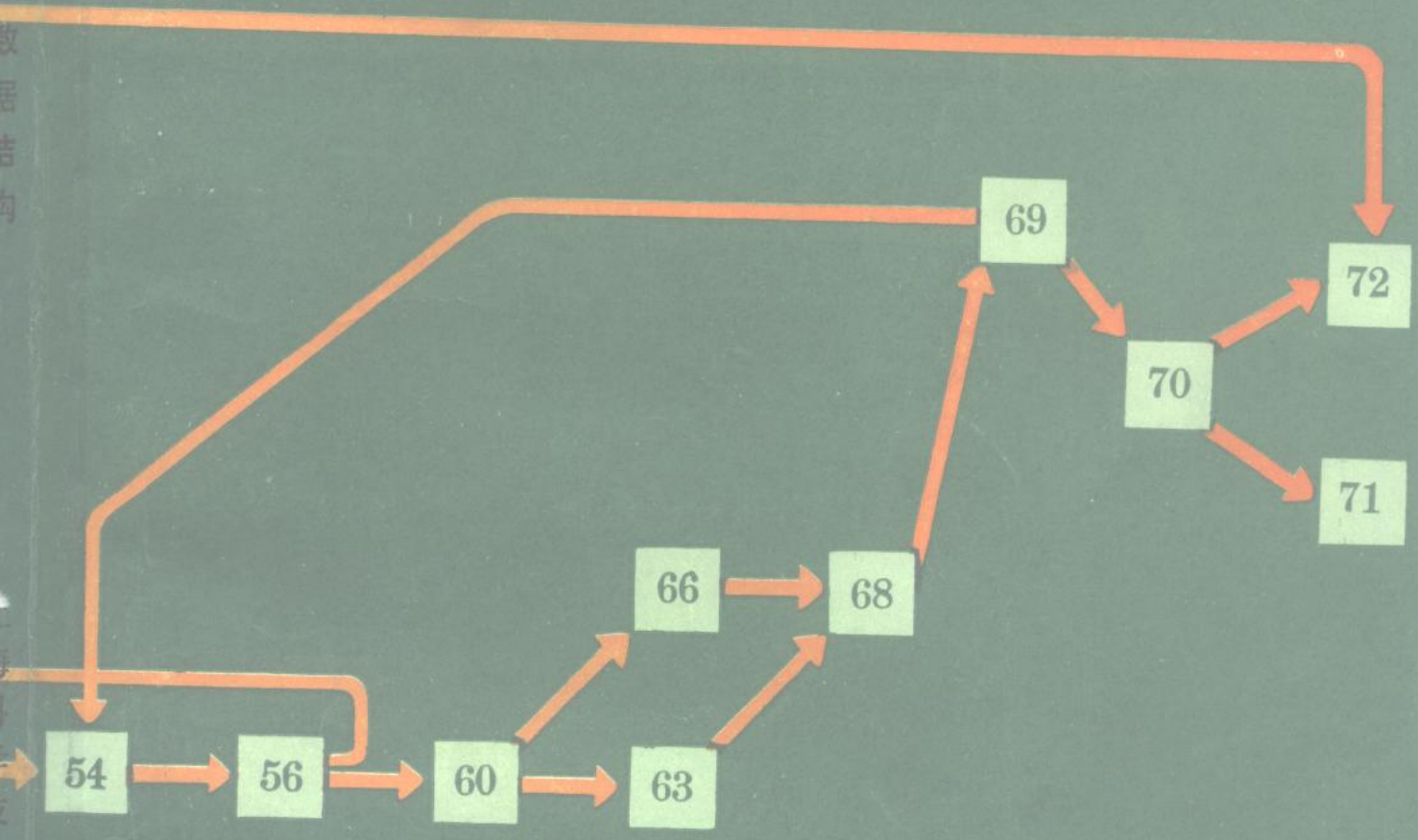
实用数据结构

霍义兴 夏 炬 汤宝骥 编



上海科学技术出版社

实用数据结构

霍义兴　夏　炬　汤宝骥　编

上海科学技术出版社

内 容 提 要

本书用大量的程序流程图、例题和精选的实例介绍了各种数据结构的基本概念和并列表、树、图的存贮方法和处理方法以及典型的应用。同时还简略介绍了算法的概念。全书共分五章：一、算法概述；二、字符串；三、并列表；四、树；五、复杂的数据结构。

本书的特点是内容选材精炼，图文并茂，直观生动，阐述深入浅出，循序渐进。章末附有习题，供读者练习。

本书可作为大专院校计算机专业教材，也可以供从事计算机科学方面工作的有关科技工作者及其它工程技术人员参考。

实用数据结构

霍义兴 夏 炬 汤宝骅 编

上海科学技术出版社出版

（上海瑞金二路 450 号）

新华书店上海发行所发行 上海商务印刷分厂印装

开本 787×1092 1/16 印张 16.5 字数 391,000

1987 年 2 月第 1 版 1987 年 2 月第 1 次印刷

印数 1—5,700

统一书号：13119·1397 定价：2.75 元

前　言

随着计算机科学的发展，电子计算机的应用已远远地超出了单纯进行数值计算的范围，数据处理已在计算机的应用中占有愈来愈重要的地位。利用计算机进行情报检索、企业管理、商业事务处理、人工智能等都是它应用于数据处理的实际领域。这样，如何有效地组织数据，以便设计出更有效的、高质量的程序来解决现实生活中的问题已成为计算机科学工作者十分关心的事情。有人曾将程序概括为：程序＝数据结构＋算法。因此，对于从事计算机科学工作及其应用方面的科技人员来说，掌握"数据结构"的有关概念是十分必要的。

数据结构实际上就是数据元素之间的组织关系，也有人称它为"信息结构"，然而有关数据结构的定义目前尚未能够统一。在本书中，我们把数据结构 $B=(K, R)$ 定义为一个二元组，其中 K 是数据元素的有限集合，而 R 是 K 上的关系的有限集合。本书着重阐述各种数据结构的基本概念及其实际应用，另外，还简要地介绍了算法概念，并对某些程序进行了适当地分析和比较。在介绍各种数据结构及其应用的有关程序实例时，除了极少数的实例用 PL/1 程序设计语言编写以外，主要采用程序流程图的方式。全书共分五章，第一章算法概述主要介绍了解决实际问题有关算法的设计思想，同时还扼要地阐述了如何分析一个算法的复杂性问题。第二章字符串是讨论非数值性数据字符串的处理问题。第三章并列表是一种常用的比较简单的结构，它在实际应用中有很大的使用价值。第四章树是极为重要的非线性的结构，它在数据处理中起着重要的作用。最后一章介绍较为复杂的数据结构，其中包括图、叶并列表及文件系统的组合查询等。

本书是在作者多年来讲授"数据结构"课程的经验的基础上编写而成。本书的初稿曾在上海交通大学等院校对四届学生进行讲授，效果良好。

限于作者水平，书中错误在所难免，望请读者批评、指正。

编者

1985 年 12 月

目　录

第一章　算法概述

1-1　算法及其复杂性

1．算法的概念

算法的研究是计算机科学的核心，而算法概念的本身是计算机程序设计中的最基本的概念之一，因此，在讲述本课程其它内容之前，首先对这个概念进行初步的分析。

一个算法就是一个有穷规则的集合，其中的规则规定了一个解决某一特定问题的运算序列。

例 1-1　欧几里德算法。给定两个正整数 m 和 n，求它们的最大公因子。

使用图 1-1 程序流程图可直观地看出它的运算过程。其中MOD(M, N)表示求两个数 M, N 的模。

存贮说明：

M, N——存放正整数的变量；

R——N 除 M 所得的余数。

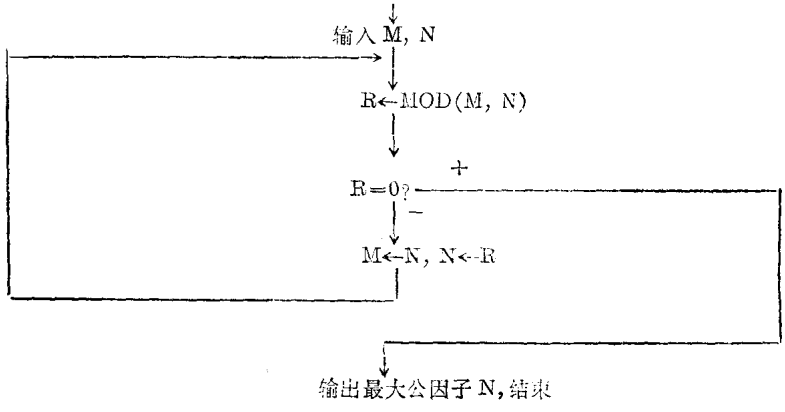


图 1-1　欧几里德算法程序流程图

图 1-1 的流程图实际上就是一个算法，它表示了求两个正整数 m 和 n 的最大公因子的一个运算序列。在例 1-1 中，如果 $m=36$, $n=24$ 执行上面流程图输出的 $N=12$，即为两个数的最大公因子，该算法到此结束。

一个算法应该具有以下五个重要特性：

(1) 有穷性　一个算法必须保证执行有限步之后结束。

(2) 确定性　算法的每一个步骤必须是确切定义的。

(3) 输入　一个算法有 0 个或多个输入，它就是在算法开始之前，对算法初始给出的量。

(4) 输出　一个算法有一个或多个输出，它是同输入有某种特定关系的量。

(5) 能行性　一般地讲，希望一个算法是能行的，就是说，它们原则上都是能够精确地

进行，而且人们用笔和纸做有穷次即可完成。

2．一个算法的分析

在计算机程序设计中，对算法进行分析是十分重要的，因为对于一个具体的应用实例，通常可能有若干个算法可以选用，我们很需要知道其中哪一个算法是最好的。

下面举例说明求极大值的算法。

例 1-2　给定 n 个元素 $x_1, x_2, \cdots, x_n$，我们要求出 m 和 j，使得 $m=x_j=\mathrm{MAX}\,x_k$；其中 $1\leqslant k\leqslant n$，并使其中的 j 尽可能地大。

我们仍用流程图直观地表示它的求解过程。为了便于对其中的每一步骤进行分析，在图 1-2 的流程图的左边标有步骤的编号。

存贮说明：

N——元素的个数；

$X(N)$——存贮 N 个元素的一维数组；

M——暂时存放极大值的变量；

J，K——辅助下标变量，J 指出已知元素序列中第 J 个元素为极大值。

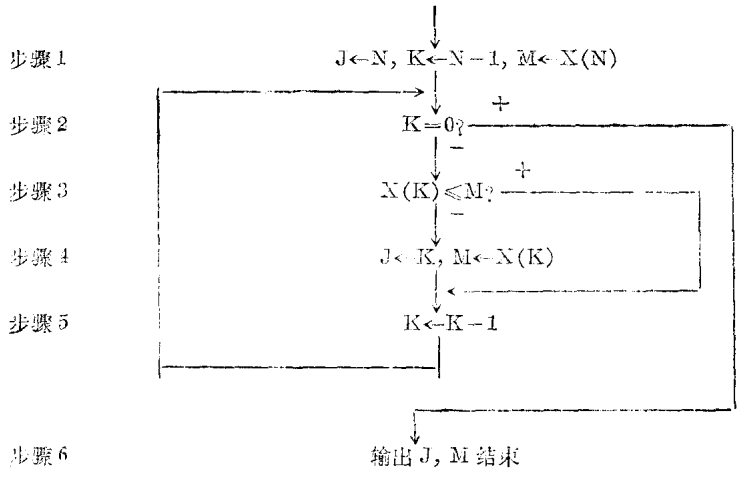


图 1-2　例 1-2 程序流程图

上述算法要求的存贮量是固定的，因此在这里仅仅分析执行所需要的时间，为此要计算执行每一步骤的次数，见表 1-1。

表 1-1　每一步骤执行次数

步　骤	次　数
1	1
2	n
3	$n-1$
4	A
5	$n-1$
6	1

从表 1-1 可以知道每一步骤执行的次数，这就给了我们为确定在一个具体的计算机上运行所花费时间的必要信息。

除了表中第 4 步指出的数量 A 外，其余各步骤执行的次数都是知道的，而数量 A 是改

变当前的极大值必须的次数。为了完成整个的分析，必须研究这个有趣的数量 A。

这种分析包括求 A 的极小值、极大值、平均值和标准离差。

在例 1-2 中，如果 $x_n=\mathrm{MAX}x_k$，其中 $1\leqslant k\leqslant n$，A 的极小值为 0。因为在步骤 1 中，$M\leftarrow X(N)$ 这个值以后一直没有被改变，这时 $A=0$。

在 $x_1>x_2>\cdots>x_n$ 的情况下，极大值 m 将改变 $n-1$ 次，所以 A 的极大值是 $n-1$，即 $A=n-1$。

显然，A 的平均值是在 0 和 $n-1$ 之间。

假定序列中的元素 x_k 的值是互不相同的，而且这些值的 $n!$ 种排列的每一种都是同等可能的。假设 $n=3$，那么有以下六种情况且其每种都是同等可能的。表 1-2 列出了每种排列情况，以及依据上述算法得到的 A 的值。

表 1-2 六种情况排列表

情 况	A 的 值
$x_3>x_2>x_1$	0
$x_2>x_3>x_1$	1
$x_3>x_1>x_2$	0
$x_1>x_3>x_2$	1
$x_2>x_1>x_3$	1
$x_1>x_2>x_3$	2

从上面可以看出 A 的平均值是 $(0+1+0+1+1+2)/6=5/6$。可见，A 的平均值并不依赖于具体元素 x_k 的精确值的大小，而仅仅涉及其相对的次序。

通常 A 的平均值定义为

$$A_n=\sum_k kP_{nk}\quad(0\leqslant k\leqslant n-1)$$

而 A 具有值 k 的概率 P_{nk} 将是

$$P_{nk}=(n\text{ 个对象的排列中满足 }A=k\text{ 的排列数})/n!$$

例如表 1-2 实例中，排列对象个数 $n=3$，可以求得 $P_{30}=1/3$。这是因为它满足 $A=0$ 的排列数是 2(即 $x_3>x_2>x_1$ 及 $x_3>x_1>x_2$ 这二排)。因此可以求得 $P_{30}=2/3!=1/3$。用同样方法，可以求得 $P_{31}=1/2$，$P_{32}=1/6$。

而在数学上可以进一步证明 $A_n=H_n-1$。

其中 $H_n=1+\dfrac{1}{2}+\cdots+\dfrac{1}{n}=\sum_{1\leqslant k\leqslant n}\dfrac{1}{k}$

最后，标准离差 σ_n 定义为 $\sqrt{V_n}$，V_n 称为方差，V_n 定义为 $(A-A_n)^2$ 的平均值，因此有

$$V_n=\sum_k(k-A_n)^2P_{nk}=\sum_k k^2P_{nk}-2A_n\sum_k kP_{nk}+A_n^2\sum_k P_{nk}$$

$$=\sum_k k^2P_{nk}-2A_nA_n+A_n^2=\sum_k k^2P_{nk}-A_n^2$$

将上例中，$n=3$，$0\leqslant k\leqslant 2$，$P_{30}=1/3$，$P_{31}=1/2$，$P_{32}=1/6$，而求得的平均值 $A_n=5/6$ 代入上式，可求得

$$V_n=\sum_k k^2P_{nk}-A_n^2=0^2\cdot P_{30}+1^2\cdot P_{31}+2^2P_{32}-A_n^2$$

$$=0^2\cdot\frac{1}{3}+1^2\cdot\frac{2}{3}+2^2\cdot\frac{1}{6}-\left(\frac{5}{6}\right)^2=\frac{1}{2}+\frac{2}{3}-\frac{25}{36}=\frac{17}{36}$$

这样标准离差 $\sigma_n=\sqrt{V_n}=\frac{\sqrt{17}}{6}$。

标准离差 σ_n 是一个定量指标，它指出预期的值可能接近平均值的程度，σ_n 越小，表示预期的值可能接近平均值的程度越大。

可以证明 $\sigma_n=\sqrt{H_n-H_n^{(2)}}$，其中 $H_n^{(2)}=1+\frac{1}{2^2}+\cdots+\frac{1}{n^2}=\sum_{1\le k\le n}\frac{1}{k^2}$

到此，我们简要地完成了对上述算法的分析，综合起来，已经得出关于数量 A 的统计：

$$A=(\min 0,\ \max n-1,\ \text{ave}\ H_{n-1},\ \text{dev}\ \sqrt{H_n-H_n^{(2)}})$$

其中 min, max, ave, dev 分别表示 A 的极小值，极大值，平均值及标准离差。

3．算法的时间和空间复杂性

我们常常用算法的计算时间和它所需的存贮空间来评价程序性能的好坏。性能评价可分为事先估计和事后测试两个主要阶段。这两个阶段是同等重要的。

首先考虑事先估计。假定在你的程序某处有下面语句

$$X\leftarrow X+Y$$

对此语句希望测定两个数，第一个数是语句执行一次所需的时间；第二个数是语句执行的次数，通常称为频数。这两个数的乘积就是该语句总的花费的时间。要准确地确定执行指令所需的时间是不可能的，除非我们拥有下列信息：

(1) 我们所用的计算机；

(2) 计算机的机器语言指令系统；

(3) 每一机器指令所需的时间；

(4) 把源程序翻译成机器语言的编译程序。

选用一台实际机器和现成的编译程序可能获得上述信息。另一个方法是定义一台假想的机器(具有想象的执行次数)，而所想象的执行次数应当合理地靠近现有的硬件，以使结果数字具有代表性。但这两种选择都没有多大实际意义。因为在这两种情况下，我们即使能定出确切次数，但也不能用到其他机器上。其次，编译程序也成问题，它是随机种而变的。再则，要取得可靠的计时数字也是困难的。因为有时钟、多道程序、分时等不同的环境限制。所有这些考虑，使我们做事先分析时，把目标仅限于集中研究语句的频数上，而把其他的问题放到做实验研究的时候去解决，并行处理问题也不予考虑。

图 1-3 示出了三个程序实例。

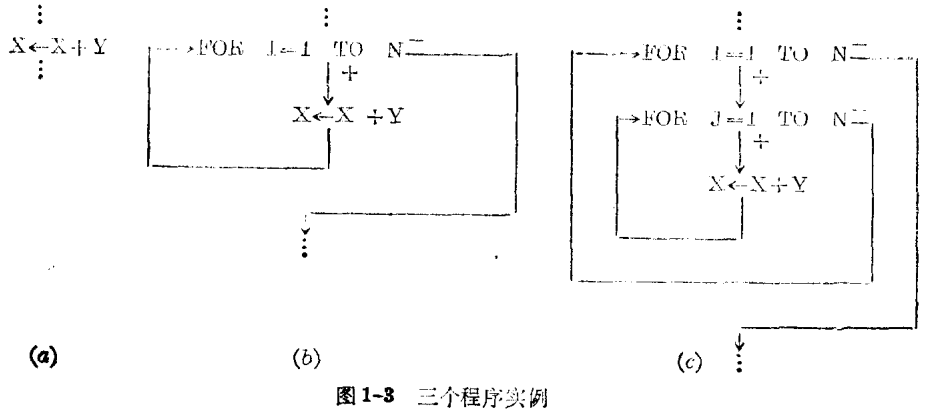


图 1-3 三个程序实例

在图(a)程序中，语句 $X\leftarrow X+Y$ 不包含在任何循环中，因此它的频数是 1。

在图(b)程序中，同样的语句将执行 n 次，而在图(c)程序中，则要执行 n^2 次(假设 $n\geqslant 1$)。

如果令 $n=10$，对于 1，n，n^2 三种不同情况，其数量的增加情况刚好是 1，10，100。而在进行具体分析时，主要关心的是确定算法的数量级，也就是说，去测定那些具有最大频数的语句。

为了测定数量级，经常会遇到下面的式子：

$$\sum_{1\leqslant i\leqslant n} 1,\quad \sum_{1\leqslant i\leqslant n} i,\quad \sum_{1\leqslant i\leqslant n} i^2$$

这三个式子的结果分别为 n，$n(n+1)/2$，$n(n+1)(2n+1)/6$。在图 1-3(c)程序中，语句 $X\leftarrow X+Y$ 执行

$$\sum_{1\leqslant i\leqslant n}\sum_{1\leqslant j\leqslant n} 1=\sum_{1\leqslant i\leqslant n} n=n^2 \text{ 次}$$

一般说来，有下面的公式

$$\sum_{1\leqslant i\leqslant n} i^k=\frac{n^{k+1}}{k+1}+\text{低价项}\quad k\geqslant 0$$

为了弄清这些概念，下面列举出一个计算第 n 项斐波那契(Fibonacci)数的简单程序。开始的斐波那契序列为

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ \cdots\cdots$$

其中每个新项是前面二项之和。如果我们称序列的第一项为 F_0，那么 $F_0=0$，$F_1=1$，而一般则有

$$F_n=F_{n-1}+F_{n-2}\quad (n\geqslant 2)$$

图 1-4 所示的流程图用来求取第 n 项斐波那契数 F_n，最后打印它的值。

存贮说明：

N——任意非负整数；

FN，FN1，FN2——存放斐波那契序列中的新项和前二项的变量；

I——循环变量。

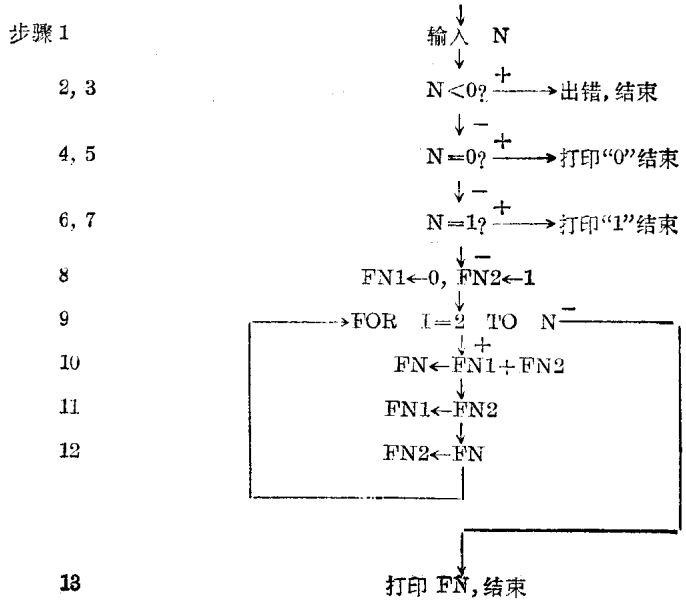


图 1-4 计算 F_n 程序流程图

我们首先分析 n 可能取什么值，N 的完全集合应包括这 $n<0$，$n=0$，$n=1$，$n>1$ 四种情况。表 1-3 列出了前面三种情况的频数。

表 1-3 各步骤语句执行频数

步骤	$n<0$	$n=0$	$n=1$
1	1	1	1
2	1	1	1
3	1	0	0
4	0	1	1
5	0	1	0
6	0	0	1
7	0	0	1
8～13	0	0	0

从表中可以看出，这三种情况都没有多大意义。因为它们没有一个使程序的作用充分发挥出来。最重要的是对 $n>1$ 的分析，此时 FOR 循环将真正地参加进去。步骤 1，2，4，6，8 只执行一次，而步骤 3，5，7 则完全不执行，步骤 8 的两条指令各执行一次。要注意 $n\geqslant 2$ 时，步骤 9 执行的次数是 n 而不是 $n-1$。虽然从 2 到 n 只有 $n-1$ 次，但最后一次要返回到步骤 9，在该处 i 增加到 $n+1$，这时测试得出 $i>n$，便转移到步骤 13，因此步骤 10，11，12 才是执行 $n-1$ 次。我们可以用表 1-4 来概括 $n>1$ 的分析。于是总的计数是 $4n+3$，它表示当 $n>1$ 时这个算法所需的时间(频数之和)。我们常常忽略 4 和 3 这两个常数，而简单表示为 $O(n)$。它称为该算法的时间复杂性。其含义是数量级与 n 成比例。$O(n)$ 是近代数学中广泛使用的一种渐近表示符号。

定义 1-1 如果存在两个常数 c 和 n_0，当 $n\geqslant n_0$ 时，$|f(n)|\leqslant c|g(n)|$ 成立，则 $f(n)=O(g(n))$。

$f(n)$ 通常用来表示某个算法的计算时间。当我们说一个算法的计算时间是 $O(g(n))$，其意思是指它所执行的次数不超过一个常数乘 $g(n)$。其中 n 是表示输入和输出特征的参量。例如 n 可以是输入的数目或输出的数目或者是它们的和，或者是它们之中某一个量值。

表 1-4 当 n>1 时各步骤语句执行频数

步骤	频数	步骤	频数
1	1	8	1
2	1	9	n
3	0	10	$n-1$
4	1	11	$n-1$
5	0	12	$n-1$
6	1	13	1
7	0		

在裴波那契程序中，n 表示输入的量值，该程序的时间复杂性可以写为 $T(n)=O(n)$，通常用 $T(n)$ 表示算法的时间复杂性。

我们写 $O(1)$ 就意味着计算时间是常数，$O(n)$ 称为线性的，$O(n^2)$ 称之为平方，$O(n^3)$ 称之为立方，$O(2^n)$ 称之为指数的。如果一个计算所用的时间是 $O(\log n)$*，则在 n 充分大时，它比 $O(n)$ 快，同样 $O(n\log n)$ 比 $O(n^2)$ 快，但没有 $O(n)$ 快。这七个计算次数 $O(1)$，$O(\log n)$，$O(n)$，$O(n\log n)$，$O(n^2)$，$O(n^3)$，和 $O(2^n)$ 在算法分析中是经常出现的。

也许有人会提出，由于现代数字计算机的进步，计算速度的增长将会削弱有效算法的重要性，然而情况恰恰相反。下面我们以表 1-5 的五个算法为例加以说明。

这里的时间复杂性是指处理一个大小为 n 的输入需要的时间(数量级)。假定一个时间单位(频数)等于 1 ms，算法 A_1 表示在 1 s 里可以处理一个大小为 1000 个的输入，而算法 A_5 表示在 1 s 里仅能处理一个大小至多为 9 个的输入。表 1-6 列出了在 1 s，1 min 和 1 h 内用这五个算法可能解决的问题的大小。

表 1-5 算法与时间复杂性

算法	时间复杂性
A_1	n
A_2	$n\log n$
A_3	n^2
A_4	n^3
A_5	2^n

表 1-6 五个不同算法解决问题大小的限制

算法	时间复杂性	最大问题的数量		
		1 s	1 min	1 h
A_1	n	1000	6×10^4	3.6×10^6
A_2	$n\log n$	140	4893	2.0×10^5
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

表 1-7 速度增加十倍后的情况

算法	时间复杂性	速度提高前解决问题的最大数量	速度提高后解决问题的最大数量
A_1	n	S_1	$10S_1$
A_2	$n\log n$	S_2	当 S_2 较大接近 $10S_2$
A_3	n^2	S_3	$3.16S_3$
A_4	n^3	S_4	$2.15S_4$
A_5	2^n	S_5	$S_5+3.3$

* 除非另有说明，所有的对数都是以 2 为底的。

假定下一代计算机比当代计算机速度快十倍，其解决问题情况可用表 1-7 来说明。从表中可见，算法 A_5 使解决问题的大小仅增加 3，而算法 A_3 能解决问题的大小大约增加 3 倍。

暂不考虑速度的增长，这时从表 1-6 中的 1min 一栏可以看出，用算法 A_3 替换 A_4 可以解决大六倍的问题。用算法 A_2 来替换 A_4 那可以解决大 125 倍的问题，而这些效果是比速度增加十倍更为满意。如果以 1 h 作为比较的基础，那差别就更加显著了。可见算法的时间复杂性的好坏是一个重要的性能标准。可以预见，随着计算机速度的增长，它将变得更为重要。

应指出的是，在上面的讨论中，算法的时间复杂性是忽略常数 c（即认为 $c=1$）这一个因素的。但是，若算法 A_1, A_2, A_3, A_4, A_5 的时间复杂性分别为 $1000n$, $100n\log n$, $10n^2$, n^3 和 2^n，则 A_5 对于 $2\leqslant n\leqslant 9$ 最好，A_3 对 $10\leqslant n\leqslant 58$ 最好，A_2 对 $59\leqslant n\leqslant 1024$ 最好，而 A_1 对于大于 1024 时最好。从上面的分析可以看出，当前计算机的程序是进行优化了的，因此应考虑到验证编制的程序是否对于非常大的问题或者对其执行次数较为频繁情况更为有效。在许多情况下，相对简单的算法已经足够，虽然可能花费一些时间用在额外性的事务性工作处理上。例如在上面的讨论中，如果算法 A_5 是十分简单的，那么当 n 较小时，算法 A_5 是可取的；而当 $n\geqslant 1024$ 时，算法 A_1 才显示它的优越性。

算法的另一个性能量度是空间复杂性 $S(n)$，它是依据某个算法在编成具体的程序后，在计算机中所占用的存贮单元总数，其中包括程序自身的长度以及它用的工作单元长度。我们常常可以用空间来换取时间，在下面几章中将会看到一些用更多的空间来得到一个较快的算法。

最后我们举一个综合性的且很有趣的数学问题来结束这一节。魔阵是一个由整数 1 到 n^2 所构成的 $n\times n$ 的矩阵，但它的每一行，每一列以及对角线的和都要相同。表 1-8 列出了 $n=5$ 的魔阵。

表 1-8 n=5 的魔阵

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

当 n 是奇数时，H. coxeter 给出了产生这个魔阵的简单规则：

“首先在第一行的中间写 1，然后按数的增加次序逐一往左上方的空格中填写。如果超出了魔阵，在上面就落到了最底层，在左边就平移到最右边，然后继续。如果方阵中已有数，那就移到下一格，然后再继续。”

表 1-8 的魔阵就是用这个规则构成的。图 1-5 列出了 n 为奇数时 $n\times n$ 魔阵的流程图。

存贮说明：

N——为任意奇数；

SQUARE(0: $N-1$, 0: $N-1$)——存放 $N\times N$ 个数的二维数组;
KEY, I, J, K, L——辅助变量。

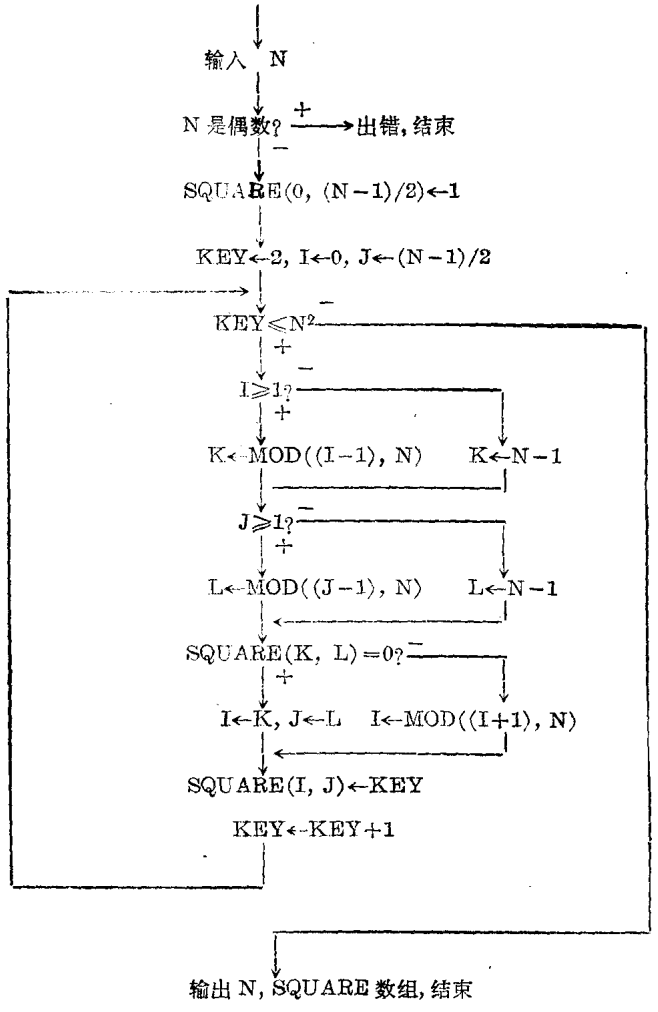


图 1-5 "魔阵"程序流程图

图中 KEY 是个整型变量,初值为 2,每次增加 1。这样在流程图相应的循环中的每一个语句执行次数不超过 n^2-1 次,因此执行流程图的时间复杂性是 $O(n^2)$。既然算法必须把数放到 n^2 个位置处,因此可以看出 $O(n^2)$ 是算法可能最好的界限。

1-2 算法设计的基本方法

为编写一个好的程序,了解并掌握一些基本数据结构知识是必不可少的,有关一些基本数据结构的概念在本书的后面几章中将进行较详细的介绍。对于算法设计人员来讲,为了获得一个既有效又优美的算法,必须了解一些基本的常用的算法的设计思想。下面我们简要介绍几种算法设计的基本方法。

1. 分治法

解一个较为复杂的问题时,尽可能把这个问题分为较小的部分,找出各部分的解,然后再把各部分的解组合成整个问题的解,这就是分治法。

下面先用十进制的乘法计算过程说明这种方法。

设 x 和 y 是两个 n 位十进制数，为简单起见，我们假定 n 是 2 的幂。我们把 x 和 y 各分成两半，每一半作为一个 $(n/2)$ 位数来处理。这样，我们可以把这个乘积表示如下：

$$\begin{aligned} xy &= (a\times 10^{n/2}+b)(c\times 10^{n/2}+d) \\ &= ac\times 10^{n}+(ad+bc)\times 10^{n/2}+bd \end{aligned} \tag{1-1}$$

$x=$ | a | b |

$y=$ | c | d |

等式(1-1)通过 4 次 $(n/2)$ 位数乘法以及一些加法和移位运算来计算 x 和 y 的乘积。x 和 y 的积也可以用下面的方法来计算：

$$\begin{aligned} U &= (a+b)(c+d) \\ V &= ac \\ W &= bd \end{aligned}$$

那么积 $xy=V\times 10^{n}+(U-V-W)\times 10^{n/2}+W$。很显然，上述方法很容易改编成相应的程序。

这方案仅需三次 $(n/2)$ 位数乘法以及一些加法和移位运算即得出 n 位数相乘的结果。例如：

$$\begin{aligned} x&=3141 \quad a=31 \quad c=59 \\ y&=5927 \quad b=41 \quad d=27 \\ &\qquad a+b=72 \quad c+d=86 \end{aligned}$$

则

$$\begin{aligned} U &= (a+b)(c+d)=72\times 86=6192 \\ V &= ac=31\times 59=1829 \\ W &= bd=41\times 27=1107 \\ xy &= V\times 10^{4}+(U-V-W)\times 10^{2}+W \\ &= 18290000+(6192-1829-1107)\times 10^{2}+1107 \\ &= 18616707 \end{aligned}$$

同理，若 x 和 y 是两个 n 位二进制数，假定 n 是 2 的幂，我们把 x 和 y 同样各分成二半，把每一半作为一个 $(n/2)$ 位数来处理，我们则可以把这个积表示为

$$\begin{aligned} xy &= (a\times 2^{n/2}+b)(c\times 2^{n/2}+d) \\ &= ac\times 2^{n}+(ad+bc)\times 2^{n/2}+bd \\ &= ac\times 2^{n}+[(a+b)(c+d)-ac-bd]\times 2^{n/2}+bd \end{aligned}$$

根据上面的讨论，两个 n 位二进制数相乘的时间复杂性的界是：

$$T(n)=\begin{cases} k & \text{对于 } n=1 \\ 3T(n/2)+kn & \text{对于 } n>1 \end{cases} \tag{1-2}$$

式中的 k 是在表达式计算中反映加法和移位运算的一个常数。上式是一个递归方程，它的解可以证明为

$$T(n)=3kn^{\log 3}-2kn$$

其解的过程如下：

假定 n 是 2 的幂，即 $n=2^{r}$。

$$T(n)=3T\left(\frac{n}{2}\right)+kn=3\left(3T\left(\frac{n}{4}\right)+\frac{1}{2}kn\right)+kn$$

$$=3^2T\left(\frac{n}{4}\right)+\frac{3}{2}kn+kn$$

$$=3^2\left(3T\left(\frac{n}{2^3}\right)+\frac{1}{2^2}kn\right)+\frac{3}{2}kn+kn$$

$$=3^rT\left(\frac{n}{2^r}\right)+\frac{3^{r-1}}{2^{r-1}}kn+\cdots+\frac{3^2}{2^2}kn+\frac{3}{2}kn+kn$$

$$=3^rk+\frac{3^{r-1}}{2^{r-1}}kn+\cdots+\frac{3^2}{2^2}kn+\frac{3}{2}kn+kn$$

$$=\frac{\left(\frac{3}{2}\right)^{r+1}-1}{\frac{3}{2}-1}kn=2\left(\frac{3}{2}\right)^{r+1}kn-2kn$$

$$=3\left(\frac{3}{2}\right)^r kn-2kn=3\cdot\frac{3^r}{2^r}kn-2kn$$

$$=3\cdot3^rk-2kn=3\cdot3^{\log n}k-2kn$$

$$=3n^{\log 3}k-2kn$$

我们也可以用归纳法给以证明：

$$n=1 \quad T(1)=3kn^{\log 3}-2kn=k$$

如果 $n=m \quad T(n)=3kn^{\log 3}-2kn$ 成立，那么

$$T(2m)=3T(m)+2km=3[3km^{\log 3}-2km]+2km$$

$$=3k(2m)^{\log 3}-2k(2m)$$

因而得证。

通过上面的分析可以知道：若考虑两个 n 位二进制数相乘，用传统的方法需要 $O(n^2)$ 次二进制"位运算"，而采用上面所讨论的方法需要的是 $n^{\log 3}$ 或者近似是 $n^{1.59}$ 阶的二进制位运算。

为了使乘法算法更加完整，我们必须考虑到 $a+b$ 和 $c+d$ 可能是 $(n/2+1)$ 的二进制位数，这样就不能直接把这个算法递归应用于大小为 $n/2$ 的问题来计算 $(a+b)(c+d)$ 的积。而应将 $a+b$ 改为 $a_12^{n/2}+b_1$，这里 a_1 是 $a+b$ 之和的头一位，而 b_1 是其余的位数。类似地将 $c+d$ 写作 $c_12^{n/2}+d_1$。这样乘积 $(a+b)(c+d)$ 可以表示为

$$a_1c_12^n+(a_1d_1+b_1c_1)2^{n/2}+b_1d_1 \tag{1-3}$$

式中第三项 b_1d_1 可以在大小为 $n/2$ 的问题上，应用这个乘法算法来计算，而在(1-3)式中，它的乘法可以在 $O(n)$ 内计算出来。

下面我们再举另一个例子来说明这种方法。如果我们要找出集合 S 中 n 个元素的极大元和极小元，对于这样一个问题，为了简单起见，假定 n 是 2 的幂。寻找 n 个元素中的极大元和极小元的一种明显的方法是分别逐个地比较查找，例如图 1-6 的流程图就是在集合 S 的元素(假设 S 的元素已事先存贮好)间进行 $n-1$ 次比较找出 S 的极大元。

存贮说明：

N——元素的个数；

$S(N)$——存放 N 个元素的数组；