算法时，能够创造出有效的问题求解方法。本章是全书的基础，首先介绍算法的定义，继而介绍分析算法的工具，即时间复杂性和空间复杂性，主要介绍时间复杂性。第三节介绍算法设计与分析的步骤。第四节是通过对一个例子的讲解，给读者示范如何分析算法的时间复杂性和空间复杂性。最后对算法描述语言进行简介。通过本章的学习，需要掌握算法的概念；算法的复杂度分析与效率分析；了解算法的设计与分析步骤；了解算法描述 C 语言的基本使用。

1.1 算法概念

凡是有过程序设计经历的人都会对 N. Wirth 提出的公式：

算法+数据结构=程序

有着深刻的领悟——算法是程序的灵魂。

程序设计主要包括两个方面：行为设计与结构设计。行为设计是对要解决的问题，提出达到目的需要实施的一些步骤，并对这些步骤加以必要的细化，给予定义，在此基础上用某种方式完整地描述出来，就是算法设计，其结果就是算法；结构设计是针对所要解决的问题，对数据定义数据结构(包括物理结构和逻辑结构)。有了好的算法、合适的数据结构，再使用某种程序设计语言加以具体实现，即可得到程序。因此，算法是程序设计的灵魂，它在产生程序的过程中，占有重要的地位。

对于计算机科学来说，算法(Algorithm)的概念是至关重要的。例如在一个大型软件系统的开发中，设计出有效的算法将起决定性的作用。通俗地讲，算法是指解决问题的一种方法或一个过程，如图 1.1 所示。

从这个角度来讲，算法的研究由来已久。中国的珠算口诀可以说是非常典型的算法，它将复杂的计算(如除法)描述为一系列的算珠拨动操作。还有，计算两个整数的最大公约数的辗转相除法(欧几里德算法)也是算法早期研究的最重要成果。

对于算法，D. E. Knuth 给出了一个非形式化的定义：一个算法，就是一个有穷规则(指令)的集合。它为某个特定类型问题提供了解决问题的运算序列。

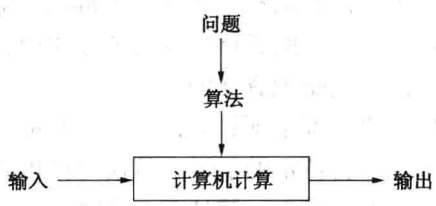


图 1.1 运用算法解决问题的过程

从这一定义可以引申出算法具有的五个特征。

(1) 有穷性(finiteness)

一个算法必须在执行有穷步之后终止，即必

须在有限的时间内完成。如何理解有限的时间呢？如果一个算法需要在计算机上运行千万年，那么这样的算法，它所需要的时间是有限的，但是这样的算法没有实用价值。所谓需要有限时间的算法，应该理解成人们可以接受的时间内完成的算法，也可以理解为算法中每条指令的执行次数是有限的，执行每条指令的时间也是有限的，即算法必须总能在执行有限步骤之后终止。

例如，计算行列式的值。依线性代数中对 n 阶行列式的定义：n 阶行列式的值等于所有取自于不同行、不同列的 n 个元素的乘积 a_{1j_1}，a_{2j_2}，…，a_{nj_n} 的代数和，这里 j_1，j_2，…，j_n 是 1，2，…，n 的一个排列。每一项按下列规则带有符号：当 j_1，j_2，…，j_n 是偶排列时带正号；当 j_1，j_2，…，j_n 是奇排列时带负号。n 阶行列式一共有 $n!$ 项，计算它需做 $n!$（$n-1$）个乘法。按这个定义设计的求值过程，只能是一种理论上可行的计算方法，不是一种实用的计算方法，更不能称做为算法。

再如，货郎担问题（售货员路线问题）。设货郎在一天内要到 n 个城市去推销货物，已知从一个城市到其他城市的费用，求总费用最少的路线。

用穷举法可以求得最少费用的路线，花费的时间按 $n!$ 增长，其时间复杂性是 $O(n!)$（与前一例有同样的时间复杂度）。这个解法实际上是不可行的。当 $n=20$ 时，$20!=2\times10^{18}$，假设计算每条路线需要 CPU 时间为 10^{-7}s，则总共需要 7000 年才能得到结果。这样的运行时间对于人们来说是不可容忍的。因此，算法的有穷性实际上应包含合理的执行时间的含义。

（2）确定性（definiteness）

算法的每一步必须是确定的。即组成算法的每条指令是清晰的，不能有二义性，是无歧义的，让计算机执行的每一步，不允许有模棱两可的解释，不允许有多义性。比如，让计算机下一步执行“将 m 或 n 与 y 相乘”之类的运算就存在二义性。因为按此指令，究竟是把 m 与 y 相乘，还是把 n 与 y 相乘不确定。

（3）可行性（effectiveness）

算法的可行性是指算法中每条指令都有明确的定义，是可行的，可在有限的时间内做完。它包括两个方面：一是算法所描述的每一步必须是基本的、有意义的。例如，在有限时间内完成的算法中，不允许做除法时，除数为 0；在实数范围内不能求一个负数的平方根等。二是算法执行的结果要能达到预期的目的。比如使 $\sin x$ 的近似值的绝对值大于 1 的求解过程，不具有可行性，它不是算法。

（4）输入（input）

一个算法有零个或多个由外部提供的量作为算法的输入。算法处理的数据来源于两种方式：一种是从外部设备获得数据的输入，另一种是由算法中自己产生被处理的数据。算法的零个输入指不需要从外部设备获取数据，数据来源于第二种方式。

（5）输出（output）

一个算法必须有一个或多个输出。输出的数据是算法对数据加工的结果。既然算法是为解决问题而设计的具体实现的若干步骤，那么算法实现的最终目的就是要获得问题的解。没有输出的算法是无意义的。

所有算法都必须具有以上 5 个特征。算法的输入是一个算法在开始前所需的最初的量，这些输入取自特定的值域。算法可以没有输入，但算法至少应产生一个输出，否则算法便失去了它存在的意义。算法是一组指令序列。一方面，每条指令的作用必须是明确、无歧义的。在算法中不允许出现诸如“计算 5+3 或计算 7-3”这样让计算机自己选择的指令。另一

方面，算法的每条指令必须是可行的。对一个计算机算法而言，可行性要求一条算法指令应当最终能够由执行有限条的计算机指令来实现。例如，一般的整数算术运算是可行的，但如果像 3÷7 这样的值就需要由无穷的十进制展开的实数表示，就不是可行的。因此，概括地说，算法是由一系列明确定义的基本指令序列所描述的，求解特定问题的过程。它能够对合法的输入，在有限时间内产生所要求的输出。如果取消了有穷性的限制，则只能称为计算机实现的计算过程(computational procedure)。当一个算法使用计算机程序设计语言描述时，就是程序(program)。

算法必须可终止，而计算机程序并没有这一限制。例如，一个操作系统是一个程序，却不是一个算法，一旦运行，只要计算机不关闭，操作系统程序就不会终止运行。所以，操作系统程序是使用计算机语言描述的一个计算过程。算法与程序(Program)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有穷性。例如操作系统，它是一个在无限循环中执行的程序，因而不是一个算法。然而我们可把操作系统的各种任务看成是一些单独的问题，每一个问题由操作系统中的一个子程序通过特定的算法来实现。该子程序得到输出结果后便终止。

算法是计算机学科的一个重要分支，计算机科学的基础，更是程序的基石。学习算法，一方面需要学习求解计算领域中典型问题的各种有效算法，在遇到问题时能灵活地应用所掌握的方法技巧；另一方面还要学习设计新算法和分析算法性能的方法，当没有现成可用的算法时，能够创造出有效的问题求解方法。

算法层出不穷，变化万千，其对象数据和结果数据名目繁多，不胜枚举。最基本的有布尔值数据、字符数据、整数和实数等；稍复杂的有向量、矩阵、记录等；更复杂的有集合、树和图，还有声音、图形、图像等数据。

算法的运算种类五花八门、多姿多彩。最基本的有赋值运算、算术运算、逻辑运算和关系运算等；稍复杂的有算术表达式和逻辑表达式等；更复杂的有函数值计算、向量运算、矩阵运算、集合运算，以及表、栈、队列、树和图上的运算等；此外，还可能有以上列举的运算的复合和嵌套。

高级程序设计语言在数据、运算和控制三方面的表达中引入许多使之十分接近算法语言的概念和工具，具有抽象表达算法的能力。高级程序设计语言的主要好处是：

(1) 高级语言更接近算法语言，易学、易掌握，一般工程技术人员只需要几周时间的培训就可以胜任程序员的工作；

(2) 高级语言为程序员提供了结构化程序设计的环境和工具，使得设计出来的程序可读性好、可维护性强、可靠性高；

(3) 高级语言不依赖于机器语言，与具体的计算机硬件关系不大，因而所写出来的程序可移植性好、重用率高；

(4) 把复杂琐碎的事务交给编译程序，所以自动化程度高，开发周期短，程序员可以集中时间和精力从事更重要的创造性劳动，提高程序质量。

算法从非形式的自然语言表达形式转换为形式化的高级语言是一个复杂的过程，仍然要做很多繁杂琐碎的事情，因而需要进一步抽象。

对于一个具体的数学问题，想设计它的算法，应该先选用一个适合此问题的数据模型。接下来，要搞清楚此问题的数据模型在已知条件下的初态和要求的终态，以及这两个状态之间的隐含关系。然后确定从数据模型的已知初态到达要求的终态所需的运算步骤。这些运算

步骤就构成了求解该问题的具体算法。

按照自顶向下逐步分解求精的原则，在确定运算步骤的时候，应该考虑算法顶层的运算步骤，然后再考虑底层的运算步骤。定义在数据模型级上的运算步骤被定义成顶层运算步骤，或称之为宏观步骤。它们是组成算法的主干部分，而表达这部分算法通常使用非形式化的自然语言。这其中所涉及的数据是数据模型中的变量，暂不关心其数据结构；而所涉及的运算将数据模型中数据变量或者作为运算对象，或作为运算结果，甚至是二者兼而为之，这些运算被简称为定义在数据模型上的运算。由于暂时不需要关心变量的数据结构，这些运算都带有抽象的性质，而不含运算细节。顶层抽象运算的具体实现被称作底层运算步骤。它们是依赖于数据模型结构及其具体表示。所以，数据模型的具体表示和定义在该数据模型上运算的具体实现是底层运算步骤的两个主要部分。将顶层运算步骤称为宏观步骤，那么底层运算就可以被认为是微观运算。二者的关系可概括为底层运算是顶层运算的细化，底层运算为顶层运算服务；顶层决定底层，顶层又是通过底层调用实现的。为了将顶层算法与底层算法相隔开，使二者在设计时独立，必须抽象化二者的接口。限制底层只能通过接口为顶层服务，顶层也只能通过接口调用底层运算。此接口就是抽象数据类型(Abstract Data Types，ADT)。

算法设计的重要概念之一就是抽象数据类型。实际上，它是算法的一个数据模型和定义在该模型上的作为算法构件的一组运算。这样定义抽象数据类型就明确地把数据模型与该模型上的运算紧密地联系起来。事实确实如此，一方面，数据模型上的运算依赖于数据模型的具体表示，数据模型上的运算以数据模型中的数据变量为运算对象或运算结果；另一方面，确定了数据模型的具体表示，也确定了数据模型上运算的具体实现，运算的效率就随之而定了。那么，问题是如何选择数据模型的具体表示使该模型上各种运算效率都尽可能高呢？显然，针对不同的运算组，为达到使该运算组中所有运算的效率都尽可能高的目的，其相应的数据模型的具体表示将有所不同。在这个前提下，数据模型的具体表示又要依赖于数据模型上定义的运算。特别地，当不同运算的效率互相牵制时，事先还须将所有的运算使用频度做相应的排序，使所选择的数据模型的具体表示首先保证使用频度较高的运算效率较高。抽象数据类型概念产生的背景和依据就是数据模型的定义在该模型上的运算之间存在的这种密不可分的联系。

使用抽象数据类型带给算法设计的好处主要有：

(1) 算法顶层设计与底层实现分离，使得在进行顶层设计时不考虑它所用到的数据，运算表示和实现；反过来，在表示数据和实现底层运算时，只要定义清楚抽象数据类型而不必考虑在什么场合引用它。这样做就使算法设计的复杂性降低了，同时条理性也随之增强，既有助于迅速开发出程序原型，又使开发过程少出差错，程序可靠性提高。

(2) 算法设计与数据结构设计相隔开，允许数据结构自由地选择，从其中做比较，优化算法的效率。

(3) 数据模型和该模型上的运算统一在抽象数据类型中，反映它们之间内在的互相依赖和互相制约的关系，便于空间和时间耗费的折衷，灵活地满足用户要求。

(4) 由于顶层设计和底层实现局部化，在设计中出现的差错问题也是局部的，因而容易查找以及容易纠正，在设计中经常要做的增、删、改也都是局部的，因而也都容易进行。因此，用抽象数据类型表述的算法具有很好的可维护性。

(5) 算法自然呈现模块化，抽象数据类型的表示和实现可以封装，便于移植和重用。

(6) 为自顶向下逐步求精和模块化提供了有效途径和工具。

(7) 算法结构清晰，层次分明，便于算法正确的证明和复杂性的分析。

综上所述，所谓算法是一组定义严谨的运算顺序规则，并且每一个规则都是有效的，且是明确的。这组规则在有限的时间内终止。

解决问题的目的不只是获得算法。算法有“好”有“坏”，那么好坏的标准是什么？也就是说，用什么样的尺度去评价一个算法的好坏。下面一节介绍这一内容。

1.2 算法的复杂度

解决同一个问题可以有多个算法，如何确定某个算法是最优的呢？这就要看它的复杂性，一个好的算法不仅要具有正确性(correctness)、简明性(simplicity)、高效性(efficiency)，还要具备最优性(optimality)。一个算法复杂性的高低体现在运行该算法所需要的计算机资源的多少上，所需资源越多，我们就说该算法的复杂性越高；反之，则该算法的复杂性越低。最重要的计算机资源是时间和空间(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。一般情况，要选择算法复杂性低的算法。

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在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，是在选用算法时遵循的一个重要原则。因此，算法的复杂性分析对算法的设计或选用有着重要的指导意义和实用价值。

算法分析是对一个算法所消耗的资源进行估算。在串行算法中资源消耗指时间、空间的耗费。时空是一对矛盾。算法分析的目的，就是通过对同一问题的多个不同算法进行时空耗费这两方面的分析。在时空资源兼受限制时，找出时空代价的平衡点，即时间相对少，空间耗费也相对少的算法。当两种资源之一受限时，选出适合这一限制的算法。算法分析是一项考验智力，而且是很有实际意义的工作。它力求完美，而这种追求正是优秀程序员所具有的品质。

如何计算算法的时空消耗？

一种办法是事后分析。具体地说，对于问题 P 的算法 A，在计算机上分别运行 A 对应的程序，输入适当的数据，测试程序 A 的开销，这一方法似乎很合理，细想一下其实不尽人意。

第一，如此测试的结果与程序的运行环境有关。如计算机主频、总线和外部设备等，都会影响测试结果。

第二，语言的编译系统对生成的机器代码的质量会产生很大的影响，从而影响运行的速度。

第三，测试结果与选取的样本数据有关。算法的时空耗费是 n 的函数，它的结果应是在 n 足够大时才有意义。对于不同的 n 个数据，运行的结果是不一样的。欲要获得真实可靠的结果，必须科学地选取样本数据，而要做到这一点，并不是一件简单的事情。

总之，这种事后分析是面向机器，面向程序员，面向语言的。从理论上讲，只有在标准环境下进行算法测试，得到的资源耗费才是可信的。

这些因素与问题 P 的两种算法的差异无关。为了公平起见，同一问题的两种算法所对

应的两个程序，应该在同样的条件下用同一个编译器编译，在同一台机器上运行，并且两次编程时所花费的精力也应尽可能地相等，使算法的实现“等效”。如果做到这几点，上面提到的那些因素就不会对结果产生影响，因为它对于每一个算法都是公平的。

但是，仅凭实验来比较算法，很有可能因为一个程序比另一个程序“写得好”，而使算法的真正质量没有得到很好的体现。再者，可能两种算法，通过艰苦的编程，测试后都超过了预算的耗费，那么只有重新设计适合预算耗费的新算法。

另一种方法是事前分析的方法，称为渐进算法分析(asymptotic algorithm analysis)，简称为算法分析，这种分析方法是求得算法耗费的限界函数。它可以估算出当问题规模变大时，一个算法以及实现它的程序的效率和耗费。

在介绍这种方法之前，首先对算法实现的机器类型作出假定，因为它对求得算法的耗费影响很大。从算法的本质特征出发，应选一种确定的机械装置作为前提进行耗费计算。这种机械装置最理想的形式模型是图灵机(Tuning machine)和 RAM 机(Random access machine，即随机存取机器)，但是我们可能会对这些严谨的、形式化的定义产生费解。现假定关于算法的讨论在一台通用机的基础上进行。这台通用机就是平时使用的顺序处理机。它每一次执行算法中的一条指令，有足够大的随机存储器，对每一个单元中数据的访问时间是固定的，在这样的前提下，进行事前分析的讨论。

对于时间耗费的计算来说，在这种理想的通用机模型上，用尺度来代替运行时间。这个尺度是处理一定“规模”的输入时，算法所需要执行的“基本操作”数。

【例 1.1】 查找一维 n 元整数数组中最大元的算法。

算法(1.1)

```
1. procedure findmax(a, n)
2. /输入：长度为 n 的数组 a。输出：a 中元素的最大者在 a 中的位置。/
3. begin
4.    curmax <— a[1];
5.    for   i <— 2   to   n   do
6.       if   a[i] > curmax   then
7.          curmax <— a[i];
8.    return(i)
9. endp
```

在这个算法中，规模指输入量的数目，即数组的长度。基本操作的选取应满足该操作具有这样的特征：完成该操作所需时间与具体的输入无关。在 findmax 中，基本操作可选择元素间的比较。因为完成比较操作所需时间与输入无关，可记作常量 C。当然在 findmax 中，也可选择其他操作作为基本操作，比如选择 i 是否有效的判断作为基本操作也可以，只要这个选择是遵循了“基本操作”的定义亦可。事实上可以看到，元素间的比较与判断 i 值是否有效在 findmax 中的地位是平等的。

由于时间耗费是规模的函数，这个函数可记为 $T(n)$。算法(1.1)的时间耗费：$T(n)=Cn$，其中 C 为常数。它反映了时间耗费对 n 的增长率。再看下面程序段。

【例 1.2】 求 a 阵 a 的和。

```
1. sum <— 0;
2. fori <—1   to   n   do
```

3. for j <—1 to n do

4. sum <— sum + a[i , j]

其中 a 是行列数为 n 的方阵。

该算法中规模为 n^2，基本操作是做加法，时间耗费：$T(n)=Cn^2$，其中 C 为常数。把时间耗费为 Cn 称为线性增长率，Cn^2 称为二次增长率，$C2^n$ 称为指数增长率。

1.2.1 算法三性态

通常，分析算法要从它的三个性态考虑，即最好性态、最坏性态和平均性态。对于某些算法，即使问题规模相同，如果输入数据不同，其时间开销也不同。例如，现在要从一个 n 元的一维数组中找出一个给定的 K(假设该数组中有且仅有一个元素值为 K)。顺序检索法将从第一个元素开始，依次检索每一个元素，直到找到 K 为止。一旦找到了 K，算法也就完成了。要找到给定的 K，必须检查每一个元素的值。这样，顺序检索法的时间开销可能在很大的一个范围内浮动。数组中的第一个元素可能恰恰就是 K，于是只要检索一个元素就行了。在这种情况下，运行时间很短。这叫做算法的最好情况(best case)——顺序检索算法不可能执行比检索一个元素更少的操作。另一种情况，如果数组的最后一个元素是 K，运行时间就会相当长，因为这个算法要检查所有的 n 个元素。这是算法的最坏情况(worst case)——该算法不可能检索 n 个以上的元素。如果用一个程序来实现顺序检索法并用该程序对许多不同的 n 元数组检索，或者在同一个数组中检索不同的 K 值，就会发现，平均检索到整个数组的一半就能找到 K。也就是说，这种算法平均要检索 $n/2$ 个元素，我们称之为算法的平均(average case)情况时间代价。

分析一种算法时，应该研究最好、最坏还是平均情况呢？一般来说，我们对最好情况没有多大兴趣，因为它发生的概率太小，而且对于条件的考虑太乐观了，不具有普遍性。换言之，最好情况不能作为算法性能的代表。不过，也有一小部分情况，最好情况分析是有用的——尤其是当最好情况以现概率较大的时候，可以用最好情况运行时间来分析该算法，非常迅速而有效。

那么最坏情况呢？分析最坏情况有一个好处：它能让你知道算法至少能运行得多快。这一点在实时系统中尤其重要。例如空运处理系统，在这个系统中，一个“绝大部分”情况下能管理 n 架飞机的算法，如果它不能在规定的时间内管理来自于同一方向的 n 架飞机，那么它是不能被接受的。那么，最坏情况复杂度 $T(n)$ 可表示如下：

$$T(n)=\max_{I\in D_n}(I)$$

即 $T(n)$ 是算法在任何规模为 n 输入时所执行的基本运算的最大次数。可以看出，$T(n)$ 的计算较平均复杂度 $T_{avg}(n)$ 的确定往往方便得多。

在另外一些情况下，特别是程序要对许多不同的输入运行多次时，最坏情况分析就不适合用来衡量一种算法的性能了。通常人们会更希望知道平均情况的时间代价，也就是说，当输入规模为 n 时算法的“典型”表现。可惜，平均情况分析并不总是可行的。首先，它要求人们清楚数据是如何分布的。例如，上面提到过顺序检索算法平均情况下要检查数组中一半的元素。但是这是基于 K 在数组中每个位置出现概率相等的假设之上的。如果这个假设不成立，那么算法的平均情况就不一定是检查一半的元素。

总之，在实时系统中，我们比较关注最坏情况算法分析。在其他情况下，通常考虑平均情况，只要知道计算平均情况所需要的输入数据的分布即可；否则，就只能求助于最坏情况分析了。

1.2.2 算法复杂度

算法的复杂度是算法运行所需要的计算机资源的量，需要时间资源的量被称为时间复杂度，需要的空间资源的量被称为空间复杂度。这个量应该集中反映算法的效率，并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要解的问题的规模、算法的输入和算法本身的函数。如果分别用 N、I 和 A 表示算法要解问题的规模、算法的输入和算法本身，而且用 C 表尔复杂度，那么，应该有 $C=F(N, I, A)$，其中 $F(N, I, A)$ 是一个由 N、I 和 A 确定的三元函数。如果把时间复杂度和空间复杂度分开，并分别用 T 和 S 来表示，应该有：$T=T(N, I, A)$ 和 $S=S(N, I, A)$。通常，让 A 隐含在复杂性函数名当中，因而将 T 和 S 分别简写为 $T=T(N, I)$ 和 $S=S(N, I)$。

1. 时间复杂度

现在研究时间复杂度，所面临的问题是如何将复杂性函数具体化，即对于给定的 N、I 和 A，如何导出 $T(N, I)$ 和 $S(N, I)$ 的数学表达式，来给出计算 $T(N, I)$ 和 $S(N, I)$ 的法则。下面以 $T(N, I)$ 为例，将复杂性函数具体化。

根据 $T(N, I)$ 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基本运算有 k 种，它们分别记为 O_1，O_2，…，O_k。又设每执行一次这些基本运算所需要时间分别为 t_1，t_2，…，t_k。对于给定的算法 A，设经统计，用到基本运算 O_i 的次数为 e_i，$i=1, 2, \cdots, k$。很清楚，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

$$T(N, I)=\sum_{i=1}^{k} t_i e_i(N, I)$$

其中，$t_i(i=1, 2, \cdots, k)$ 是与 N 和 I 无关的常数。

显然，不可能对规模 N 的每一种合法的输入 I 都去统计 $e_i(N, I)$，$i=1, 2, \cdots, k$。因此 $T(N, I)$ 的表达式还要进一步简化，或者说，只能在规模为 N 的某些成某类有代表性的合法输入中统计相应的 $e_i(N, I)$，$i=1, 2, \cdots, k$，评价其时间复杂性。

考虑时间复杂性的三性态，并分别记最坏情况、最好情况和平均情况下的时间复杂度为 $T_{\max}(N)$、$T_{\min}(N)$ 和 $T_{\text{avg}}(N)$。在数学上有

$$T_{\max}(N)=\max_{I \in D_N} T(N, I)=\max_{I \in D_N} \sum_{i=1}^{k} t_i e_i(N, I)=\sum_{i=1}^{k} t_i e_i(N, I^*)=T(N, I^*)$$

$$T_{\min}(N)=\min_{I \in D_N} T(N, I)=\min_{I \in D_N} \sum_{i=1}^{k} t_i e_i(N, I)=\sum_{i=1}^{k} t_i e_i(N, \bar{I})=T(N, \bar{I})$$

$$T_{\text{avg}}(N)=\sum_{I \in D_N} P(I) T(N, I)=\sum_{I \in D_N} P(I) \sum_{i=1}^{k} t_i e_i(N, I)$$

式中，D_N 是规模为 N 的合法输入的集合；I^* 是 D_N 中一个使 $T(N, I^*)$ 达到 $T_{\max}(N)$ 的合法输入；$\bar{I}$ 是 D_N 中一个使 $T(N, \bar{I})$ 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。

以上三种情况下的时间复杂性从不同角度来反映算法的效率，各有共局限性，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。

显然，对于算法工作量的度量，应该有助于认清一个算法的好坏，有助于比较同一问题的各种算法之优劣，以便可以确定一种算法的效率是否比另一些更高。当然，如果能够选择算法的实际执行时间，这种比较将非常方便。然而，这种时间却受到诸多因素的影响，如程

序所依赖的算法、问题的规模和输入的数据、计算机系统性能及所用的机器语言等。所以，用实际执行时间作为标准并不客观。因此，以考察算法所做的基本运算的数目来代替其实际执行时间，可以说明这种做法是合理的。

另外，一个算法往往由各种各样的运算构成，如加、减、乘、除和两个元素比较等。对于不同的运算，计算机执行每种运算的时间也不相同，有的少些，有的则多些。这样，就需要在算法中找出一种(有时也可多于一种)运算，使该算法的运行时间与此种运算的次数相匹配。也就是说，为了分析一个算法，需要将一种对于所研究的问题来说是“基本的”运算分离出来，忽略其他运算或细节(这些其余的运算细节可称为“薄记”工作)，而只计算算法所执行的基本运算的次数。

对于一个问题，通常可以选择一种基本运算。所谓基本运算就是可以用来衡量算法运行时间的主要运算。但有时会发现，还有一些其他的运算，其数目也是巨大而不可忽视的。例如，计算两矩阵的积时，如果有一种算法，只做了少量乘法，而做了大量加法，那么，把基本运算定为乘法和加法才是合理的。这样，尽管乘法和加法的运算时间不同，但总体不至于有太多的偏差，否则有可能丢失一些关于两种算法相对优点的有用信息(例如可以设计做较多的加法而做较少的乘法来改进算法)。

只要合理地选择基本运算，是算法执行的时间大致和基本运算次数成比例，就有了一种很好的测度来衡量一个算法的工作量，也就有了一个很好的标准来比较几种算法的优劣。

此处再考虑算法的平均性态。设 D_N 是对于所考虑的问题来说规模为 n 的输入集合，而 I 是 D_N 的一个元素，$P(I)$ 是 I 出现的概率。记 $t(I)$ 是算法在输入 I 时所执行的基本运算的次数，于是，算法的平均复杂度的实际定义为

$$T_{\text{avg}}(n) = \sum_{I \in D_n} P(I)t(I)$$

这里的 $t(I)$ 一般可仔细分析算法得到，而 $P(I)$ 则往往由经验或根据与使用算法的场合有关的信息得到，也可以做出假定。

2. 空间复杂度

算法在运行时所需的存储空间大小 $S(I)$ 称为算法的空间耗费。对于一个算法，空间耗费的度量方法通常定义为算法在运行时所占用内存单元的总数，即存放数据的变量单元、程序代码、工作变量、常数以及运行时引用型变量所占用空间和递归栈空间的总和。它与输入规模 n 有关，S 是 n 的函数。

由于空间复杂度概念与时间复杂度雷同，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度。但有一点必须清楚：除了程序和输入量以外所占用的额外空间量如果相对于问题规模来说是常数，则称该算法是“原地工作的”。原地工作往往是一种我们所期望的性质。

3. 复杂度的表示与估算方法

现以顺序检索数组中是否存在元素 x 算法为例，说明如何对算法进行复杂性分析。

【例 1.3】 设 A 是一个具有 n 个元素的数组，给定一个数 x，按下述算法判定 x 是否在数组 A 中出现?

```
1. int search(A[n], x)
2. {
3.    for(k=0; k<n; k++)
```

```
4.      if(A[k] == x)
5.          return k;
6.   return -1;
7. }
```

解：假定 x 可以出现在数组中的任意位置或者根本不在数组中，且所有输入的概率相同。试分析算法的时间复杂度。

顺序检索算法复杂度分析：

（1）最好情况复杂度分析　最理想的情况就是比较一次就找到了，显然是要查找的元素就是数组的第一个元素，$T_{\min}(n)=1$（次）。

（2）平均时间复杂度 $T_{\text{avg}}(n)$ 分析　首先应根据 x 在表中的位置将输入分类，这样，共有 $n+1$ 种输入情况：对于 $1\leqslant i\leqslant n$，记 I_i 表示 x 在表中第 i 个位置出现的情况；I_{n+1} 表示 x 不在表中的情况。

若记 $t(I_i)$ 为算法在输入 I_i 时所做的比较次数，显然有当 $1\leqslant i\leqslant n$ 时，$t(I_i)=i$，而且 $t(I_{n+1})=n$。

假定 q 是 x 出现在表中的概率，并假定 x 出现在每个位置的可能性均等。于是，对 $1\leqslant i\leqslant n$，有 $P(I_i)=q/n$，并且 $P(I_{n+1})=1-q$，故有

$$\begin{aligned}T_{\text{avg}}(n)&=\sum_{i=1}^{n+1}P(I)t(I_i)=\sum_{i=1}^{n}(q/n)i+(1-q)n\\&=(q/n)\sum_{i=1}^{n}i+(1-q)n\\&=(q/n)n(n+1)/2+(1-q)n\\&=q(n+1)/2+(1-q)n\end{aligned}$$

在这里，如果已知 x 在表中，即 $q=1$，则 $T_{\text{avg}}(n)=(n+1)/2$，说明算法平均要检查半张表；如果 $q=1/2$，即 x 有一半可能在表中，则 $T_{\text{avg}}(n)\approx 3n/4$，说明在这种情况下，平均要检查 3/4 张表。如果 $q=0$，即 x 不在表中，则 $T_{\text{avg}}(n)=n$。

（3）最坏情况复杂度分析　显然，$T(n)=\max\limits_{1\leqslant i\leqslant n+1}t(I_i)=n$。此情况发生在 x 是表中的最后一项或 x 不在表中的时候。此时，算法需要检查完整张表。

4. 渐进复杂度

随着时代的发展、技术的进步和科学研究的不断深入，需要使用计算机解决的问题越来越复杂，越来越庞大。所以，对求解这类问题的算法作复杂性分析的意义是非常重大的。然而，为了对这类算法时间复杂度的分析所需要的计算也越来越复杂，越来越庞大。事实上，通常是没有必要精确地计算出算法的时间复杂度的，只要大概地计算出相应的数量级(order)就可以了。因此，引入复杂性渐近性态的定义，使得能在数量级上可以估计一个算法的执行时间，从而简化算法复杂度的分析过程。

设 $T(n)$ 是前面所定义的关于算法 A 的时间复杂度函数。通常，当 n 单调增加且趋于∞时，$T(n)$ 也将单调增加趋于∞。对于 $T(n)$，如果存在一个 $t(n)$，使得当 $n\to\infty$ 时有 $(T(n)-t(n))/T(n)\to\infty$，那么，就说 $t(n)$ 是 $T(n)$ 当 $n\to\infty$ 时的渐近性态，或者称 $t(n)$ 为算法 A 当 $n\to\infty$ 的渐近复杂性而区别于 $T(n)$。也就是，当问题的规模递增 n 时，将复杂度的极限称为渐进复杂度。因为在数学上，$T(n)$ 是 $t(n)$ 当 $n\to\infty$ 时的渐近表达式，直观上，$t(n)$ 是 $T(n)$ 中略去低阶项所保留下的主要的项，所以它无疑会比 $T(n)$ 来得简单方便一些。比如当