国家精品课程主讲教材

数据结构：题解与拓展

翁惠玉　俞　勇　编著

高等教育出版社
HIGHER EDUCATION PRESS

国家精品课程主讲教材

数据结构：题解与拓展

Shuju Jiegou: Tijie yu Tuozhan

翁惠玉　俞　勇　编著

高等教育出版社·北京
HIGHER EDUCATION PRESS　BEIJING

内容提要

本书是国家精品课程“数据结构”（上海交通大学）的主讲教材之一，并与主教材《数据结构：思想与实现》（翁惠玉、俞勇编著）相配套。本书总结了主教材各章的主要内容以及重点难点，并对主教材中的习题进行了分析和解答。作为对主教材的补充，本书在大多数章中都增加了一个拓展部分，使学有余力的学生能够进一步深入地学习数据结构。本书概念清楚，内容丰富，通过学习，可以帮助学生进一步巩固数据结构的知识。

本书可作为高等学校计算机及相关专业“数据结构”课程的教学辅导教材，也可以作为全国计算机专业硕士研究生入学考试的辅导用书。

图书在版编目(CIP)数据

数据结构：题解与拓展/翁惠玉，俞勇编著．—北京：高等教育出版社，2011.7

ISBN 978-7-04-032639-0

Ⅰ．①数… Ⅱ．①翁… ②俞… Ⅲ．①数据结构-高等学校-教学参考资料 Ⅳ．①TP311.12

中国版本图书馆 CIP 数据核字(2011)第 112972 号

策划编辑 刘 艳　责任编辑 刘 艳　封面设计 于文燕　版式设计 范晓红
插图绘制 尹 莉　责任校对 胡美萍　责任印制 田 甜

出版发行 高等教育出版社
社　　址 北京市西城区德外大街 4 号
邮政编码 100120
印　　刷 北京铭传印刷有限公司
开　　本 787 × 1092　1/16
印　　张 28
字　　数 630 000
购书热线 010-58581118
咨询电话 400-810-0598
网　　址 http://www.hep.edu.cn
　　http://www.hep.com.cn
网上订购 http://www.landraco.com
　　http://www.landraco.com.cn
版　　次 2011 年 7 月第 1 版
印　　次 2011 年 7 月第 1 次印刷
定　　价 38.00 元

物 料 号 32639-00

前　言

数据结构是计算机专业最基础，也是最重要的课程之一。它和程序设计一起为学习计算学科的其他后继课程奠定了基础。

数据结构又是实践性很强的课程。要学好数据结构，不仅需要理解教材中的每个知识点，还需要做一定数量的习题，编写一定量的代码，这样才能够熟练掌握各种数据结构的实现，并灵活应用各种数据结构。

本书是作者编著的《数据结构：思想与实现》的配套教学辅导教材。希望通过学习本书，可以进一步帮助读者解决学习中的重点和难点，从而更好地掌握数据结构的知识。

本书在内容安排上与《数据结构：思想与实现》相呼应。每章基本上都包括4个部分：重点难点、主要内容、习题解答和进一步拓展。重点难点部分总结了每一章学习的重点和难点；主要内容部分是对主教材相应章的内容的概括；习题解答部分给出了各章章后所附习题的分析与解答；进一步拓展部分是对主教材的补充，介绍了一些主教材没有提到，但也会被经常用到的数据结构以及这些数据结构的应用。

尽管本书基本上给出了所有习题的解答，但读者切莫盲目依赖答案。本书正确的使用方法应该是首先独立思考，找出解题的思路；若确实没有思路，可以先复习相关的知识，然后再看本书的解题思路，编写相应的程序；若程序设计的基础不是太好，有了思路而写不出程序，则可参考本书的程序来编写自己的程序。

本书在编写过程中得到了上海交通大学2009级ACM班全体同学的大力支持。其中的很多解题思路或程序都是由他们提供的。在此表示衷心的感谢！

本书可作为高等学校计算机专业及相关专业的“数据结构”课程的教学辅导用书，也非常适合作为读者的自学参考用书。

由于作者水平有限，本书可能存在很多不足，敬请读者批评指正。

作　者

2011年3月

目　录

第1章 绪 论

1.1 重点难点

虽然应用系统千变万化,但不同应用系统中数据元素之间的逻辑关系的种类是有限的。数据结构抛开了各种具体应用、具体数据元素的内容,通过抽象的方法研究被处理的数据元素之间有哪些逻辑关系(称为逻辑结构),对于每种逻辑关系可能有哪些操作。此外,还研究每种逻辑关系在计算机内部如何表示(称为物理结构),以及对于每一种物理结构,对应的操作如何实现。

每个数据结构处理一类逻辑关系,包括逻辑关系的物理表示以及操作的实现。因此,数据结构的讨论可以分为两层:抽象层和实现层。抽象层讨论数据的逻辑结构和所需的操作,对应于一个抽象类;实现层讨论数据的存储表示及操作的实现,每种实现方法对应一个具体的类。

一个操作可能有多种实现的方法,每种实现方法就是一个算法。如何评价这些算法也是本章的重点之一。本章介绍了大 O 表示法、大 O 表示法的计算方法以及如何通过大 O 表示法比较算法的优劣。

经过对本章的学习,应该掌握:

- 数据的逻辑结构有哪几种;
- 数据的物理结构有哪几种;
- 算法的概念;
- 算法的空间复杂度及计算;
- 算法的时间复杂度及计算。

1.2 主要内容

1.2.1 数据的逻辑结构

从概念上讲,一个数据结构是由一组同类的数据元素依据某种联系组织起来的。数据元素间的逻辑关系称为数据的**逻辑结构**。不管应用如何变化,从抽象层面上看,数据的逻辑结构有以下 4 种。

(1) 集合结构:元素间的次序是任意的。元素之间除了“属于同一集合”的联系外没有其

他的关系。由于集合结构的元素间没有固有的关系，因此可以借助于其他结构来表示集合结构。

（2）线性结构：数据元素构成一个有序序列。其中，第一个元素只有后继没有前驱，最后一个元素只有前驱没有后继。除了第一个和最后一个元素外，其余元素都有一个前驱和一个后继。

（3）树形结构：除了一个特殊的根元素外，每个元素有且仅有一个前驱，后继数目不限。根元素没有前驱。树形结构表示的是一种层次关系。

（4）图形结构：图是最一般的逻辑结构，图中的每个元素的前驱和后继数目都不限。

这 4 种结构如图 1-1 所示。有时，也把线性结构以外的其他 3 种结构称为非线性结构。

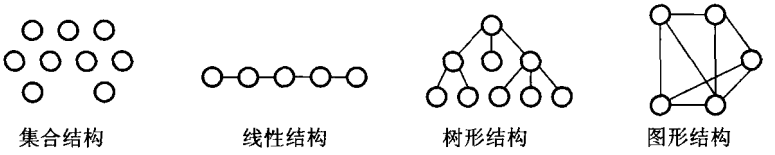

图 1-1　数据的逻辑结构

1.2.2　数据结构的存储实现

每个数据元素被存放在一个称为结点的单元中。一个结点通常是一个内置类型、记录类型或类类型的对象。数据元素之间的关系通常有下列 4 种方式来存储。

（1）顺序实现：所有的结点存放在一块连续的存储区域中，结点之间的逻辑关系可以通过结点的存储位置来体现。在高级语言中，一块连续的存储空间通常可用一个数组来表示。因此，顺序实现通常用一个数组来存储，数组元素的类型就是结点的类型，结点之间的关系通过数组元素的下标来体现。

（2）链接实现：结点可以分散地存放在存储器的不同位置，结点之间的关系通过一个或多个指针显式地指出。因此，在链接存储中，每个存储结点包含两个部分：数据元素部分和指针部分。数据元素部分保存数据元素的值；指针部分保存一组指针，每个指针指向一个与本结点有逻辑关系的结点。

（3）散列存储方式：是专用于集合结构的数据存放方式。在散列存储中，各个结点均匀地分布在一块连续的存储区域中，用一个散列函数将数据元素和存储位置关联起来。

（4）索引存储方式：所有的结点按照数据生成的次序连续存放。另外设置一个索引区域表示结点之间的关系。

1.2.3　算法分析

算法分析确定算法的性能随处理的数据量变化的规律，是评价算法优劣的依据。算法分析通常从时间和空间两个方面展开。算法的**时间性能**称为**时间复杂度**，**空间性能**称为**空间复杂度**。

前者指算法包含的运算量，后者指算法需要的存储量。

空间复杂度的讨论比较简单，它关心的是除了被处理的数据所占的空间之外所需的额外空间的大小。空间复杂度一般按最坏情况来分析。

在算法分析中，时间性能通常用**最好情况的时间复杂度**、**最坏情况的时间复杂度**以及**平均情况的时间复杂度**来描述。

算法的运算量可以用一个运行时间函数来刻画。算法的运行时间函数可能是一个很复杂的函数，解决同一问题的不同算法会有不同的运行时间函数，如何比较这些函数并从中选取出一个好的算法是一项很困难的工作。我们希望能找到一种比较简单、直观的描述方法。时间性能主要考虑的是当处理的数据量很大时运行时间随问题规模的变化规律。也就是对问题规模比较小的情况忽略不计，只考虑在问题规模大于一定值以后的情况。

同理，在问题规模很大时的运行时间函数也可以有各种各样的形式，那么如何判断一个函数的值总比另一个函数小呢？这将是很复杂的数学问题。在算法分析中，将这个问题进行了简化，不考虑具体的运行时间函数，只考虑运行时间函数的数量级。这种方法称为**渐进表示法**。最常用的渐进表示法是大 O 表示法。

定义 如果存在两个正常数 c 和 N_0，使得当 $N \geqslant N_0$ 时有 $T(N) \leqslant cF(N)$，则记为

$$T(N) = O(F(N))。$$

大 O 表示法为运行时间函数 $T(N)$ 找到了一个上界 $F(N)$。为便于比较算法的优劣，在选择 $F(N)$ 时通常选择比较简单的函数形式，并忽略低次项和系数。表 1-1 给出了常用的 F 函数及其名称。

表 1-1 常用的时间复杂度函数

函数	名称	函数	名称
1	常量	N^3	立方
$\log N$①	对数	2^N	指数
N	线性	$N!$	指数
$N\log N$	$N\log N$	N^N	指数
N^2	平方		

时间复杂度函数为多项式的算法称为**多项式时间算法**，时间复杂度函数为指数函数的算法称为**指数时间算法**。最常见的多项式时间算法的时间复杂度之间的关系为

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

① 本书中的对数函数没有给出对数基，原因是对于任何大于 1 的常数 a 和 b，都有 $\log_a n = \log_b n / \log_b a$，所以 $\log_a n$ 和 $\log_b n$ 只差一个常数，而常数在大 O 表示法中是被忽略的。

即 $O(1)$是最好的算法,其次是 $O(\log N)$。指数时间算法的时间复杂度之间的关系为

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

而多项式时间的算法要比指数时间的算法好。这样就能为算法的时间性能评价给出依据。例如,一个时间复杂度为 $O(N)$的算法要比一个时间复杂度为 $O(N^3)$的算法好。在用大 O 表示法表示算法的时间性能时,要注意不要包含低次项或系数。例如,$O(2N)$或 $O(N^3+N)$都是不好的表示方式,正确的表示方式是 $O(N)$和 $O(N^3)$。因为当 N 很大时,这些系数和低次项都可忽略。

为计算算法的时间复杂度,有如下两个定理。

求和定理 假定 $T_1(n)$,$T_2(n)$是程序段 P_1,P_2的运行时间,并且 $T_1(n)$是 $O(f(n))$的,而 $T_2(n)$是 $O(g(n))$的。那么,先运行 P_1,再运行 P_2的总的运行时间是 $T_1(n)+T_2(n)=O(\mathrm{MAX}(f(n),g(n)))$。

求积定理 如果 $T_1(n)$和 $T_2(n)$分别是 $O(f(n))$和 $O(g(n))$的,那么 $T_1(n)\times T_2(n)$是 $O(f(n)\times g(n))$。

这两个定理为时间复杂度的计算带来了很大的便利,从而可以不必根据算法的过程抽取出确切的运行时间函数。从这两个定理可以得到如下的计算规则。

- 规则 1 每个简单语句,如赋值语句、输入输出语句,它们的运行时间与问题规模无关,在每个计算机系统中运行时间都是一个常量,因此时间复杂度为 $O(1)$。
- 规则 2 条件语句,if <条件> then <语句> else <语句>的运行时间为执行条件判断的代价,一般为 $O(1)$,再加上执行 then 后面的语句的代价(若条件为真),或执行 else 后面的语句代价(若条件为假)之和。根据求和定理可知,条件语句的时间复杂度为 MAX(O(then 子句),O(else 子句))。
- 规则 3 循环语句往往与处理的数据量有关,是分析的重点。循环语句的执行时间是循环控制行和循环体执行时间的总和。循环控制行一般是一个简单的条件判断,因此循环语句的执行时间是循环体的运行时间乘以循环次数。
- 规则 4 嵌套循环语句,对外层循环的每个循环周期,内存循环都要执行它的所有循环周期,因此,可用求积定理计算整个循环的时间复杂度,即最内层循环体的运行时间乘以所有循环的循环次数。
- 规则 5 连续语句,利用求和定理把这些语句中的时间复杂度的最大者作为整个语句段的时间复杂度。

有了这些简化方法后,求算法的时间复杂度就简单多了。只要找出最复杂、运行时间最长的程序段,该程序段的时间复杂度就是整个程序的时间复杂度。

1.3 习题解答

1.3.1 简答题

1. 什么是算法的时间、空间复杂度?如何度量算法的时间、空间复杂度?

解 算法所需的运算量与问题规模之间的关系称为算法的“时间复杂度”。“空间复杂度”指出了除被处理的数据所占空间之外所需的额外空间的大小，空间复杂度一般按照最坏情况来分析。时间复杂度则通常用“最好情况的时间复杂度”、“最坏情况的时间复杂度”以及“平均情况的时间复杂度”来描述。通常在计算时，使用“渐进表示法”，即只考虑运行时间函数的数量级。

2. 从抽象层面上讲，有哪些基本的逻辑结构？

解 有 4 种基本的逻辑结构：集合结构、线性结构、树形结构和图形结构。

3. 数据结构研究的主要内容是什么？

解 数据结构研究的主要内容是一组有特定关系的信息的存储和处理方法。数据结构抛开具体的数据元素的类型，仅研究数据元素之间的关系，将数据之间的逻辑关系抽象出线性、树形、集合和图形 4 种结构，然后分别研究如何在计算机中存储每种结构及在这种存储结构下基本操作的实现方法。

4. 什么是存储实现？什么是运算实现？

解 存储实现的基本目标是建立数据的机内表示，包括两个部分：数据元素的存储和数据元素之间关系的存储。在某种存储实现基础上的各种操作的具体实现称为“运算实现”。运算实现的核心是设计实现某一操作的处理步骤，即算法设计。

5. 为什么数据结构都用类模板来描述？

解 由于数据结构关注的是数据元素之间的关系是如何保存的，基于这些关系的操作是如何实现的，而数据元素可以是任意类型。使用类模板来描述，可以避免对于具体的数据元素类型的依赖。

6. 在面向对象的数据结构中，抽象类的作用是什么？

解 抽象类为其派生类规定了统一的用户接口。在数据结构中，用抽象类规定一类数据结构的基本运算的调用方法。具体的数据结构的实现类必须从抽象类继承。这样就保证了数据结构的不同实现有同样的用户接口。

7. 什么是最好、最坏和平均情况下的时间复杂度？

解 最好情况的时间复杂度是算法在所需运行时间最少的情况下的时间复杂度；最坏情况的时间复杂度是算法在所需运行时间最多的情况下的时间复杂度；平均情况的时间复杂度是在所有可能的情况下，算法平均所需的时间。

8. 什么是多项式时间算法？什么是指数时间算法？

解 时间复杂度为多项式函数的算法称为多项式时间算法，时间复杂度为指数函数的算法称为指数时间算法。

9. 把以下函数按照等价的大 O 分组。

$$x^2, x, x^2+x, x^2-x, \text{和 } x^3/(x-1)$$

解 可以将这些函数分成两大类：一类是 $O(x)$；另一类是 $O(x^2)$。

$O(x)$ 的函数有 x；

$O(x^2)$的函数有 $x^2, x^2+x, x^2-x, x^3/(x-1)$。

10. 给出下列代码的大 O 分析。

(1)
```
sum = 0;
for (i = 0; i < n; i++)
    sum++;
```

(2)
```
sum = 0;
for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
        sum++;
```

(3)
```
sum = 0;
for(i = 0; i < n; i++)
    for(j = 0; j < n * n; j++)
        sum++;
```

(4)
```
sum = 0;
for(i = 0; i < n; i++)
    for(j = 0; j < i; j++)
        sum++;
```

解 (1) 语句段由一个赋值语句和一个重复 n 次的循环组成。赋值语句的时间复杂度是 $O(1)$。根据求和定理,循环语句的时间复杂度就是整个语句段的时间复杂度。该循环的循环体是一个自增语句,其时间复杂度是 $O(1)$。循环的循环次数是 n,因此整个循环的时间复杂度是 $O(n)$。

(2) 语句段由一个赋值语句和一个嵌套循环组成。根据求和定理,嵌套循环语句的时间复杂度就是整个语句段的时间复杂度。该嵌套循环的外层循环和里层循环的循环次数都是 n,最内层的循环体的时间复杂度是 $O(1)$。根据求积定理,该循环的时间复杂度是 $O(n^2)$。

(3) 语句段由一个赋值语句和一个嵌套循环组成。根据求和定理,循环语句的时间复杂度就是整个语句段的时间复杂度。该循环的外层循环 n 次,内层循环 $n \times n$ 次,最内层循环体的时间复杂度是 $O(1)$。根据求积定理,该循环的时间复杂度是 $O(n^3)$。

(4) 语句段由一个赋值语句和一个嵌套循环组成。根据求和定理,循环语句的时间复杂度就是整个语句段的时间复杂度。该循环的外层循环的循环次数是 n,内层循环的可能的最大循环次数也是 n,最内层的循环体的时间复杂度是 $O(1)$。根据求积定理,该循环的时间复杂度是 $O(n^2)$。

1.3.2 程序设计题

1. 素数是除了 1 和它本身之外没有其他因子的数,写一个程序,判断一个正整数 n 是不是素数。对于 n,你的程序在最坏情况下的时间复杂度是多少?

解 判断一个数是否为素数，是一个有趣的问题。它有很多种不同的解法。这些算法有不同的时间复杂度。

方法一：直接从素数的定义出发。最直接的方法是统计因子的个数，检查因子个数是否为 2 个。如果正好是 2 个因子，则为素数，否则为非素数。由常识知，n 的因子必须小于或等于 n，因此只要检查所有 1 到 n 之间的数，就会找出所有的因子。根据这个结果，可以设计出下列决定 n 是否为素数的算法。

（1）检查 1 到 n 之间的每个数，看它是否能整除 n。

（2）每次遇到一个因子，计数器加 1。

（3）在所有的数都被测试以后，检查计数器的值是否为 2。

这个策略的实现体现在函数 IsPrime 中，见代码清单 1-1。如该函数原型所指出，IsPrime 接收一个整数 n，返回一个布尔值，是一个谓词函数。该实现中用变量 divisors 保存至今为止发现的因子数。divisors 的初值被设为 0，当发现一个新的因子时，divisors 加 1。如果在检查了 1 到 n 之间的所有数之后，divisors 正好等于 2，则 n 是素数。这个测试被表示为布尔表达式 divisors == 2，函数将这个值作为它的返回值返回。

代码清单 1-1 判断 n 是否为素数的最直接的算法

```
1.  bool IsPrime(int n)
2.  {  int divisors = 0;
3.     for(int i = 1;i <= n; ++ i)
4.        if(n % i == 0) ++ divisors;
5.     return(divisors == 2);
6.  }
```

上述函数检查了 1 到 n 之间的每个数，因此它的时间复杂度是 $O(n)$ 的。

方法二：通过几个途径可以修改方法一中的算法，以改善 IsPrime 函数实现的效率。

（1）IsPrime 没有必要检查所有的因子。只要发现任何一个大于 1 且小于 n 的因子，就能停下来报告 n 不是素数。

（2）一旦函数已经检查了 n 是否能被 2 整除，就不需要检查是否能被其他偶数整除。如果 n 能被 2 整除，程序就停下来，报告 n 不是素数。如果 n 不能被 2 整除，那么它也不可能被 4 或 6 或其他偶数整除。因此，IsPrime 只需要检查 2 和奇数。但注意有个特例，2 能被 2 整除，但 2 是素数。

（3）该问题不需要检查到 n 为止。实质上，它可以在一半的地方就停止，因为任何大于 $n/2$ 的值不可能被 n 整除。然而再进一步思考一下，还可以证明，该程序不需要试探任何大于 n 的平方根的因子。当 n 能被某一个整数 d_1 整除时，由可整除的定义，n/d_1 是一个整数，称之为 d_2。d_1 和 d_2 的范围是什么？因为 n 是等于 $d_1 \times d_2$，如果其中一个因子大于 n 的平方根，另一个因子一定小于 n 的平方根。因此，如果 n 有任何因子的话，肯定有一个小于它的平方根。这个结果意味着程序中的 for 循环的次数 $i \leqslant \sqrt{n}$。

按照上面3个途径修改后的算法如代码清单1-2所示，该算法的时间复杂度为$O(\sqrt{n})$。这意味着，当$n=10\ 000$时，算法一要检查10 000个数，而算法二只需要检查100个数。

代码清单1-2　改进的IsPrime算法

```
1.  bool IsPrime(int n)
2.  { int limit;
3.    if(n <= 1) return false;
4.    if(n == 2) return true;
5.    if(n % 2 == 0) return false;
6.    limit = sqrt(n) + 1;
7.    for(int i = 3;i <= limit;i += 2)          //最坏情况下是n的平方根
8.          if(n % i == 0) return false;
9.    return true;
10. }
```

2. 设计一个函数，计算$S=1-2+3-4+5-6+\cdots+/-N$的值。要求时间复杂度为$O(1)$。

解　对于这样的问题，程序员首先想到的方法可能是用一个N次的计数循环，这个算法的时间复杂度是$O(N)$的。进一步观察这个序列，可以发现第1项和第2项的和是-1，第3项和第4项的和也是-1，任何一个奇数项和它后面的偶数项之和都是-1。据此可以得到：如果N是偶数，那么和为$-N/2$；如果N是奇数，则和为$-(N-1)/2+N$。这样就不需要用循环，直接用一个表达式就可以得到结果，其时间复杂度为$O(1)$，按照这个思想得到的代码见代码清单1-3。

代码清单1-3　程序设计题2的实现

```
1.  int sum(int n)
2.  { if((n % 2) == 0)
3.          return -n/2;
4.    else return -n/2 + n;
5.  }
```

1.4　进一步拓展

任何一个问题都可能有多种解决方案。本节将以最大公因子问题为例进一步加以说明。

计算时间复杂度时，通常假设表达式语句的时间复杂度都是$O(1)$，所以分析的重点在于与问题规模有关的循环语句。但如果程序中的某个表达式语句包含了一个函数调用，则此语句的时间复杂度不再是$O(1)$。下面将以计算$n!$的递归函数为例说明如何分析带有函数调用的算法的时间复杂度。

1.4.1　最大公因子问题

最大公因子的问题：给出两个正整数x和y，最大公因子（或缩写为GCD）是能够同时被两个

正整数整除的最大数。试设计一个函数 int GCD(int x,int y),求出 x 和 y 两个正整数的最大公因子。

计算最大公因子问题有多种算法。下面讨论两种常用的方法。

最简单的方法是采用蛮力算法。该方法测试每一种可能性。一开始,简单地"猜测"GCD(x,y)是 x,因为 x 的因子不可能大于 x。然后检查这个假设,将其去除 x 和 y,检查能否整除。如果能整除,答案就有了。如果不能,将这个假设值减 1,再继续测试,直到找到一个都能整除 x 和 y 的数或假设值减到了 1。前者找到了最大公因子,后者表示 x 和 y 没有最大公因子。蛮力算法的实现如代码清单 1-4 所示。

代码清单 1-4　GCD 的蛮力算法

```
1.  int GCD(int x,int y)
2.  { int g = x;
3.
4.    while(x % g != 0 || y % g != 0) --g;
5.    return g;
6.  }
```

蛮力算法不是一个有效的策略。事实上,如果只关心效率的话,蛮力算法是一个很差的选择。例如,考虑一下对 1 000 005 和 1 000 000 调用这个函数会发生什么。蛮力算法在找到最大公因子 5 之前将运行 100 万次 while 的循环体。

古希腊的数学家欧几里得提出了一个解决这个问题的非常出色的算法,称为辗转相除法,也被称为欧几里得算法。该算法描述如下:

1. 取 x 除以 y 的余数,称余数为 r。
2. 如果 r 是 0,过程完成,答案是 y。
3. 如果 r 非 0,设 x 等于原来 y 的值,y 等于 r,重复整个过程。

将这个算法翻译成的 GCD 函数如代码清单 1-5 所示。

代码清单 1-5　欧几里得算法

```
1.  int GCD(int x,int y)
2.  { int r;
3.    while(true){
4.      r = x % y;
5.      if(r == 0)break;
6.      x = y;
7.      y = r;
8.    }
9.    return y;
10. }
```

为了说明两个算法效率上的不同，考虑两个数 10 001 005 和 1 000 000。为了找出这两个数的最大公因子，欧几里得算法只需要两步。欧几里得算法开始时，x 是 1 000 005，y 是 1 000 000，在第一个循环周期中 r 被设为 5。由于 r 的值不为 0，程序设 x 为 1 000 000，设 y 为 5，重新开始。在第二个循环周期，r 的新值是 0，因此程序从 while 循环退出，并报告答案为 5。

1.4.2 递归函数的时间复杂度的计算

代码清单 1-6 是计算 $n!$ 的递归实现，试计算它的时间复杂度。

代码清单 1-6 $n!$ 的递归实现

```
1.  int p(int n)
2.  { if(n<=0)return 0;
3.    if(n==0)return 1;
4.    else return n*p(n-1);
5.  }
```

设 $T(n)$ 是计算 p(n) 所需的时间。当 n 小于或等于 0 时，满足递归终止条件，函数直接返回。因此 $T(0)$ 是一个常量 c。当 n 大于 0 时，执行 else 子句。该子句先调用 p(n-1)，然后将结果与 n 相乘并返回。执行 p(n-1) 所需的时间是 $T(n-1)$，乘法操作是一个常量时间，因此 $T(n)=T(n-1)+c$。将此公式用于 $n-1$，$n-2$ 等情况，可得

$$T(n-1)=T(n-2)+c$$

$$T(n-2)=T(n-3)+c$$

$$\cdots$$

$$T(2)=T(1)+c$$

$$T(1)=c$$

将上面各式相加，可得

$$T(n)=nc$$

所以，p 函数的时间复杂度是 $O(n)$ 的。