高等学校电子信息类专业
“十三五”规划教材

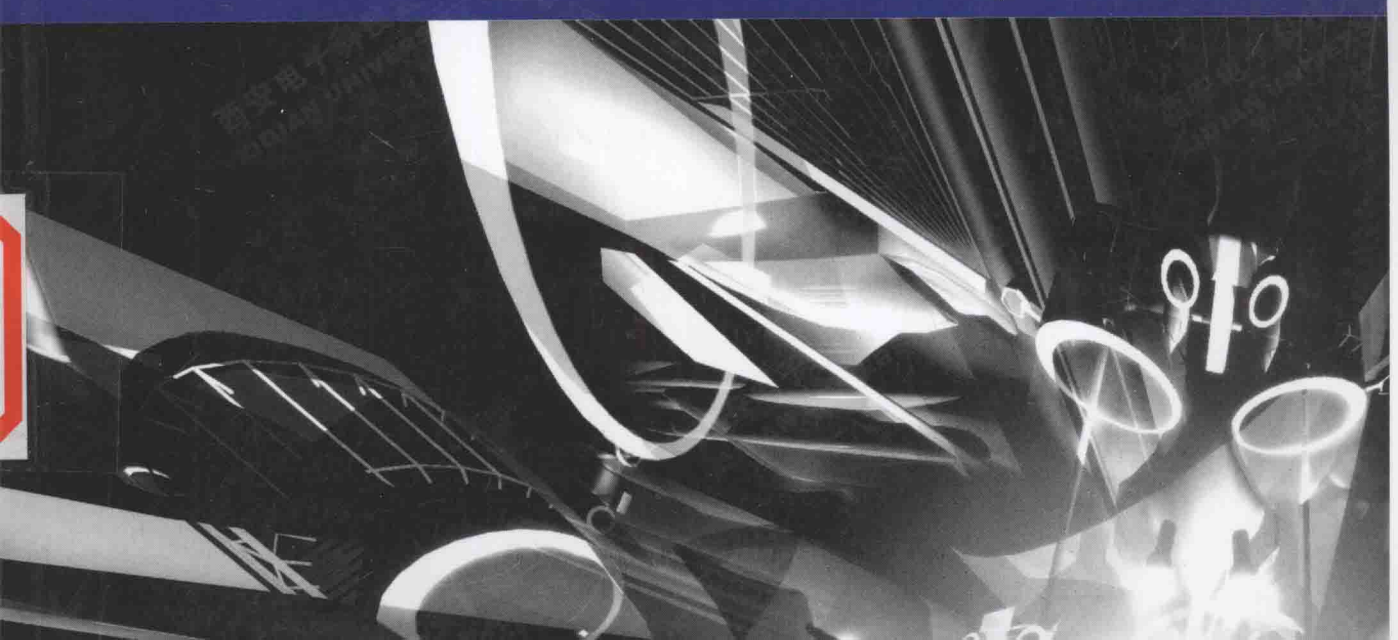
ELECTRONIC
INFORMATION SPECIALTY

算法分析与设计技巧

Algorithm Analysis and Design Techniques

司存瑞　司栋　苏秋萍　艾庆兴　编著
Cunrui Si　Dong Si　Qiuping Su　Qingxing Ai

西安电子科技大学出版社
http://www.xduph.com



高等学校电子信息类专业“十三五”规划教材

算法分析与设计技巧

Algorithm Analysis and Design Techniques

司存瑞　司　栋　苏秋萍　艾庆兴　编　著
Cunrui Si　Dong Si　Qiuping Su　Qingxing Ai

西安电子科技大学出版社

内 容 简 介

本书集作者多年的教学经验及国内外关于算法分析与设计的最新内容于一体。

全书共分5章，第1章介绍了算法的概念与评价，第2章介绍了递归法、分治法、贪心法、搜索法和回溯法等常用算法的概念、基本思想及其应用，第3章对动态规划算法的基本思想与概念、解题方法与步骤及其简单应用与优化等进行了全面深入的研究，第4章着重讨论了搜索算法中的优化技巧，第5章对图上的算法：并查集、生成树、最短路、强连通分量、2－SAT、差分约束、二分图以及网络流进行了全面梳理与分析。为了使学生尽快掌握算法分析与设计技巧，除第1章外，其余各章特意从近年来国际、国内信息学竞赛试题中精选了若干试题作为例题，对这些例题从算法分析、设计技巧到代码实现均给出了完整的解决方案。相信这些内容会给读者带来诸多方便。

本书内容深入浅出，层次清晰，不仅能帮助程序设计者掌握算法分析与设计技巧，更从启迪思维、开发智力的角度引导程序设计者使用计算机来分析问题和解决问题。

本书既可以作为ACM大学生程序设计竞赛及大专院校相关专业的参考教材，同时也可以作为软件开发者和广大工程技术人员的参考书。

图书在版编目(CIP)数据

算法分析与设计技巧/司存瑞等编著. —西安：
西安电子科技大学出版社，2016.1
高等学校电子信息类专业“十三五”规划教材
ISBN 978－7－5606－3900－0

Ⅰ. ① 算… Ⅱ. ① 司… Ⅲ. ① 电子计算机－算法分析－高等学校－教材 ② 电子计算机－算法设计－高等学校－教材 Ⅳ. ① TP301.6

中国版本图书馆CIP数据核字(2015)第292074号

策　　划　云立实
责任编辑　云立实　张驰
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
网　　址　www.xduph.com　　电子邮箱　xdupfxb001@163.com
经　　销　新华书店
印刷单位　陕西华沐印刷科技有限责任公司
版　　次　2016年1月第1版　2016年1月第1次印刷
开　　本　787毫米×1092毫米　1/16　印张 19.5
字　　数　465千字
印　　数　1～3000册
定　　价　35.00元
ISBN 978－7－5606－3900－0/TP
XDUP　4192001－1

前　言

Anany Levitin 在他的名著《算法设计与分析基础》第 1 章绪论中一开篇引用了 David Berlinski 对算法的评价：

科技殿堂里陈列着两颗熠熠生辉的宝石，一颗是微积分，另一颗就是算法。微积分以及在微积分基础上建立起来的数学分析体系成就了现代科学，而算法则成就了现代世界。

算法是当代信息技术的重要基石，同时也是计算科学的一个永恒主题。在计算机科学技术领域内，算法更是处于核心地位。用计算机解决实际问题，除了要求开发者具有扎实的基础知识、掌握计算机的程序设计语言、熟悉数据结构外，更需要算法的强力支持。这是因为，解决实际问题的一般过程遵循下面的流程：

理解问题并建立相应的数学模型

↓

根据数学模型的特点选用适当的数据结构与算法

↓

对算法进行分析与优化

↓

将算法用代码实现并调试正确

在实际中，往往会看到这样的情况：开发者找出了解决问题的算法，但在软件测试(时间与空间方面)时却难以通过。为什么会出现这样的情况呢？其主要原因就是开发者忽视了将算法转变为程序实现这一环节的重要性和必要性。

在解决实际问题的过程中，我们强调算法是首要的，具有战略性的地位。但是一个算法再好，如果不能转化为正确的程序，那么，对解决这一实际问题来说，其效果就是零。这就是算法用程序实现的必要性。与此同时，算法实现的快慢，决定了能否在有限时间和空间内去解决其他问题；算法实现的好坏，决定了算法的表现，这主要体现在对程序的测试过程中。在有很多人想到了正确算法的情况下，用优化了的算法编写高质量程序实现就尤为重要了。这就是算法用程序实现的重要性。

算法与其程序实现的关系，是战略与战术的关系，算法起决定性作用，从根本上决定了算法质量的优劣，但算法的程序实现反过来也影响着算法，并且在一定的条件下，这种影响超过了算法本身的作用。为此，我们组织了从事算法研究多年、具有丰富教学经验的一线教师共同精心编写了本书，试图从算法分析与设计技巧的高度来提高开发者的程序设计技能。

目前，国内外算法分析与设计方面的教科书大都是从理论上加以阐述，代码部分是以伪代码的形式给出，而读者和用户迫切需要了解的是：这种算法是否最优，它用程序如何实现？本书作者从事数据结构、算法分析与设计等课程教学多年，对这些内容本身有比较深刻的理解，对学生们学习该课程的需求和难点所在有着比较清楚的认识，因此，在本书的内容安排和知识点的处理方面我们作了一些尝试，力求体现以下特点：

(1) 对每一种算法的基本思想与概念给予完整的介绍，以方便读者掌握这些算法的适用范围及解决问题的思路和步骤。

(2) 在叙述各种算法内容时，穿插了算法设计和分析技巧，并在程序实现部分给出了完整代码。

(3) 文字简练明了，难点剖析详尽，对重点算法和典型问题的分析均给予注释，以方便读者彻底弄懂、弄通。

(4) 删繁就简，着力突出算法分析和设计的核心内容。事实上，算法分析和设计所涉及的内容还有很多，受篇幅所限，这里略去。

(5) 为了使学生尽快掌握算法分析与设计技巧，除第 1 章外，其余各章特意从近年来国际、国内信息学奥林匹克竞赛试题中精选了若干试题作为例题，对这些例题从算法分析、设计技巧到代码实现均给出了完整的解决方案。相信这些内容会给读者带来更多益处。

本书由司存瑞全面规划，司存瑞、司栋、苏秋萍、艾庆兴共同编写。王袁广、黄希敏等同志参加了部分章节的编写与程序调试，他们为本书的出版付出了辛勤的劳动。

本书的前身是课程讲义，作者在原基础上进行了大量修改。从某种程度上说，本书的出版要归功于使用讲义的读者们，是他们给出了许多很好的想法和一些困难问题的解决思路。尽管作者讲授数据结构、算法分析与设计等课程并从事软件开发多年，但由于水平有限，书中不妥之处在所难免，恳请各位读者批评指正。

作　者

2015 年 6 月

目　录

第1章　算法的概念

自从 1946 年世界上第一台计算机诞生以来，计算机科学与技术的飞速发展和广泛应用远远超出了人们的预料。如今，计算机的应用已经渗透到各个领域，一定程度上改变了人类的活动方式和思维习惯。与此同时，计算机处理的对象也从单纯的数值计算发展到各种不同形式的数据，如字符、表格、声音、图像等。

我们知道，应用计算机处理实际问题时，首先需要很好地分析问题，找出正确、合理的数学模型，据此设计相应的算法；其次一定要分析、估计算法的复杂程度，评价算法的优劣；最后才是编写程序并运行。所以，问题分析、算法设计、算法评价是一个系统工程，也是本书讨论的课题。

1.1　算法的概念和描述

算法，对于计算机专业人士来说，无论从理论还是从实践的角度，都是有必要学习和研究的。这是因为，从实践的角度来看，必须了解计算领域中不同问题的一系列标准算法，此外还要具备设计新的算法和分析其效率的能力；从理论的角度来看，对算法的研究已被公认为是计算机科学的基石。

对于非计算机的相关人士，学习算法的理由也是非常充分的，坦率地说，没有算法，计算机程序将不复存在，更不用说使用它了。而且，随着计算机日益渗透到我们的工作和生活的方方面面，需要学习算法的人也越来越多。

1.1.1　算法的概念

虽然对算法的概念没有一个大家公认的定义，但我们对它的含义还是有基本共识的。

算法是一系列解决问题的清晰指令，也就是对于符合一定规范的输入在有限步骤内求解某一问题所使用的一组定义明确的规则。通俗点说，就是计算机解题的过程。在这个过程中，无论是形成解题思路还是编写程序，都是在实施某种算法。前者是推理实现的算法，后者是操作实现的算法。

这个定义可以用一幅简明的图来说明，如图 1.1 所示。

一个算法应该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，并且每一步都在有穷时间内完成。

② 确定性：算法的每一步骤都必须有确切的定义。

③ 输入：一个算法有零个或多个输入，以刻画运算对象的初始情况。

④ 输出：一个算法有一个或多个输出，以反映对输入数

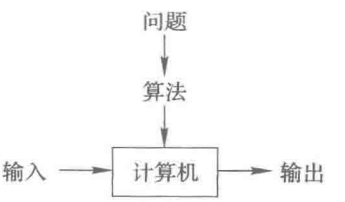


图 1.1　算法的概念

据加工后的结果。没有输出的算法是毫无意义的。

⑤ 可行性：算法应该是可行的，这意味着所有待实现的算法都是能够理解和实现的，并可通过有限次运算完成。

为了阐明算法的概念，本节将以三种方法为例来解决同一个问题：计算两个整数的最大公约数。这些例子会帮助我们阐明以下几项要点：

① 算法的每一个步骤都必须没有歧义，不能有半点含糊。

② 必须认真确定算法所处理的输入的值域。

③ 同一算法可以用几种不同的形式来描述。

④ 同一问题，可能存在几种不同的算法。

⑤ 针对同一问题的算法可能会基于完全不同的解题思路而且解题速度也会有显著不同。

还记得最大公约数的定义吗？将两个不全为 0 的非负整数 m 和 n 的最大公约数记为 gcd(m, n)，代表能够整除(即余数为 0)m 和 n 的最大正整数。亚历山大的欧几里得(公元前 3 世纪)所著的《几何原本》，以系统论述几何学而著称，在其中的一卷里，他简要地描述了一个最大公约数算法。用现代数学的术语来表述，欧几里得算法基于的方法重复应用下列等式，直到 m mod n 等于 0。

$$gcd(m, n)=gcd(n, m \bmod n)(m \bmod n \text{ 表示 m 除以 n 之后的余数})$$

因为 gcd(m, 0)=m，m 最后的取值也就是 m 和 n 的初值的最大公约数。

举例来说，gcd(60, 24)可以这样计算：

$$gcd(60, 24)=gcd(24, 60 \bmod 24)=gcd(24, 12)$$
$$=gcd(12, 24 \bmod 12)=gcd(12, 0)=12$$

下面是该算法的一个更加结构化的描述。

用于计算 gcd(m, n)的欧几里得算法：

第一步：如果 n=0，返回 m 的值作为结果，同时函数结束；否则，进入第二步。

第二步：m 除以 n，将余数赋给 r。

第三步：将 n 的值赋给 m，将 r 的值赋给 n，返回第一步。

我们也可以使用伪代码来描述这个算法：

```
算法 Euclid(m, n)
    //使用欧几里得算法计算 gcd(m, n)
    //输入：两个不全为 0 的非负整数 m, n
    //输出：m, n 的最大公约数
        while n≠0 do
        { r ←m mod n
          m ← n
          n ← r
        }
        return m
```

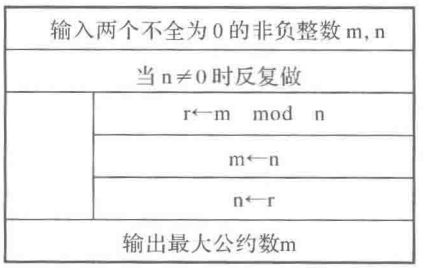


图 1.2 欧几里得算法的流程图

上面的伪代码也可以用流程图来加以描述，如图 1.2 所示。

我们怎么知道欧几里得算法最终一定会结束呢？通过观察，我们发现，每经过一次循环，参加运算的两个算子中的后一个都会变得更小，而且绝对不会变成负数。确实，下一

次循环时，n 的新值是 m mod n，这个值总是比 n 小。所以，第二个算子的值最终会变成 0，此时，这个算法也就停止了。

就像其他许多问题一样，最大公约数问题也有多种算法。让我们看看解这个问题的另外两种方法。第一个方法只基于最大公约数的定义：m 和 n 的最大公约数就是能够同时整除它们的最大正整数。显然，这样一个公约数不会大于两数中的较小者，因此，我们先有：t=min{m，n}。现在可以开始检查 t 是否能够整除 m 和 n：如果能，t 就是最大公约数；如果不能，我们就将 t 减 1，然后继续尝试(我们如何确定该算法最终一定会结束呢?)。

例如，对于 60 和 24 这两个数来说，该算法会先尝试 24，然后是 23，这样一值尝试到 12，算法就结束了。我们给这种算法命名为连续整数检测算法，下面是该算法的具体描述。

用于计算 gcd(m，n)的连续整数检测算法：

第一步：将 min{m，n}的值赋给 t。

第二步：m 除以 t，如果余数为 0，进入第三步；否则，进入第四步。

第三步：n 除以 t，如果余数为 0，返回 t 的值作为结果；否则，进入第四步。

第四步：把 t 的值减 1。返回第二步。

注意：和欧几里得算法不同，按照这个算法的当前形式，当它的一个输入为 0 时，计算出来的结果是错误的。这个例子说明了为什么必须认真、清晰地规定算法输入的值域。

求最大公约数的第三种方法，我们应该在中学时就很熟悉了。

中学里计算 gcd(m，n)的方法：

第一步：找到 m 的所有质因数。

第二步：找到 n 的所有质因数。

第三步：从第一步和第二步求得的质因数分解式中找出所有的公因数(如果 p 是一个公因数，而且在 m 和 n 的质因数分解式分别出现过 pm 和 pn 次，那么应该将 p 重复 min {pm，pn}次)。

第四步：将第三步中找到的公因数相乘，其结果作为给定数 m 和 n 的最大公约数。

这样，对于 60 和 24 这两个数，我们得到：

$$60=2\times2\times3\times5$$

$$24=2\times2\times2\times3$$

$$\gcd(60,24)=2\times2\times3=12$$

我们能够看到，第三种方法比欧几里得算法要复杂得多，也慢得多。撇开低劣的性能不谈，以这种形式表述的中学求解过程还不能称为一个真正意义上的算法。为什么？因为其中求质因数的步骤并没有明确定义。

上面的例子似乎有一些数学味道，尽管应用于计算机程序的算法不见得都涉及数学问题，但我们可以看到，无论是工作还是生活中，算法每天都在帮助我们处理各种事务。算法在当今社会是无处不在的，它是信息时代的引擎，希望这个事实能够使大家下定决心，深入地去学习。

1.1.2　算法的描述

我们一旦设计了一个算法，就需要用一定的方式对其进行详细描述。上一节中，我们已经用文字、伪代码及流程图分别描述了欧几里得算法。这是当今描述算法的三种最常用

的做法。

使用自然语言描述算法显然很有吸引力，但是自然语言固有的不严密性使得要简单清晰地描述算法变得很困难。不过，这也是我们在学习算法的过程中需要努力掌握的一个重要技巧。

伪代码是自然语言和类编程语言组成的混合结构。伪代码往往比自然语言更精确，而且用伪代码描述的算法通常会更简洁。令人惊讶的是，计算机科学家从来没有就伪代码的形式达成过共识，而是让教材的作者去设计他们自己的“方言”。幸运的是，这些方言彼此十分相似，任何熟悉一门现代编程语言的人都完全能够理解。

在计算机应用早期，描述算法的主要工具是流程图。流程图使用一系列相连的几何图形来描述算法，几何图形内部包含对算法步骤的描述。实践证明，除了一些非常简单的算法以外，这种表示方法使用起来非常不便。如今，我们只能在早期的算法教材里找到它的踪影。

本书选择的描述方式力求不给读者带来困难，出于对简单性的偏好，我们忽略了对变量的定义，并使用了缩进来表示 for、if 和 while 语句的作用域。正像大家在前一节里看到的那样，我们将使用箭头“←”表示赋值操作，用双斜线“//”表示注释。

当代计算机技术还不能将自然语言或伪代码形式的算法描述直接“注入”计算机。我们需要把算法变成用特定编程语言编写的程序。尽管这种程序应当属于算法的具体实现，但我们也能将其看作算法的另一种表述方式。

1.2 算法的时间复杂度和空间复杂度

1.2.1 算法的评价

很多时候我们往往忽略或者轻视了分析算法的复杂程度，评价算法的优劣，但这很重要，也很必要。事实上，对同一个问题的不同算法、不同程序，有些时候，执行所花费的时间相差很大，有的程序的运行时间甚至无法接受，比如 30 分钟、40 分钟还没有完整的结果。这正是我们本节要讨论的问题——算法的复杂度，即估计、评价算法运行时所需要花费的时间及空间。

算法的时间复杂度和空间复杂度合称为算法的复杂度。

那么，算法一旦确定，如何计算它的复杂度，衡量其优劣呢？通常从下面几个方面来考虑：

(1) 正确性：也称有效性，是指算法能满足具体问题的要求。即对任何合法的输入，算法都会得出正确的结果。确认正确性的根本方法是进行形式化的证明。但对一些较复杂的问题，这是一件相当困难的事。许多计算机科学工作者正致力于这方面的研究，目前尚处于初级阶段。因此，实际中常常用测试的方法验证算法的正确性。测试是指用精心选定的输入(测试数据)去运行算法，检查其结果是否正确。但正如著名的计算机科学家 E. Dijkstra 所说的那样，“测试只能指出有错误，而不能指出不存在错误”。

(2) 可读性：指算法被理解的难易程度。人们常把算法的可读性放在比较重要的位置，主要是因为晦涩难懂的算法不易交流和推广使用，也难以修改、扩展与调试，而且可能隐藏较多的错误。可读性实质上强调的是越简单的东西越美。

(3) 健壮性：对非法输入的抵抗能力。它强调的是：如果输入非法数据，算法应能加以识别并做出处理，而不是产生误操作或陷入瘫痪。

(4) 时间复杂度与空间复杂度。时间复杂度是算法的计算复杂性的时间度量，粗略地讲，就是该算法的运行时间。同一个问题，不同的算法可能有不同的时间复杂度。问题规模较大时，时间复杂度就变得十分重要。尽管计算机的运行速度提高很快，但这种提高无法满足问题规模加大带来的速度要求。所以追求高速算法仍然是件重要的事情。空间复杂度是指算法运行所需的存储空间的多少。相比起来，人们一般会更多地关注算法的时间复杂度，但这并不是因为计算机的存储空间是海量的，而是由人们面临的问题的本质决定的。时间复杂度与空间复杂度往往是一对矛盾。常常可以用空间换取速度，反之亦然。

1.2.2　算法的时间复杂度

给定一个算法，如何确定该算法的时间复杂度呢？我们可以采用事后统计的办法得出执行算法所用的具体时间。因为很多计算机都有这种计时的功能，甚至可以获取精确到毫秒级的统计数据。但这种办法有两个缺陷：第一，必须先运行才能分辨出算法的好坏；第二，所得的时间的统计量过分依赖于计算机的硬件、软件等环境因素，这些因素容易掩盖算法本身的优劣，因为一个笨拙的算法在先进的大型机上运行可能比一个同功能的优化算法在微型机上运行更省时间。

因此，人们常常对算法进行事前的时间估算。可利用语句的“频度”和算法的渐进时间复杂度这两个与软、硬件无关的度量来讨论算法执行的时间消耗。

设 n 称为问题的规模，当 n 不断变化时，算法中的基本语句执行的频度(我们且称为时间频度)T(n)也会不断变化。一般情况下，算法中基本操作重复执行的次数是问题规模 n 的某个函数，用 T(n)表示，若有某个辅助函数 f(n)，使得当 n 趋近于无穷大时，T(n)/f(n)的极限值为不等于零的常数，则称 f(n)是 T(n)的同数量级函数。记作 T(n)=O(f(n))，称 O(f(n))为算法的渐进时间复杂度，简称时间复杂度。

时间频度不同，但时间复杂度可能相同。如：$T(n)=n^2+3n+4$ 与 $T(n)=4n^2+2n+1$ 它们的频度不同，但时间复杂度相同，都为 $O(n^2)$。

用两个算法 A1 和 A2 来求解同一问题，时间复杂度分别是 $T1(n)=100n^2$，$T2(n)=5n^3$。

(1) 当输入量 $n<20$ 时，有 $T1(n)>T2(n)$，后者花费的时间较少。

(2) 随着问题规模 n 的增大，两个算法的时间开销之比 $5n^3/100n^2=n/20$ 亦随着增大。即当问题规模较大时，算法 A1 比算法 A2 要有效得多。它们的渐近时间复杂度 $O(n^2)$和 $O(n^3)$从宏观上评价了这两个算法在时间方面的质量。在算法分析时，往往对算法的时间复杂度和渐近时间复杂度不予区分，而经常是将渐近时间复杂度 T(n)=O(f(n))简称为时间复杂度，其中的 f(n)一般是算法中频度最大的语句的频度。

研究算法的时间复杂度，通常考虑的是算法在最坏情况下的时间复杂度，称为最坏时间复杂度。一般不特别说明，讨论的时间复杂度均是最坏情况下的时间复杂度。这样做的原因是：最坏情况下的时间复杂度 T(n)=O(n)，是算法在任何输入实例上运行时间的上界，即它表示对于任何输入实例，该算法的运行时间不可能大于 O(n)。

另一个衡量算法复杂度的指标是平均时间复杂度，它是指所有可能的输入实例均以等概率出现的情况下，算法的期望运行时间。

下面是一个 n×n 矩阵 A，求得 $B=A^2$ 的算法。

```
    void MTXMCT(A, n, Var B);
    //A 是原始矩阵，B 存放结果
    {
1       for (i=1; i<=n; n++)
2       for (j=1; j<=n; n++)
3       {  b[i][j]=0;
4          for (k=1; k<=n; n++)
5          b[i][j]=b[i][j]+a[i][k]*a[k][j]
        }
    }
```

上述算法中，语句 3 在二重循环之内，重复执行的次数为 n^2。语句 5 在三重循环之内，重复执行的次数为 n^3。假设语句 3 执行一次的时间是 t_1，语句 5 执行一次的时间是 t_2，若只考虑算法中这两个主要赋值语句的执行时间，而忽略步进语句中其他成分，如步长加 1、终值判断、控制转移等所需时间，则可以认为此算法耗用时间近似为

$$T(n)=t_1n^2+t_2n^3$$

式中，矩阵的阶 n 表示问题的规模，当 n 很大时，显然有

$$\lim_{n\to\infty}\frac{T(n)}{n^3}=\lim_{n\to\infty}\frac{t_1n^2+t_2n^3}{n^3}=t_2$$

这表明，当 n 充分大时，$T(n)/n^3$ 为一个常数，即 $T(n)$ 和 n^3 是同阶的，可记作 $T(n)=O(n^3)$。

下面分析几个程序中标有 # 语句的频度和该程序段的时间复杂度。

(1) 如果算法的执行时间不随着问题规模 n 的增加而增长，即使算法中有若干条语句，其执行时间也只是一个常数。我们认为这类算法的时间复杂度是 O(1)。

```
for (i=0; i<1000; i++)
#1 {if (i%5==0)printf("\n");
#2 printf("i=%5d", i);
}
```

显然，程序中的语句总共循环执行了 1000 次，但这段程序的运行是和 n 无关的，所以该程序段的时间复杂度为 T(n)=O(1)。

(2) 频度统计法。频度统计法指以程序中语句执行次数的多少作为算法时间度量分析的一种方法。通常情况下，算法的时间效率主要取决于程序中包含的语句条数和采用的控制结构这两者的综合效果。因此，最原始且最可靠的方法是求出所有主要语句的频度f(n)，然后求所有频度之和。当有若干个循环语句时，算法的时间复杂度是由嵌套层数最多的循环语句中最内层语句的频度决定的。

```
for(i=1; i<=n-1; i++)
{ #1 y=y+1;                    //基本语句
    for (j=1; j<=2*n; j++)     //控制语句
    #2 x=x+1;
}
```

这个由两个 for 语句构成的程序段，外循环的重复执行次数是 n－1 次，即语句＃1 的频度为 n－1，内循环的单趟重复执行次数是 2×n 次，则语句＃2 的频度为(n－1)(2n)＝$2n^2-2n$，所以

$$T(n)=O(\sum f(n))=(n-1)+(2n^2-n)=O(n^2)$$

(3) 频度估算法。先找出对于所求解的问题来说是共同的原操作，并求出原操作的语句频度 f(n)，然后直接以 f(n)衡量 T(n)。在使用频度估算法时应注意一个显著的标志，就是原操作往往是最内层循环的循环体，并且完成该操作所需的时间与操作数的具体取值无关。这种方法比较适用于带有多重循环的程序。

例如，对于有多个串行的循环语句的程序来说，算法的时间复杂度是由嵌套层次最多的循环语句的里层语句决定的。

```
x=1;
y=1;
for (k=1; k<=n; n++)
    #1 x=x+1;
for (i=1; i<=n; n++)
for (j=1; j<=n; n++)
    #2 y=y+1;
```

语句＃1 的频度为 n，语句＃2 的频度为 n^2，显然，此程序段的时间复杂度为 $T(n)=O(n^2)$。

对于一些复杂的算法，可以将算法分解成容易估算的几个部分，利用频度估算法分别求出这几部分的时间复杂度，然后利用求和的原则即可得到整个算法的时间复杂度。

频度估算法的结果较精确，方法简单且易掌握。

(4) 算法的时间复杂度不仅仅依赖于问题的规模，还与输入实例的初始状态有关。例如：

```
i=1;
while (i<n)and (x≠a[i])
     # i=i+1;
if a[i]=x then return(i)
```

此程序段中语句＃的频度不仅是 n 的函数，而且与 x 及数组 a 中各元素的具体值有关，在这种情况下，通常按最坏的情况考虑。由于 while 循环执行的最大次数为 n－1，则语句＃的频度的最大值为 f(n)＝n－1，则认为此程序段的时间复杂度为 T(n)＝O(n)。

(5) 递归算法的频度不容易估算，须由递推公式计算求得。其基本方法是：方程右边较小的项根据定义被依次替代，如此反复扩展，直到得到一个没有递归式的完整数列，从而将复杂的递归问题转化为了新的求和问题。

以有名的 Hanoi 塔问题为例，其解是一个递归形式的算法：

```
void hanoi(int n, char A, char B, char C)
{
    if(n==1)move(n, A, C);
    else
    { hanoi(n-1, A, C, B);
      move(n, A, C);
      hanoi(n-1, B, A, C);
    }
}
```

我们不打算分析这个算法的来龙去脉，只关心算法的时间复杂度。显见问题的规模是n，move(n，A，C)语句的频度为1，若整个算法的频度为f(n)，则递归调用hanoi(n－1，A，C，B)和hanoi(n－1，B，A，C)语句的频度应为f(n－1)，于是有

$$f(n)=f(n-1)+1+f(n-1)$$

即

$$f(n)=2f(n-1)+1,$$

进行递推，有

$$\begin{aligned}f(n)&=2(2f(n-2)+1)+1\\&=4f(n-2)+3\\&=4(2f(n-3)+1)+3\\&=8f(n-3)+7\\&\cdots\\&=2kf(n-k)+2k-1\end{aligned}$$

当n=k+1时，f(n)=2n－1f(1)+2n－1－1。

f(1)是n=1时算法的频度，它只有move(n，A，C)语句，其值为1。最后得到f(n)=2n－1。时间复杂度T(n)=O(2n)。

总之，频度和时间复杂度虽不能精确地确定一个算法或程序的执行时间，但可以让人们知道随着问题的规模增大，算法耗用时间的增长趋势。由于讨论算法的好坏不是针对某个特定大小的问题(这样做本身没有意义)，因此，时间复杂度对算法来说是一个较恰当的量度。

较常见的时间复杂度有O(1)(常量型)、O(n)、$O(n^2)$、…、$O(n^k)$(多项式型)、O(lbn)、O(nlbn)(对数型)和O(n!)(阶乘型)。显然，时间复杂度为$O(2^n)$或$O(e^n)$的指数型算法的效率极低，在n较大时无法实用。如图1.3所示表明了各种时间复杂度的增长率。

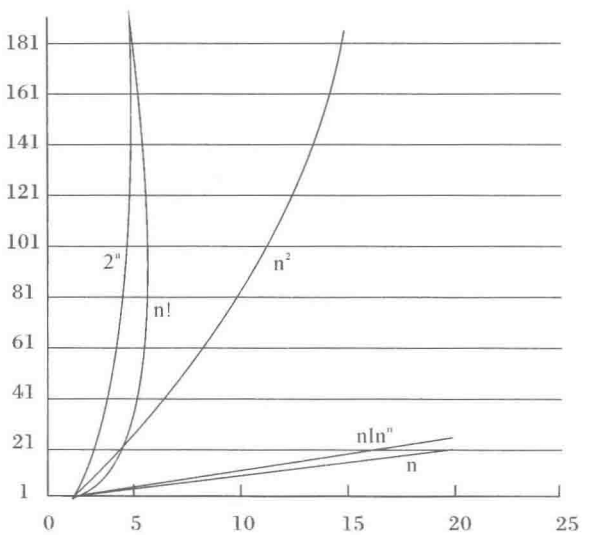


图1.3　各种时间复杂度的增长率

表 1.1 是我们所熟悉的各种排序算法的复杂度，读者可以分析思考。

表 1.1　常用排序算法的时间复杂度和空间复杂度

排序法	最差时间分析	平均时间复杂度	稳定度	空间复杂度
冒泡排序	$O(n^2)$	$O(n^2)$	稳定	O(1)
快速排序	$O(n^2)$	O(n * lbn)	不稳定	O(nlbn)
选择排序	$O(n^2)$	$O(n^2)$	稳定	O(1)
二叉树排序	$O(n^2)$	O(n * lbn)	不稳定	O(n)
插入排序	$O(n^2)$	$O(n^2)$	稳定	O(1)
堆排序	O(n * lbn)	O(n * lbn)	不稳定	O(1)
希尔排序	$O(n^2)$	$O(n^2)$	不稳定	O(1)

实践中我们可以把事前估算和事后统计两种办法结合起来使用，例如，若上机运行一个 10×10 矩阵模型，执行时间为 12 ms，则由算法的时间复杂度 $T(n)=O(n^3)$ 可估算一个 31×31 矩阵自乘的时间大约为 $(31/10)^3\times 12$ ms≈358 ms。这种办法很有实用价值，它用小模型可推得大尺寸问题的耗用机时。

根据算法时间复杂度的定义，容易证明它有如下性质：

(1) O(f)+O(g)=O(max(f, g))。

(2) O(f)+O(g)=O(f+g)。

(3) O(f)×O(g)=O(f×g)。

(4) 如果 g(N)=O(f(N))，则 O(f)+O(g)=O(f)。

(5) O(Cf(N))=O(f(N))，其中 C 是一个正常数。

有兴趣的读者可以探讨并具体证明。

对于多数问题，其时间复杂度是可以事先估算的。例如，读者可能熟悉的著名的 Fibonacci数列：

0，1，1，2，3，5，8，13，21，34，55……

其数列中每个数都是其两个直接前项的和，下面的式子给出 Fibonacci 数列 F_n 的生成规律：

$$F_n=\begin{cases}F_{n-1}+F_{n-2}, & \text{若 } n>1\\ 1, & \text{若 } n=1\\ 0, & \text{若 } n=0\end{cases}$$

欲求得 Fibonacci 数列的第 n 项，第一个解决问题的方法自然就是直接按照 Fibonacci 数列的定义写出相应的程序代码：

```
void fib1(int n)
{
    if(n==0)return 0;
    if(n==1)return 1;
    return fib1(n-1)+fib1(n-2);
}
```

对此解决方法，我们必须考虑如下三个问题：

问题 1：算法正确吗？

问题 2：它将耗费多少时间，时间复杂度是一个什么样的函数？

问题 3：这种算法还能改进吗？

考虑问题 1 实际上没有意义，因为该程序是严格按照 Fibonacci 数列的递归定义给出的。

对于问题 2，我们令函数 f(n)表示计算 fib1(n)所需要的基本操作次数，显然，当 $n\leqslant 1$ 时，仅执行了几次操作，程序很快就结束了，从而有：当 $n\leqslant 1$ 时，$f(n)\leqslant 2$。

但当 $n>2$ 且逐渐增大时，fib1 将被递归调用两次，运行时间分别是 fib1(n−1)和 fib1(n−2)，另外还有三次(检查 n 的值和一个最终的加法操作)基本操作，从而有：当$n>1$时，$f(n)=f(n-1)+f(n-2)+3$。

将上式与 F_n 的递推关系式比较，我们发现 $f(n)\geqslant F_n$。

也就是说，fib1 运行时间增长的速度与 Fibonacci 数增长的速度一样快！由于 f(n)关于 n 是指数级的，这就意味着除了 n 取一些很小的值外，该算法将很慢，因此并不实用。

让我们用实际数据来说明 f(n)关于 n 是指数级时间的算法 fib1 并不实用的问题出在哪里。如果要计算 F_{200}，算法就要执行 f(200)次操作，其中 $f_{(200)}\geqslant F_{200}\geqslant 2^{138}$。其实际运行时间当然要依赖于所使用的计算机，假定你所使用的计算机的时钟频率是每秒 40 万亿次基本操作，f(200)至少要耗时 2^{92} 秒，这就意味着大约需要计算 157 019 284 536 451 074 940 年，这简直让人不可思议！

可见，这个简单的递归算法虽然正确，但却毫无效率，令人失望。那么，fib1 算法能改进吗？

首先，我们分析 fib1 算法为什么如此之慢呢？如图 1.4 所示提示了由一个单独的 fib1(n)调用过程触发的一系列递归操作。请注意很多计算步骤都是重复的！

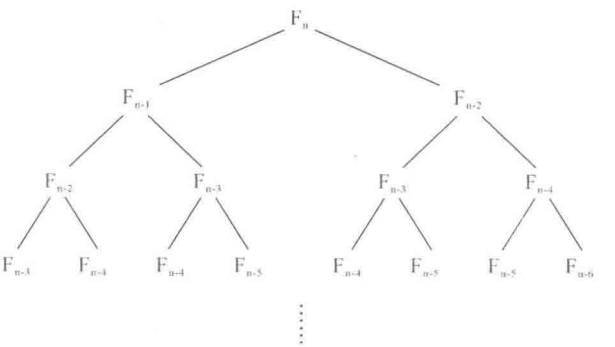


图 1.4　fib1(n)递归调用的扩张过程

正是由于重复计算才导致了 fib1 算法如此之慢。一种更合理的算法是随时存储中间计算结果——F_0，F_1，…，F_{n-1} 的值。

```
void fib2(int n)
{
    if(n==0)return 0;
    f[0]=0; f[1]=1;
```