
计算机算法
设计、分析与实现
Computer Algorithms
Design, Analysis and Implementation
王晓云　陈业纲/著
科学出版社

# 计算机算法设计、分析与实现

王晓云　陈业纲　著

科学出版社
北京

## 内 容 简 介

算法设计、分析与实现是计算机软件开发人员应掌握的基本要素，在大型程序开发中越来越受到重视。本书将典型的经典问题和算法设计技术巧妙地进行结合，系统地论述算法设计技术及其在经典问题中的应用。全书共14章，第1章介绍算法的基本概念和算法分析相关的数学问题，第2～13章分别介绍递归的应用、迭代算法、常见排序算法、动态规划法、回溯法、贪心算法、分治算法、概率算法、近似算法、分支限界法、遗传算法、蚁群算法等算法设计技术，第14章介绍查找。书中所有算法均在VC6.0环境下调试通过，并截图显示其运行过程。

本书内容丰富，深入浅出，图例丰富，可作为计算机专业本科高年级学生和研究生学习算法的教材，也可供工程技术人员、软件设计师和自学者参考。

**图书在版编目(CIP)数据**

计算机算法设计、分析与实现 / 王晓云，陈业纲著. —北京：科学出版社，2012

ISBN 978-7-03-035142-5

Ⅰ.①计… Ⅱ.①王…②陈… 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(2012)第161916号

责任编辑：童安齐 隽青龙 / 责任校对：耿 耘

责任印制：吕春珉 / 封面设计：耕者设计工作室

科学出版社 出版

北京东黄城根北街16号

邮政编码：100717

http://www.sciencep.com

双青印刷厂 印刷

科学出版社发行 各地新华书店经销

*

2012年7月第 一 版 开本：B5（720×1000）

2012年7月第一次印刷 印张：23 1/4

字数：455 000

**定价：68.00元**

（如有印装质量问题，我社负责调换）

销售部电话 010-62134988 编辑部电话 010-62135763 转 8228

# 前　言

以最低的成本、最快的速度、最优的质量开发软件是软件工程师追求的目标，而要设计这种软件，不但要遵循软件工程的原则，而且还需要合理的数据组织和高效、简洁的算法。在软件设计的各个领域，算法都扮演着重要角色，被公认为是计算机科学的灵魂。没有算法，计算机程序将不复存在。学会读懂算法、设计算法、实现算法是软件开发中的最基本要求，而发明算法并证明该算法的正确性则是软件设计最高境界。

在理解算法的基础上最终实现算法是本书所追求的目标，通过对计算机算法系统的学习与研究，理解和掌握算法设计的主要方法，培养对算法的计算复杂性进行正确分析的能力，从而为独立设计算法并对算法的复杂性分析奠定坚实的基础。因此，无论是否涉及计算机，特定的算法设计技术都可以看成是问题求解的有效策略。

本书根据作者多年的软件开发所取得的实践经验撰写而成。书中将典型的经典问题和算法设计技术进行了巧妙的结合，系统地论述了常见的算法及其在经典问题中的应用。通过同一问题用不同算法实现并进行比较，使读者更容易体会到算法设计技术的思想，达到融会贯通的效果。

本书有配套的源程序代码，所有代码均在实际工程中得到很好的运用，软件开发人员可将书中代码进行直接或简单修改后用于工程实际；书中所有算法均在VC6.0环境下调试通过，并截图显示其运行过程，可以让算法学习者由理解算法到亲自实现算法的能力得到迅速提高。

在撰写本书的过程中，得到相关软件开发人员和同行的大力支持，他们提出了许多宝贵的意见，同时参考了其他院校的一些算法实验和相关文献，在此特向关心和支持本书撰写的各方人士表示衷心的感谢。

由于作者水平有限，书中难免存在不足之处，敬请读者指正。为方便读者的学习，本书有配套的源程序代码(配套光盘)，可直接与 wxyc93@163.com 联系获取。

# 目　录

# 第 1 章　与算法相关的数学问题

凡编写过程序的人，也许都有这样一种体会，学会编程容易，但是要想编出好程序难。计算机科学是一种创造性思维活动，其运用必须面向设计。虽然人们对算法一词非常熟悉，可到目前为止，对于算法尚没有统一而精确的定义。有人说：算法就是一组有穷的规则，它们规定了解决某一特定类型问题的一系列运算。而权威的描述为：算法是任何定义好了的计算程式，它取某些值或值的集合作为输入，并产生某些值或值的集合作为输出。因此，算法是将输入转化为输出的一系列计算步骤。人们也可以把算法看成解特定的计算问题的工具。一个算法必须具备的性质如下：

(1) 必须是正确的，即对于任意的一组输入，包括合理的输入与不合理的输入，总能得到预期的输出。如果一个算法只是对合理的输入才能得到预期的输出，而在异常情况下却无法预料输出的结果，那么它就不是正确的。

(2) 必须是由一系列具体步骤组成的，并且每一步都能够被计算机所理解和执行，而不是抽象和模糊的概念。

(3) 每个步骤都有确定的执行顺序，即上一步在哪里，下一步是什么，都必须明确，无二义性。

(4) 无论算法有多么复杂，都必须在有限步之后结束并终止运行，即算法的步骤必须是有限的。在任何情况下，算法都不能陷入死循环中。

同时每个算法都有如下特点：

(1) 有穷性。一个算法应包含有限的操作步骤，而不能是无限的。事实上，“有穷性”往往是指“在合理的范围之内”。

(2) 确定性。算法中的每一个步骤都应当是确定的，而不应当是含糊的、模棱两可的。算法中的每一个步骤应当不致被解释成不同的含义，而应是十分明确的。

(3) 有零个或多个输入。所谓输入是指在执行算法时需要从外界取得必要的信息。

(4) 有一个或多个输出。算法的目的是为了求解，没有输出的算法是没有意义的。

(5) 有效性。算法中的每一个步骤都应当能有效执行，并得到确定的结果。

程序是算法的计算机高级语言描述，算法是程序的灵魂。一般来说，一个算法要经过设计、确认、分析、编码、检查、调试、维护等阶段。据此，算法的学习可以分为设计算法、表示算法、确认算法、分析算法、测试算法。求解同一计算问题可能有

许多不同的算法，那么究竟如何来评价这些算法的好坏以便从中选出较好的算法呢？选用的算法首先应该是“正确”的。此外，还应考虑如下三点：

(1) 执行算法所耗费的时间。

(2) 执行算法所耗费的存储空间，其中主要考虑辅助存储空间。

(3) 算法应易于理解、易于编码、易于调试，等等。

一个占存储空间小、运行时间短而其他性能也好的算法是很难做到的。因为上述要求有时相互抵触：要节约算法的执行时间往往要以牺牲更多的空间为代价；而为了节省空间可能要耗费更多的计算时间。因此，我们只能根据具体情况有所侧重：若该程序使用次数较少，则力求算法简明易懂；对于反复多次使用的程序，应尽可能选用快速的算法；若待解决的问题数据量极大，机器的存储空间较小，则相应算法主要考虑如何节省算法的运行时间。

一个算法所耗费的时间等于算法中每条语句的执行时间之和。每条语句的执行时间等于语句的执行次数[即频度(frequency count)]乘以语句执行一次所需时间。

算法转换为程序后，每条语句执行一次所需的时间取决于机器的指令性能、速度以及编译所产生的代码质量等因素。

若要独立于机器的软、硬件系统来分析算法的时间耗费，则设每条语句执行一次所需的时间均是单位时间，一个算法的时间耗费就是该算法中所有语句的频度之和。

## 1.1 复杂性分析初步

程序性能是指运行一个程序所需的内存大小和时间多少，所以程序的性能一般是指程序的空间复杂性和时间复杂性。性能评估主要包含性能分析与性能测量。前者采用分析的方法；后者采用实验的方法。

首先要考虑空间复杂性，在多用户系统中运行时，需指明分配给该程序的内存大小；想预先知道计算机系统是否有足够的内存来运行该程序；一个问题可能有若干个不同的内存需求解决方案，从中择取合适的空间复杂性来估计要解决的问题最大规模。

其次，我们还要考虑时间复杂性，因为某些计算机用户需要提供程序运行时间的上限；所开发的程序需要提供一个满意的实时反应。

对于一个问题有多种解决可选的方案，于是方案的选取要基于这些方案之间的性能差异。对于各种方案的时间及空间的复杂性，最好采取加权的方式进行评价。但是随着计算机技术的迅速发展，对空间的要求往往不如对时间的要求那样强烈，因此主要对时间复杂性进行分析。

### 1.1.1　空间复杂度

程序所需要的空间包括指令空间、数据空间和环境栈空间。

1. 指令空间

程序所需的指令空间(用来存储经过编译之后的程序指令)的大小取决于如下因素:把程序编译成机器代码的编译器;编译时实际采用的编译器选项;目标计算机。所使用的编译器不同,则产生的机器代码的长度就会有差异;有些编译器带有选项,如优化模式、覆盖模式等。所取的选项不同,产生机器代码也会不同。此外,目标计算机的配置也会影响代码的规模。一般情况下,指令空间对于所解决的特定问题不够敏感。

2. 数据空间

数据空间用来存储所有常量和变量的值,分成两个部分:①存储常量和基本变量;②存储复合变量。前者所需的空间取决于所使用的计算机和编译器,以及变量与常量的数目,实际上就是计算所需内存的字节数,而每个字节所占的位数依赖于具体的机器环境;后者包括数据结构所需的空间及动态分配的空间。结构变量所占空间等于各个成员所占空间的累加;数组变量所占空间等于数组大小乘以单个数组元素所占的空间,例如:int a[100],所需空间为 100×4=400。

3. 环境栈空间

环境栈空间保存函数调用返回时恢复运行所需要的信息。当一个函数被调用时,下面数据将被保存在环境栈中。

(1) 返回地址。

(2) 所有局部变量的值、递归函数的传值形式参数的值。

(3) 所有引用参数以及常量引用参数的定义。

在分析空间复杂性时,实例特征的概念非常重要。所谓实例特征是指决定问题规模的那些因素。如输入和输出的数量或相关数的大小,如对 $n$ 个元素进行排序、$n\times n$ 矩阵的加法等,都可以 $n$ 作为实例特征,而两个 $m\times n$ 矩阵的加法应该以 $n$ 和 $m$ 两个数作为实例特征。

指令空间的大小对于所解决的待定问题不够敏感。常量及简单变量所需要的数据空间也独立于所解决的问题,除非相关数的大小对于所选定的数据类型来说实在太大。这时,要么改变数据类型,要么使用多精度算法重写该程序,然后再对新程序进行分析。

复合变量及动态分配所需要的空间同样独立于问题的规模,而环境栈通常独

立于实例的特征，除非正在使用递归函数。在使用递归函数时，实例特征通常会影响环境栈所需要的空间数量。

递归函数所需要的栈空间主要依赖于局部变量及形式参数所需要的空间。除此以外，该空间还依赖于递归的深度（即嵌套递归调用的最大层次）。

综上所述，一个程序所需要的空间可分为两个部分：①固定部分，它独立于实例的特征，主要包括指令空间、基本变量以及定长复合变量所占用的空间、常量所占用的空间；②可变部分，主要包括复合变量所需的空间（其大小依赖于所解决的具体问题）、动态分配的空间（依赖于实例的特征）、递归栈所需的空间（依赖于实例的特征）。

令 $S(P)$ 表示程序 $P$ 需要的空间，则有

$$S(P) = c + SP(\text{实例特征})$$

其中，$c$ 表示固定部分所需要的空间，是一个常数；$SP$ 表示可变部分所需要的空间。在分析程序的空间复杂性时，一般着重于估算 $SP$（实例特征）。实例特征的选择一般受到相关数的数量以及程序输入和输出规模的制约。

另外一个精确的分析还应当包括编译期间所产生的临时变量所需的空间，这种空间与编译器直接关联，在递归程序中除了依赖于递归函数外，还依赖于实例特征，但是在考虑空间复杂性时，其一般都被忽略。

### 1.1.2 时间复杂度

一个算法执行所耗费的时间，从理论上是不能算出来的，必须上机运行测试才能知道，但我们不可能也没有必要对每个算法都上机测试，只需知道哪个算法花费的时间多，哪个算法花费的时间少就可以了。一个算法花费的时间与算法中语句的执行次数成正比例，即哪个算法中语句执行次数多，它花费的时间就多。一个算法中的语句执行次数称为语句频度或时间频度，记为 $T(n)$，其中 $n$ 称为问题的规模，当 $n$ 不断变化时，时间频度 $T(n)$ 也会不断变化。但有时我们想知道它变化时呈现什么规律。为此，我们引入时间复杂度概念。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。

在各种不同算法中，若算法中语句执行次数为一个常数，则时间复杂度为 $O(1)$，另外，在时间频度不相同时，时间复杂度有可能相同，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 $O(n^2)$。按数量级递增排列，常见的时间复杂度有：常数阶 $O(1)$ < 对数阶 $O(\log_2 n)$ < 线性阶 $O(n)$ < 线性对数阶 $O(n\log_2 n)$ < 平方阶 $O(n^2)$ < 立方阶 $O(n^3)$，…，< $k$ 次方阶

$O(n^k)$<指数阶 $O(2^n)$。随着问题规模 $n$ 的不断增大，上述时间复杂度不断增大，算法的执行效率就越低。

若要比较不同算法的时间效率，首先要确定一个度量标准，最直接的办法就是将算法转化为程序在计算机上运行，并通过计算机内部的计时功能获得精确的时间，然后进行比较。但该方法受计算机的硬件、软件等因素的影响，会掩盖算法本身的优劣，所以一般采用事先分析估算的方法，即撇开计算机软硬件等因素，只考虑问题的规模（一般用自然数 $n$ 表示），认为一个特定的算法的时间复杂度，只与采取问题的规模，或者说它是问题的规模的函数有关。为了方便比较，通常的做法是从算法中选取一种对于所研究的问题（或算法模型）来说是基本运算的操作，以其重复执行的次数作为评价算法时间复杂度的标准。该基本操作多数情况下是由算法最深层的循环体语句表示的，基本操作的执行次数实际上就是相应语句的执行次数，一般 $T(n)=O(f(n))$，其中 $O(1)<O(\log_2 n)<O(n)<O(n\log_2 n)<O(n^2)<O(n^3)<O(2^n)$，通常要选择时间复杂度量级低的算法。

## 1.2　复杂性的计量

算法的复杂性是算法运行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（即存储器）资源的量称为空间复杂性。这个量应该集中反映算法中所采用方法的效率，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 $A$ 来表示算法要解问题的规模、算法的输入和算法本身，用 $C$ 表示算法的复杂性，那么应该有

$$C=F(N,I,A)$$

其中，$F(N,I,A)$ 是 $N,I,A$ 的一个确定的三元函数。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，即

$$T=T(N,I,A)\qquad 和\qquad S=S(N,I,A)$$

通常简写为

$$T=T(N,I)\qquad 和\qquad S=S(N,I)$$

由于时间复杂性和空间复杂性概念类同、计算方法相似，且空间复杂性分析相对简单些，将主要讨论时间复杂性。

$T(N,I)$ 是算法在一台抽象的计算机上运行所需的时间。设此抽象的计算机所提供的元运算有 $k$ 种，它们分别记为 $O_1,O_2,\cdots,O_k$；再设这些元运算每执行一次所需要的时间分别为 $t_1,t_2,\cdots,t_k$。对于给定的算法 $A$，设经过统计，用到元运算 $O_i$ 的次数为 $e_i(i=1,2,\cdots,k)$，很明显，对于每一个 $i$，$1\leqslant i\leqslant k$，$e_i$ 是 $N$ 和 $I$ 的函数，即 $e_i=e_i(N,I)$，于是有

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

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

但是，我们不可能对规模 $N$ 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 $T(N,I)$ 的表达式还得进一步简化。实际中只考虑最坏情况、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 $T_{\text{avg}}(N)$。

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

$$T_{\min}(N)=\max_{I\in D_N}T(N,I)=\max_{I\in D_N}\sum_{i=1}^{k}t_ie_i(N,I)=\sum_{i=1}^{k}t_ie_i(N,\tilde{I})=T(N,\tilde{I})$$

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

其中，$D_N$是规模为 $N$ 的合法输入的集合；$I^*$ 是 $D_N$ 中一个使 $T(N,I^*)$达到 $T_{\max}(N)$的合法输入；$t_i$ 是$D_N$中一个使 $T(N)$到 $T_{\min}(N)$的合法输入；$P(I)$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各从某一个角度来反映算法的效率，各有各的用处，也各有各的局限性。但实践表明，可操作性最好的且最有实际价值的是最坏情况下的时间复杂性。

考虑到分析算法的复杂性的目的在于比较求解同一问题的两个不同算法的效率，而当要比较的两个算法的渐近复杂性的阶不相同时，只要能确定出各自的阶，就可以判定哪一个算法的效率高。换句话说，这时的渐近复杂性分析只要关心 $T'(N)$ 的阶就够了，不必关心包含在 $T'(N)$ 中的常数因子。所以，我们常常又对 $T'(N)$ 的分析进一步简化，即假设算法中用到的所有不同的元运算各执行一次，所需要的时间都是一个单位时间。

综上所述，我们已经给出简化算法复杂性分析的方法，即只要考察当问题的规模充分大时，算法复杂性在渐近意义下的阶。与此简化的复杂性分析方法相适应，特引入五个渐近意义下的记号，即 $\Theta$、$O$、$\Omega$、$o$、$\omega$。

用来表示算法的渐近时间的记号是用定义在自然数 $\boldsymbol{N}=\{0,1,2,\cdots\}$ 上的函数来定义的。因为 $T(n)$ 一般仅定义在整数的输入规模上。

1. $\Theta$ ——渐近确界(上、下限，lower bound and upper bound)

$\Theta(g(n))=\{f(n)$：存在正常数 $c_1$，$c_2$和 $n_0$，使对所有的 $n\geqslant n_0$，有 $0\leqslant c_1g(n)\leqslant f(n)\leqslant c_2g(n)\}$，即对于任意一个函数 $f(n)$，若存在正常数 $c_1$，$c_2$，使当 $n$ 充分大时，$f(n)$能被夹在 $c_1g(n)$和 $c_2g(n)$中间，则 $f(n)$属于集合 $\Theta(g(n))$。

由于时间和空间需求都是非负值，当 $n$ 足够大时 $f(n)$ 是非负值(渐近正函数)，$g(n)$ 也必须是渐近非负的。

**例 1.1**　证明 $1/2n^2-3n=\Theta(n_2)$。

**证明**　要证明存在常数 $c_1$，$c_2$ 和 $n_0$，使得对所有的 $n\geqslant n_0$，有 $c_1n^2\leqslant 1/2n^2-3n\leqslant c_2n^2$ 成立。

用 $n^2$ 除以上述不等式得

$$c_1\leqslant\frac{1}{2}-\frac{3}{n}\leqslant c_2$$

于是

$$c_2\geqslant\frac{1}{2}-\frac{3}{n}$$

得知：$c_2\geqslant 1/2$ 时对所有 $n\geqslant 1$ 成立。

同样可知，$c_1\leqslant 1/2-3/n$ 在 $c_1\leqslant 1/14$　时对所有 $n\geqslant 7$ 成立。

因此，取 $c_1=1/14$，$c_2=1/2$，$n_0\geqslant 7$，有 $c_1n^2\leqslant n^2-3n\leqslant c_2n^2$ 成立。

2. $O$——(渐近上界，upper bound)

$O$ 记号一般是指最小上界。

$O(g(n))=\{f(n)$：存在正常数 $c$ 和 $n_0$，使对所有的 $n\geqslant n_0$，有 $0\leqslant f(n)\leqslant cg(n)\}$，即对于任意一个函数 $f(n)$，若存在正常数 $c$，使当 $n$ 充分大时，$f(n)$ 的值总在 $cg(n)$ 之下，则 $f(n)$ 属于集合 $O(g(n))$。

为了表示一个函数 $f(n)$ 是集合 $O(g(n))$ 的一个元素，记

$$f(n)=O(g(n))$$

按照大 $O$ 的定义，容易证明它有如下运算规则为

$$O(f)+O(g)=O(\max(f,g))$$
$$O(f)+O(g)=O(f+g)$$
$$O(f)\cdot O(g)=O(f\cdot g)$$

如果 $g(N)=O(f(N))$，则

$$O(f)+O(g)=O(f)$$
$$O(Cf(N))=O(f(N))$$

其中，$C$ 是一个正的常数；$f=O(f)$。

3. $\Omega$——(渐近下界，lower bound)

$\Omega$ 记号一般是指最大下界。

$\Omega(g(n))=\{f(n)$：存在正常数 $c$ 和 $n_0$，使对所有的 $n\geqslant n_0$，有 $0\leqslant cg(n)\leqslant f(n)\}$，即对于任意一个函数 $f(n)$，若存在正常数 $c$，使当 $n$ 充分大时，$f(n)$ 的值等

于或大于 $cg(n)$，则 $f(n)$ 属于集合 $\Omega(g(n))$。

4. $o$ 记号

$o$ 记号所提供的渐近上界可能是也可能不是渐近紧确的。例如，$2n^2=O(n^2)$ 是渐进紧确的，$2n=o(n^2)$ 就不是渐近紧确的。使用 $o$ 符号来表示非渐近紧确的上界，即 $f(n)=o(g(n))=\{$对任意正常数 $c$，存在常数 $n_0>0$，使得所有的 $n\geqslant n_0$，有 $0\leqslant f(n)\leqslant cg(n)\}$。

5. $\omega$ 记号

$\omega$ 记号与 $\Omega$ 记号的关系就像 $o$ 记号和 $O$ 记号的关系一样。

$f(n)=\omega(g(n))=\{f(n)$：对任意正常数 $c>0$，存在常数 $n_0>0$，使得所有的 $n\geqslant n_0$，有 $0\leqslant cg(n)<f(n)\}$。

我们在学习算法的时候，总会有这样的疑惑：如此众多的算法，是怎样想到的？这些算法为什么是正确的呢？其实林林总总的算法背后，无不隐藏着真正的解题之道。解题之道博大精深，该从何谈起？上面的疑惑又该如何解决？两千五百多年前，著名的哲学家和思想家老子对道之本质参透得可谓淋漓尽致。参悟解题之道，仍数学归纳法开始。

## 1.3 数学归纳法

归纳思想是从特殊到一般的思维方法，即通过对有关数据和资料的分析，建立数学模型，探索并发现数学问题中蕴涵的规律。因此，归纳思想是一种重要的数学思想，是人类探索真理和发现真理的主要工具之一。

### 1.3.1 第一数学归纳法

设 $P(n)$ 是一个与正整数有关的命题，如果：

(1) 当 $n=n_0(n_0\in N)$ 时，$P(n)$ 成立；

(2) 假设 $n=k(k\geqslant n_0,k\in N)$ 成立，由此推得 $n=k+1$ 时，$P(n)$ 也成立。

于是，根据(1)、(2)对一切正整数 $n\geqslant n_0$ 时，$P(n)$ 成立。

**例 1.2** 设 $T$ 是 $m$ 元正则树，其树叶数为 $t$，分支数为 $i$，求证：$(m-1)i=t-1$。

**证明** 当 $i=1$ 时，结论显然成立。

设当 $i=k$ 时，定理成立，即有 $(m-1)k=t-1$。

现在分支数增加 1，得 $i=k+1$，此时，树叶数增加 $m-1$（因为原来的一个树叶变成分支了），此时的式子 $(m-1)k=t-1$ 中，左端 $k$ 变为 $k+1$，而右端需改为 $(t-1)+(m-1)$，仍有 $(m-1)i=t-1$，即定理得证。