第 1 部分　算法和算法分析

第 1 章　算法问题求解基础

算法是计算机学科的一个重要分支，它是计算机科学的基础，更是计算机程序的基石。算法是计算机求解问题的特殊方法。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，还要学习设计新算法和分析算法性能的方法。本章给出算法的基本概念，介绍使用计算机求解问题的过程和方法，讨论递归算法及证明递归算法正确性的归纳法。

1.1　算法概述

1.1.1　什么是算法

在学习一门计算机程序设计语言，如 C/C++或 Pascal 之后，应该对算法一词不再陌生。编写一个程序，实际上是在实现使用计算机求解某个问题的方法。在计算机科学中，算法一词用于描述一个可用计算机实现的**问题求解**（problem-solving）方法。

什么是算法？笼统地说，**算法**（algorithm）是求解一类问题的任意一种特殊的方法。较严格的说法是，一个算法是对特定问题求解步骤的一种描述，它是指令的有限序列。此外，算法具有下列 5 个特征。

（1）**输入**（input）：算法有零个或多个输入量；

（2）**输出**（output）：算法至少产生一个输出量；

（3）**确定性**（definiteness）：算法的每一条指令都有确切的定义，没有二义性；

（4）**能行性**（effectiveness）：算法的每一条指令必须足够基本，它们可以通过已经实现的基本运算执行有限次来实现；

（5）**有穷性**（finiteness）：算法必须总能在执行有限步之后终止。

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。算法是一个指令序列。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”这样的指令。另一方面，算法的每条指令必须是能行的。对一个计算机算法而言，能行性要求一条算法指令应当最终能够由执行有限条计算机指令来实现。例如，一般的整数算术运算是能行的，但如果 1÷3 的值需由无穷的十进制展开的实数表示，就不是能行的。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。如果取消有穷性限制，则只能称为**计算过程**（computational procedure）。

描述一个算法有多种方法，可以用自然语言、流程图、伪代码和程序设计语言来描述。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。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

机，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。

算法概念并不是计算机诞生以后才有的新概念。计算两个整数的最大公约数的辗转相除法是由古希腊欧几里得（约公元前 330—275 年）在他的《几何原本》（Euclid's Elements）中提出的，它是算法研究最重要的早期成果。直到 1950 年左右，**算法**一词还经常与欧几里得算法（Euclid's algorithm）联系在一起。中国的珠算口诀可视为典型的算法，它将复杂的计算（如除法）描述为一系列的算珠拨动操作。

欧几里得算法又称辗转相除法，用于计算两个整数 m 和 n（$0 \leq m < n$ 的最大公约数，记为 gcd(m, n)。其计算过程是重复应用下列等式，直到 $n \bmod m = 0$。

$$\gcd(m, n) = \gcd(n \bmod m, m)，对于 m > 0 \qquad (1\text{-}1)$$

式中，$n \bmod m$ 表示 n 除以 m 之后的余数。因为 gcd(0, n) = n，n 的最后取值也就是 m 和 n 的最大公约数。例如，gcd(24, 60) = gcd(12, 24) = gcd(0, 12) = 12。

欧几里得算法使用了递归，其 C/C++语言描述见程序 1-1。欧几里得算法的迭代形式描述见程序 1-2。请注意数学上的运算符 mod 与 C/C++语言的“%”运算符的区别①。

【程序 1-1】 欧几里得递归算法

```
void Swap(int&a,int&b)
{
    int c=a;a=b;b=c;
}
int RGcd(int m,int n)
{
    if(m==0) return n;
    return RGcd(n%m,m);
}
int Gcd(int m,int n)
{
    if (m>n) Swap(m,n);
    return RGcd(m,n);
}
```

【程序 1-2】 欧几里得迭代算法

```
int Gcd(int m,int n)
{
     if (m==0) return n; if (n==0) return m;
     if (m>n) Swap(m,n);
     while(m>0){
          int c=n%m;n=m;m=c;
     }
     return n;
}
```

① 运算符 mod 是对模数求剩余。设 $M>0$，$x \bmod M$ 的值在[0, M–1]中。使用 C/C++语言的“%”运算符实现 mod 运算的方法为：$x = x \,\%\, M$; if ($x<0$) $x = M+x$。

上述程序必定会结束，因为每循环一次，m 的新值就会变小，但绝对不会成为负数，当 $m = 0$ 时程序终止。

最大公约数问题还可以有其他算法。程序 1-3 的连续整数检测算法的依据直接来自最大公约数的定义：m 和 n 的最大公约数是能够同时整除它们的最大正整数。显然，一个公约数不会大于两数中的较小者，因此，可以先令 $t = \min\{m, n\}$，然后检查 t 能否分别整除 m 和 n，若能，则 t 就是最大公约数，否则令 t 减 1 后继续检测。程序 1-3 必定会终止。如果 m 和 n 的最大公约数是 1，则当 $t = 1$ 时，程序终止。

【程序 1-3】 Gcd 的连续整数检测算法

```
int Gcd(int m,int n)
{
    if (m==0) return n;if (n==0) return m;
    int t=m>n?n:m;
    while (m%t || n%t) t--;
    return t;
}
```

从上面的讨论可知，一个问题可以设计不同的算法来求解，针对同一个问题的算法也许基于完全不同的解题思路。求两个整数的最大公约数可以采用欧几里得算法，也可以采用连续整数检测算法，两种算法的解题速度会有显著差异。此外，同一个算法可采用不同的形式来表示。例如，欧几里得算法可以写成递归形式，也可以写成迭代形式。

1.1.2 为什么学习算法

算法是计算机科学的基础，更是程序的基石，只有具有良好的算法基础才能成为训练有素的软件人才。对于计算机专业的学生来说，学习算法的理由是非常充分的。因为你必须知道来自不同计算领域的重要算法，你也必须学会设计新的算法、确认其正确性并分析其效率。随着计算机应用的日益普及，各个应用领域的研究和技术人员都在使用计算机求解他们各自专业领域的问题，他们需要设计算法，编写程序，开发应用软件，所以学习算法对于越来越多的人来说变得十分必要。

著名的美国计算机科学家克努特（D. E. Knuth）说过：“一个受过良好的计算机科学知识训练的人知道如何处理算法，即构造算法、操纵算法、理解算法和分析算法。算法知识远不只是为了编写好的计算程序，它是一种具有一般意义的智能工具，必定有助于对其他学科的理解，不论化学、语言学或者音乐等。”

哈雷尔（David Harel）在他的《算法学——计算的灵魂》一书中说：“算法不仅是计算机学科的一个分支，它更是计算机科学的核心，而且可以毫不夸张地说，它和绝大多数科学、商业和技术都是相关的。”

1.2 问题求解方法

软件开发的过程是使用计算机求解问题的过程。使用计算机解题的核心任务是设计算法。算法并非就是问题的解，它是准确定义的，用来获得问题解的计算过程的描述。算法是问题的程序化解决方案。显然，算法能够求解的问题种类是有局限性的，它们不可能求解现实世界中的所有问题。本书讨论的问题都是指能用算法求解的问题。

1.2.1 问题和问题求解

什么是**问题**（problem）？只要目前的情况与人们所希望的目标不一致，就会产生问题。例如，排序问题是：任意给定一组记录，排序的目的是使得该组记录按关键字值非减（或非增）次序排列。

问题求解（problem solving）是寻找一种方法来实现目标。问题求解是一种艺术，没有一种通用的方法能够求解所有问题。有时，人们不得不一次又一次地尝试可能的求解方法，直到找到一种正确的求解途径。一般说来，问题求解中存在着猜测和碰运气的成分。然而，当我们积累了问题求解的经验，这种对问题解法的猜测就不再是完全盲目的，而是形成某些问题求解的技术和策略。**问题求解过程**（problem solving process）是人们通过使用问题领域知识来理解和定义问题，并凭借自身的经验和知识去选择和使用适当的问题求解策略、技术和工具，将一个问题描述转换成问题解的过程。

现在，很多问题可以用计算机求解，计算机的应用已渗透到人类活动的方方面面。有些问题，如四色问题，如果没有计算机，至今恐怕难以求解。计算机求解问题的过程就是一个计算机软件的开发过程，被称为**软件生命周期**（software life cycle）。

计算机求解问题的关键之一是寻找一种**问题求解策略**（problem solving strategy），得到求解问题的算法，从而得到问题的解。例如，求解前面提到的排序问题是指设计一种排序算法，能够把任意给定的一组记录排成有序的记录序列。

1.2.2 问题求解过程

匈牙利数学家乔治·波利亚（George Polya）在 1957 年出版的《How to Solve It》一书中概括了如何求解数学问题的技术。这种问题求解的四步法对于大多数其他科学也是适用的，同样也可用于求解计算机应用问题。

问题求解的四步法简述如下。

（1）**理解问题**（understand the problem）

毫无疑问，为了求解问题必须首先理解问题。如果不理解问题，当然就不可能求解它。此外，对问题的透彻理解有助于求解问题。这一步很重要，它看似简单，其实并不容易。在这一步，我们必须明确定义所要求解的问题，并用适当的方式表示问题。对于简单问题，不妨直接用自然语言描述问题，如排序问题。

（2）**设计方案**（devise a plan）

求解问题时，首先考虑从何处着手，考虑以前有没有遇到类似的问题，是否解决过规模较小的同类问题。此外，还应选择该问题的一些特殊例子进行分析。在此基础上，考虑选择何种问题求解策略和技术进行求解，以得到求解问题的算法。

（3）**实现方案**（carry out the plan）

实现求解问题的算法，并使用问题实例进行测试、验证。

（4）**回顾复查**（look back）

检查该求解方法是否确实求解了问题或达到了目的。评估算法，考虑该解法是否可简化、改进和推广。

对于上一小节讨论的求最大公约数问题，理解起来并不困难，但为了求解问题，则需要相关的数学知识。最简单的求解方案可以直接从最大公约数的定义出发得到，这就是程序 1-3 的连续整数检测算法。欧几里得算法建立在已经证明式（1-1）成立的基础上。对于这两种求解算法，

可以使用 m 和 n 的若干值进行测试，验证算法的正确性。通过比较，发现对于求最大公约数问题，连续整数检测算法和欧几里得算法的时间效率差别很大。递归的欧几里得算法又可改写成迭代形式，迭代算法的效率一般高于其对应的递归算法。

1.2.3 系统生命周期

一个计算机程序的开发过程就是使用计算机求解问题的过程。**软件工程**（software engineering）将软件开发和维护过程分成若干阶段，称为**系统生命周期**（system life cycle）或软件生命周期。系统生命周期法要求每个阶段完成相对独立的任务；各阶段都有相应的方法和技术；每个阶段都有明确的目标，要有完整的文档资料。这种做法便于各种软件人员分工协作，从而降低软件开发和维护的困难程度，保证软件质量，提高开发大型软件的成功率和生产率。

通常把软件生命周期划分为**分析**（analysis）、**设计**（design）、**编码**（coding 或 programming）、**测试**（testing）和**维护**（maintenance）等 5 个阶段。前 4 个阶段属于开发期，最后一个阶段处于运行期。

软件开发过程的前两个阶段“分析”和“设计”非常重要。“分析”是弄清楚需要“**做什么**（what）”，而“设计”是解决“**如何做**（how）”。

在问题求解的分析阶段，我们试图理解问题，弄清楚为了求解它必须做什么，而不是怎样做。在分析阶段，必须理解问题的需求。需求常常被分为功能需求和非功能需求两类。功能需求描述求解问题的程序必须具有的功能和特性，非功能需求是软件必须满足的约束等。例如，对一个整数序列进行排序的问题，其功能需求是将一个任意整数序列排列成非减有序序列，而非功能需求也许是代码和数据使用的内存空间不能超过 20MB，运行时间不超过 5min 等。这些需求应当被充分审查和讨论，并明确定义，形成**需求规范**（requirement specification）。问题定义必须明确，无二义性，且具有一致性。

设计阶段确定如何求解问题，包括选择何种问题求解策略和技术，如算法设计策略。在软件开发中，常采用逐步求精的方法，并用伪代码和流程图来设计和描述算法。

编码实现阶段的任务是编写程序，运行程序，并使用测试用例测试程序，验证程序的正确性。

1.3 算法设计与分析

1.3.1 算法问题求解过程

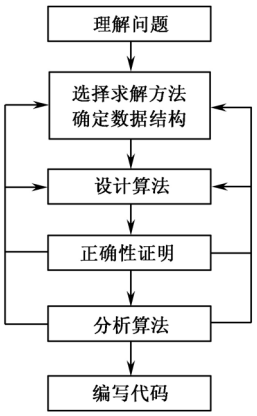

图 1-1 算法问题求解过程

算法问题的求解过程在本质上与一般问题的求解过程是一致的。具体求解步骤如图 1-1 所示。求解一个算法问题，需要先理解问题。通过仔细阅读对问题的描述，充分理解所求解的问题。为了完全理解问题，可以列举该问题的一些小例子，考虑某些特殊情况。

算法一般分两类：精确算法和启发式算法。一个**精确算法**（exact algorithm）总能保证求得问题的解。而一个**启发式算法**（heuristic algorithm）通过使用某种规则、简化或智能猜测来减少问题求解时间。它们也许比精确算法更有效，但其求解问题所需的时间常常因实例而异。它们也不能保证求得的解必定是问题的最优解，甚至不一定是问题的**可行解**（feasible solution）。一般来讲，启发式算法往往缺少理论依据。对于最优化问题，一个算法如果致力于寻找近似解而不是最优解，被称为**近似算法**（approximation algorithm）。近似算法求得

的应当是问题的可行解，但可能不是最优解。如果在算法中需做出某些随机选择，则称为**随机算法**（randomized algorithm）。随机算法执行的随机选择一般依赖于随机数发生器所产生的随机数。

在理解问题之后，算法设计者需要选择是否采取精确解法。有一些问题的确无法求得精确解，例如求平方根、求定积分和求解非线性方程。另一些问题虽然存在精确算法，但这些算法的求解时间慢得让人无法接受。例如，设计一个导致赢局的人机对弈程序并不困难，可以采用穷举算法。对于任何一种棋类，尽管其可能的棋局数目可谓天文数字，但总是有限的。我们总能设计一个算法对任意给定的一种棋局，判断这一棋局是否可能导致赢局，并由此决定下一步应走哪一着棋。采用这种以穷举方式逐一检查棋局的算法，每一步决策都将异常费时，即使在当代速度最快的计算机上运行也是不实际的。

启发式算法并不总能导致理想的解，但常常能在合理的时间内得到令人满意的结果。

此外，虽然有些算法并不要求精心组织输入数据，但另一些算法的确依赖于精心设计的数据结构。对问题实例的数据进行恰当组织和重构，有助于设计和实现高效的算法。数据结构对于算法的设计常常至关重要。对于具有一定“数据结构”知识的读者，应不难理解这一点。

1.3.2 如何设计算法

使用计算机的问题求解策略主要指**算法设计策略**（algorithm design strategy）。一般来说，算法的设计是一项创造性活动，不可能完全自动化，但学习一些基本的算法设计策略是非常有用的。对于所求解的问题，只要符合某种算法设计策略的前提，便可以利用它设计出精致而有效的算法。算法设计技术（也称“策略”）是使用算法解题的一般性方法，可用于解决不同计算领域的多种问题。如果所求问题符合某种算法设计策略处理的问题特性，就可使用该算法设计策略设计算法、求解问题。例如，读者熟知的排序问题符合分治策略求解的问题特征，可以用分治法求解。然而，由于在使用分治策略解题时的思路不同，会得到不同的排序算法。在第 4 章中将看到，合并排序和快速排序都可视为由分治法产生的排序算法，但两者是不同的算法。

1.3.3 如何表示算法

算法所表示的计算过程需要以某种方式描述出来。算法可以使用自然语言描述，但自然语言不够严谨。在计算机应用的早期，算法主要用流程图描述。实践证明，流程图通常只适于描述简单算法，对于复杂算法，流程图也会十分复杂，难以建图和理解。伪代码是自然语言和程序设计语言的混合结构。它所描述的算法通常比自然语言精确，又比实际程序设计语言简洁。但对于伪代码，并没有形成一致的语法规则，需要事先约定。使用一种实际的程序设计语言描述算法，虽然有时会多一些细节，但有助于算法的精确描述。此外，用 C++语言描述的算法本身就是很好的 C/C++程序范例，对学生掌握算法思想和进行程序设计都是有益的。

在本书中，我们使用 C/C++语言描述算法。C/C++语言类型丰富、语句精练，既能描述算法所处理的数据结构，又能描述算法过程。同时，用 C/C++语言描述算法可使算法结构简洁明了，可读性好。

1.3.4 如何确认算法

如果一个算法对于所有合法的输入，都能在有限时间内输出预期的结果，那么此算法是正确的。确认一个算法是否正确的活动称为**算法确认**（algorithm validation）。算法确认的目的在于确认一个算法能否正确无误地工作。使用数学方法证明算法的正确性，称为**算法证明**（algorithm proof）。对于有些算法，正确性证明十分简单，但对于另一些算法，却可能十分困难。

证明算法正确性常用的方法是数学归纳法。对于程序 1-1 的求最大公约数的递归算法 RGcd，

可用数学归纳法证明如下：

设 m 和 n 是整数，$0 \leqslant m < n$。若 $m = 0$，则因为 $\gcd(0, n) = n$，故程序 RGcd 在 $m = 0$ 时返回 n 是正确的。归纳法假定，当 $0 \leqslant m < n < k$ 时，函数 RGcd(m, n)能在有限时间正确返回 m 和 n 的最大公约数，那么，当 $0 < m < n = k$ 时，考察函数 RGcd(m, n)，它将具有 RGcd($n\%m$, m)的值。这是因为 $0 \leqslant n\%m < m$ 且 $\gcd(m, n) = \gcd(n \bmod m, m)$（数论定理），故该值正是 m 和 n 的最大公约数。证毕。

若要表明算法是不正确的，只需给出能够导致算法不能正确处理的输入实例即可。

到目前为止，算法的正确性证明仍是一项很有挑战性的工作。在大多数情况下，人们通过程序测试和调试来排错。**程序测试**（program testing）是指对程序模块或程序总体，输入事先准备好的样本数据（称为**测试用例**，test case），检查该程序的输出，来发现程序存在的错误及判定程序是否满足其设计要求的一项积极活动。测试的目的是为了“发现错误”，而不是“证明程序正确”。程序经过测试暴露了错误，需要进一步诊断错误的准确位置，分析错误的原因，纠正错误。**调试**（debugging）是诊断和纠正错误的过程。

1.3.5 如何分析算法

算法的分析（algorithm analysis）活动是指对算法的执行时间和所需空间的估算。求解同一问题可以编写不同的算法，通过算法分析，可以比较两个算法效率的高低。对于算法所需的时间和空间的估算，一般不需要将算法写成程序在实际的计算机上运行。当然在算法写成程序后，便可使用样本数据，实际测量一个程序所消耗的时间和空间，这称为程序的**性能测量**（performance measurement）。

1.4 递归和归纳

递归是一个数学概念，也是一种有用的程序设计方法。在程序设计中，为了处理重复性计算，最常用的办法是组织迭代循环，除此之外还可以采用递归计算的办法。美国著名计算机科学家约翰·麦卡锡极力主张将递归引入 Algol 60 语言，该语言是后来的 Pascal、PL/1 和 C 语言的基础。他本人提出的表处理语言 Lisp 不仅允许函数递归，数据结构也是递归的。

递归和归纳关系紧密。归纳法证明是一种数学证明方法，可用于证明一个递归算法的正确性。在下一章中还将看到，归纳法在算法分析中也很有用。

1.4.1 递归

1．递归定义

定义一个新事物、新概念或新方法，一般要求在定义中只包含已经明确定义或证明的事物、概念或方法。然而递归定义却不然，**递归**（recursive）定义是一种直接或间接引用自身的定义方法。一个合法的递归定义包括两部分：**基础情况**（base case）和**递归部分**。基础情况以直接形式明确列举新事物的若干简单对象，递归部分给出由简单（或较简单）对象定义新对象的条件和方法。所以，只要简单或相对简单的对象已知，用它们构造的新对象是明确的，无二义性的。

例 1-1 斐波那契数列

说明递归定义的一个典型例子是**斐波那契**（Fibonacci）数列，它的定义可递归表示成：

$$\begin{cases} F_0 = 0, \quad F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \qquad n > 1 \end{cases} \tag{1-2}$$

根据这一定义，可以得到一个无穷数列 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …，称为斐波那契数列。斐波那契数列产生于 12 世纪，但直到 18 世纪才由 A. De. Moivre 提出了它的非递归定义式。从 12 世纪到 18 世纪期间，人们只能采用斐波那契数列的递归定义来计算。斐波那契数列的直接计算公式为：

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n) \qquad (1\text{-}3)$$

式中，$\phi = \frac{1}{2}(1+\sqrt{5}) = 1.618\,033\,98\cdots$，$\hat{\phi} = 1-\phi = \frac{1}{2}(1-\sqrt{5}) = -0.618\,033\,98\cdots$。

2．递归算法

当一个算法采用递归方式定义时便成为递归算法。一个递归算法是指直接或间接调用自身的算法。递归本质上也是一种循环的算法结构，它把“较复杂”的计算逐次归结为“较简单”情形的计算，直至归结到“最简单”情形的计算，并最终得到计算结果为止。

使用递归来解决问题，与使用一本大词典查询一个单词的情形是类似的。在词典中查一个单词时，首先得到对该单词的解释，如果在该单词的解释中包含不认识的单词，还需要继续查这些不认识的单词的词义，直到所有相关单词都已有明确解释为止。如果其中至少有一个单词在词典中没有解释或出现循环定义，那么这一过程是循环不确定和错误的。

许多问题可以采用递归方法来编写算法。一般来说，递归算法结构简洁而清晰，可以用归纳法证明其正确性，并易于算法分析。

根据斐波那契数列的递归定义，可以很自然地写出计算 F_n 的递归算法。为了便于在表达式中直接引用，可以把它设计成一个函数过程，见程序 1-4。

【程序 1-4】 求 F_n

```
long Fib( long n)
{
    if(n<=1) return n;
    else return Fib(n-2)+Fib(n-1);
}
```

函数 Fib(n)中又调用了函数 Fib(n−1)和 Fib(n−2)。这种在函数体内调用自己的做法称为**递归调用**，包含递归调用的函数称为**递归函数**（recursive function）。从实现方法上讲，递归调用与调用其他函数没有什么两样。设有一个函数 P，它调用函数 Q(T x)，P 被称为**调用函数**（calling function），而 Q 称为**被调函数**（called function）。在调用函数 P 中，使用 Q(a)来引起被调函数 Q 的执行，其中 a 称为**实在参数**（actual parameter），x 称为**形式参数**（formal parameter）。当被调函数是 P 本身时，P 是递归函数。有时，递归调用还可以是间接的。对于间接递归调用，在这里不做进一步讨论。编译程序利用系统栈实现函数的递归调用，系统栈是实现函数嵌套调用的基础。

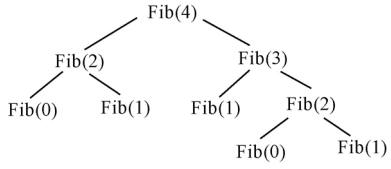

图 1-2　计算 Fib（4）的递归树

可以用所谓的**递归树**（recursive tree）来描述程序 1-4 的函数 Fib 执行时的调用关系。假定在主函数 main 中调用 Fib(4)，让我们来看 Fib(4)的执行过程。这一过程可以用图 1-2 所示的递归树描述。从图中可见，Fib(4)需要分别调用 Fib(2)和 Fib(3)，Fib(2)又分别调用 Fib(0)和 Fib(1)，……。其中，Fib(0)被调用了两次，Fib(1)被调用了三次，Fib(2)被调用了两次。可见，许多计算工作是重复的，当然这是费时的。

3．递归数据结构

在数据结构中，树、二叉树和列表常采用递归方式来定义。原则上，线性表、数组、字符串等也可以进行递归定义。但是习惯上，许多数据结构并不采用递归定义方式，而是直接定义。线性表、字符串和数组等数据结构的直接定义方式更自然、更直截了当。使用递归方式定义的数据结构称为**递归数据结构**（recursive data structure）。

1.4.2 递归算法示例

设计递归算法需要使用一种新的思维方式。递归概念较难掌握，本节的例子可以加深对递归算法的理解。

例 1-2 逆序输出正整数的各位数

设有正整数 n = 12345，现希望以各位数的逆序形式输出，即输出 54321。设 k 位正整数为 $d_1d_2\cdots d_k$，为了以逆序形式输出各位数 $d_kd_{k-1}\cdots d_1$，可以分成两步：

（1）首先输出末位数 d_k；

（2）然后输出由前 $k-1$ 位组成的正整数 $d_1d_2\cdots d_{k-1}$ 的逆序形式。

上面的步骤很容易写成程序 1-5 的递归算法。

【程序 1-5】 逆序输出正整数的各位数

```
#include<iostream.h>
void PrintDigit(unsigned int n)
{   //设 k 位正整数为 d1d2…dk, 按各位数的逆序 dkdk–1…d1 形式输出
    cout<<n%10;                                  //输出最后一位数 dk
    if(n>=10) PrintDigit(n/10);                  //以逆序输出前 k-1 位数
}
void main()
{
    unsigned int n;
    cin>>n;
    PrintDigit(n);
}
```

例 1-3 汉诺塔问题（tower of Hanoi）

假定有三个塔座：x，y，z，在塔座 x 上有 n 个直径大小各不相同的圆盘，它们按直径大小从小到大编号为 1，2，…，n。现要求将 x 塔座上 n 个圆盘移到塔座 y 上，并仍按同样顺序叠排，即初始状态如图 1-3（a）所示，最终状态如图 1-3（d）所示。圆盘移动时必须遵循下列规则：

（1）每次只能移动一个圆盘；

（2）圆盘可以加到塔座 x，y，z 中任意一个之上；

（3）任何时刻都不能将一个较大的圆盘放在较小的圆盘之上。

为了将圆盘全部从塔座 x 移到塔座 y，并且仍按原顺序叠排，一种朴素的想法是：如果能够将塔座 x 的上面 $n-1$ 个圆盘移至空闲的塔座 z 上，并且这 $n-1$ 个圆盘要求以原顺序叠排。这样，塔座 x 上就只剩下一个最大的圆盘，如图 1-3（b）所示。于是，便可以轻而易举地将最大圆盘放到塔座 y 上，如图 1-3（c）所示。余下的问题是如何将 $n-1$ 个圆盘从塔座 z 借助于空闲塔座 x 移到塔座 y 上。相对于原始问题而言，现在要解决的问题的性质与原问题相同，但被移动的圆盘数目

少了一个，是相对较小的问题。使用递归很容易写出求解此问题的算法。

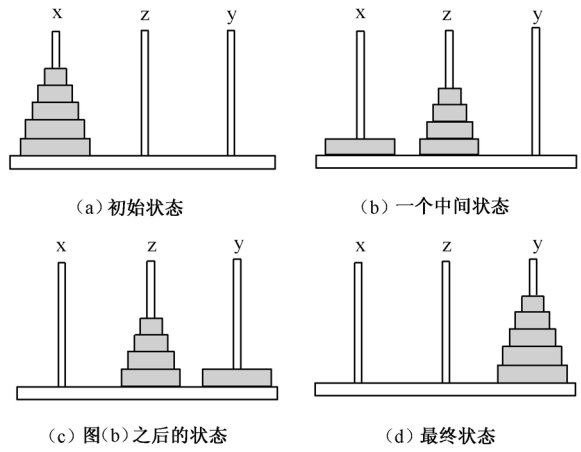

图 1-3　汉诺塔问题

假定圆盘从小到大编号为 1～n，移动圆盘的算法可以粗略描述如下：

（1）以塔座 y 为中介，将前 $n-1$ 个圆盘从塔座 x 移到塔座 z 上；

（2）将第 n 个圆盘移到塔座 y 上；

（3）以塔座 x 为中介，将塔座 z 上的 $n-1$ 个圆盘移到塔座 y 上。

注意，（1）和（3）求解的是移动 $n-1$ 个圆盘的汉诺塔问题，在程序 1-6 求解汉诺塔问题的模拟程序中，它们分别表现为一次递归函数调用。

【程序 1-6】　汉诺塔问题

```
#include <iostream.h>
enum tower { A='X', B='Y', C='Z'};
void Move(int n,tower x,tower y)
{ //将第 n 个圆盘从塔座 x 移到塔座 y 的顶部
    cout << "The disk "<<n<<" is moved from "
    << char(x) << " to top of tower " << char(y) << endl;
}
void   Hanoi(int n, tower x, tower y, tower z)
{ // 将塔座 x 上部的 n 个圆盘移到塔座 y 上，顺序不变
    if (n) {
        Hanoi(n-1, x, z, y);          //将前 n-1 个圆盘从塔座 x 移到塔座 z 上，塔座 y 为中介
        Move(n,x,y);                  //将第 n 个圆盘从塔座 x 移到塔座 y 上
        Hanoi(n-1, z, y, x);          //将塔座 z 上的 n-1 个圆盘移到塔座 y 上，塔座 x 为中介
    }
}
void main()
{
    Hanoi(4,A,B,C);                   //假定 n＝4
}
```

例 1-4　产生各种可能的排列

给定 n 个自然数{0, 1, …, $n-1$}的集合，设计一个算法，输出该集合所有可能的**排列**（permutation）。例如，集合{0, 1, 2}有 6 种可能的排列：（0, 1, 2）, (0, 2, 1), (1, 0, 2), (1, 2, 0), (2, 0, 1), (2, 1, 0)。容易看到，n 个自然数的集合有 $n!$ 个不同的排列。下面以 4 个自然数的集合{0, 1, 2, 3}为例，介绍一种求解此问题的简单递归算法。

由 4 个自然数组成的排列通过下列方式构造：

（1）以 0 开头，紧随其后为{1, 2, 3}的各种排列；

（2）以 1 开头，紧随其后为{0, 2, 3}的各种排列；

（3）以 2 开头，紧随其后为{0, 1, 3}的各种排列；

（4）以 3 开始，紧随其后为{0, 1, 2}的各种排列。

语句（1）中“紧随其后为{1, 2, 3}的各种排列”实质上是求解比原始问题少一个数的排列生成问题。相对原问题而言，这是一个同类子问题，但规模小一些。这也意味着可用递归算法求解这一问题。程序 1-7 描述这一算法，可用 Perm(a, 0, n)调用之。

【程序 1-7】　排列产生算法

```
template <class T>
void Perm(T a[], int k, int n)
{
    if (k==n-1){                                          //输出一种排列
        for (int i=0; i<n; i++)
            cout << a[i] << " "; cout << endl;
    }
    else                                                  //产生{a[k],…,a[n-1]}各种排列
        for (int i=k; i<n; i++) {
            T t=a[k]; a[k]=a[i]; a[i]=t;
            Perm(a, k+1, n);                              //产生{a[k+1],…,a[n-1]}各种排列
            t=a[k]; a[k]=a[i]; a[i]=t;
        }
}
```

1.4.3　归纳证明

证明一个定理不成立的最好方法是举一个反例。那么，如何证明一个程序是正确的？程序的正确性证明是一个非常困难的问题，一个完整的程序正确性证明过程常常比编写程序费时得多。两种最常见的证明方法是归纳法和反证法。下面我们采用**非形式证明**（informal proof）方式讨论算法的正确性问题。

先来看归纳法。对于无限对象集上的命题，归纳法往往是唯一可行的证明方法。递归数据结构的特性和递归算法问题常使用归纳法证明。在多数情况下，归纳法在自然数或正整数集合上进行，当归纳法应用于递归定义的数据结构（如树和表）时，称为**结构归纳法**（structural induction）。下面将看到，递归函数和归纳证明二者在结构上非常类似，这对于运用归纳法证明复杂的递归数据结构和算法命题是很有帮助的。

程序 1-5 和程序 1-6 分别是逆序打印正整数和汉诺塔问题的递归函数。对于给定的一些测试用例，它们都能正确工作，但并不意味着它们一定是正确的程序。程序正确性证明是非常有用的，

只有证明是正确的程序才能确认该程序对所有输入都能得到正确的结果。

使用归纳法进行证明的过程由两部分组成。

（1）**基础情况**（base case）确认被证明的结论在某种（某些）基础情况下是正确的。

（2）**归纳步骤**（induction step）这一步又可分成两个子步：首先进行归纳假设，假定当问题实例的规模小于某个量 k 时，结论成立；然后使用这个假设证明对问题规模为 k 的实例，结论也成立。至此，结论得证。

定理 1-1 对于 $n \geqslant 0$，程序 1-5 是正确的。

证明：（归纳法证明）

首先，当 n 是 1 位数时，程序显然是正确的，因为它仅执行了语句“cout<<n%10;”。

假定函数 PrintDigit 对所有位数小于 k（$k>1$）的正整数都能正确运行，当 n 的位数为 k 位时，此时有 $n \geqslant 10$，算法必定先执行语句“cout<<n%10;”然后执行语句“if(n>=10) PrintDigit(n/10);”。由于$\lfloor n/10 \rfloor$[①]是 n 的前 $k-1$ 位数字形成的数，归纳法假设函数调用 PrintDigit(n/10)能够将它正确地（并在有限步内）按数字的逆序输出，那么，现在先执行语句输出个位数字（n%10），然后由于按逆序输出前 $k-1$ 位数字的做法是能够正确按逆序输出全部 k 位数字的，所以程序 1-5 是正确的。本例中，归纳证明使用的量是十进制数的位数。

上述证明看起来非常类似于对一个递归算法的描述。这正是由于递归和归纳是密切相关的，它们有很多相似之处。二者都是由一个或多个基础情况来终止的。递归函数通过调用自身得到较小问题的解，并由较小问题的解来形成相对较大的问题的解。同样，归纳法证明依靠归纳法假设的事实来证明结论。因此，一个递归算法比较容易用归纳法证明其正确性。

同样地，不难运用归纳法证明程序 1-6 汉诺塔程序的正确性，我们将其留作练习。

在本书的算法证明和分析中，还常运用反证法。为了使用反证法证明一个结论，首先应假设这个结论是错误的，然后找出由这个假设导致的逻辑上的矛盾。如果引起矛盾的逻辑是正确的，则表明假设是错误的，所以原结论是正确的。下面举一个经典的例子说明反证法的运用。

定理 1-2 存在无穷多个素数。

证明：（反证法证明）

反面假设：假设定理不成立，则存在最大素数，记为 P。令 P_1，P_2，…，P_{k-1}，P 是从小到大依次排列的所有素数。设 $N = P_1P_2\cdots P_{k-1}P+1$，显然 $N>P$，根据假设 N 不是素数。但 P_1，P_2，…，P_{k-1}，P 都不能整除 N，都有余数为 1。这就产生矛盾，因为每一个整数，或者自己是素数，或者是素数的乘积。现在 N 不是任何素数的乘积，这也意味着不存在最大素数，定理得证。

本章小结

本章概述有关算法、问题、问题求解过程及算法问题求解方法等贯穿本书的一些重要概念和方法。算法可以看作求解问题的一类特殊的方法，它是精确定义的，能在有限时间内获得答案的一个求解过程。对算法的研究主要包括如何设计算法，如何表示算法，如何确认算法的正确性，如何分析一个算法的效率，以及如何测量程序的性能等方面。算法设计技术是问题求解的有效策略。算法的效率通过算法分析来确定。递归是强有力的算法结构。递归和归纳关联紧密。归纳法是证明递归算法正确性和进行算法分析的强有力工具。

① 符号$\lfloor x \rfloor$表示不大于 x 的最大整数，符号$\lceil x \rceil$表示不小于 x 的最小整数。

习题 1

1-1　什么是算法？它与计算过程和程序有什么区别？

1-2　程序证明和程序测试的目的各是什么？

1-3　用欧几里得算法求 31 415 和 14 142 的最大公约数。估算一下程序 1-2 的算法比程序 1-3 的算法快多少倍？

1-4　证明等式 gcd(m, n) = gcd(n mod m, m)，对每对正整数 m 和 n，$m>0$ 都成立。

1-5　解释名词：问题、问题求解、问题求解过程、软件生命周期。

1-6　简述匈牙利数学家乔治·波利亚在《如何求解》一书中提出的思想，如何体现在算法问题求解过程中。

1-7　算法研究主要有哪些方面？

1-8　请举出至少一个算法问题的例子，说明因为数据组织方式不同，导致解题效率有显著差异。

1-9　试给出 n！的递归定义式，并设计一个递归函数计算 n！。

1-10　使用归纳法，证明上题所设计的计算 n！的递归函数正确性。

1-11　请使用归纳法证明汉诺塔函数的正确性。

1-12　试用归纳法证明程序 1-7 的排列产生器算法的正确性。

1-13　写一个递归算法和一个迭代算法计算二项式系数：

$$C_n^m = C_{n-1}^m + C_{n-1}^{m-1} = n!/m!(n-m)!$$

1-14　给定一个字符串 s 和一个字符 x，编写递归算法实现下列功能：

（1）检查 x 是否在 s 中；

（2）计算 x 在 s 中出现的次数；

（3）删除 s 中所有 x。

1-15　写一个 C++函数求解下列问题：给定正整数 n，确定 n 是否是它所有因子之和。

1-16　S 是有 n 个元素的集合，S 的幂集是 S 所有可能的子集组成的集合。例如，S= {a, b, c}，则 S 的幂集= {(), (a), (b), (c), (a, b), (a, c), (b, c), (a, b, c)}。写一个 C++递归函数，以 S 为输入，输出 S 的幂集。