张益新　沈　雁　编著

算法引论

国防科技大学出版社

算 法 引 论

张益新　沈雁　编著

国防科技大学出版社

内 容 提 要

本书介绍了高效率非数值算法的设计方法，引入了时间、空间复杂度的概念，讨论了递归方程的解法，介绍了评价算法的准则；着重讨论了动态规划法、回溯法、贪婪法、分而治之法、分枝界限法、局部搜索法六种常用的算法设计技术；还介绍了排列、组合算法，外排序算法及广义 FIBONACCI 数。还用一章的篇幅讨论了将递归算法改写成非递归算法的技术。最后一章介绍 NP 完全理论方面的结果，作为进一步研究算法理论的基础。

本书可供计算机科研及计算机技术人员参考。

算 法 引 论

张益新 沈雁 编著

责任编辑 张建军

*

国防科技大学出版社出版发行

（长沙市砚瓦池正街 47 号）

邮编：410073 电话(0731)4436564

新华书店总店科技发行所经销

国防科技大学印刷厂印装

*

开本：787×1092 1/16 印张：8.75 字数：202 千

1995 年 8 月第 1 版第 1 次印刷 印数：3000 册

ISBN 7—81024—335—7

TP · 64 定价：12.00 元

前　言

有不少人(包括一些计算机专业的学生)认为计算机软件(计算机科学)仅仅是编程序,调程序而已,其实不然。计算机科学虽然年轻,但它和数学、物理、化学、生物等历史悠久的学科一样有自身的规律和理论。它还有一些尚未解决的"难题",如集合 P 是否等于集合 NP 就是一个典型例子。类似这样的难题长期以来困扰着计算机科学家,有可能还会困拢很久。除了软件工程学,编译原理,操作系统,数据库原理外,算法分析与设计也是区别专业软件工作者与其它人的一门重要课程。

计算机科学的一个核心问题是算法理论。通过对常用的,有代表性的算法的研究,理解并掌握算法设计的技术,掌握分析算法复杂度的能力,掌握算法评判的准则是算法分析设计课程的目的。本书以此为宗旨,通过研究与分析一些有代表性的算法以期读者通过对这些研究与分析过程的了解达到掌握与运用的目的。

全书分五章,第一章引入复杂度概念,讨论了算法复杂度的计算方法,递归方程的解法及评价算法的准则。第二章介绍动态规划法、回溯法、贪婪法、分而治之法、分枝界限法、局部搜索法六种有代表性的算法设计技术。第三章介绍了无遗漏,不重复产生排列,组合的算法,外排序算法及广义 FIBONACCI 数。第四章介绍了如何将递归定义的问题用非递归算法解决。由于汇编语言及某些高级语言(如 FORTRAN)不允许递归调用,故这一章讨论的问题有很大的实际意义。最后一章讨论 NP 理论,介绍了"是否"问题,图灵机,集合 P,集合 NP 完全问题,介绍了库克定理,还介绍了证明一个问题是 NP 完全问题的方法。

作者

一九九四年十二月

目　录

第一章　复杂度及其分析

早在电子计算机出现之前，对算法的研究就进行了多年，如欧几里德算法给出两数的最大公因子、孙子算法给出最小公倍数。本世纪40年代以来，由于电子计算机的出现，使许多原来难以处理的问题得以解决。这更进一步推动了对算法的研究，算法至今还是一个非常活跃的研究领域。

对于实际的问题要寻求并设计出求解的算法，不仅如此，还更应当分析算法的品性。一个问题往往有多个算法，本章所要讨论的则是判定这些解决同一问题算法优劣的标准。

§1.1　算法的复杂度

算法是一个出现频繁的术语。准确地说，问题的算法由有限个指令的序列组成。其中每条指令都有明确的含义，每条指令的执行包含着有限的工作量；序列的执行会在有限时间内停止下来，并给出对问题实例的解答。

问题实例与问题的差别在于：问题是一个一般化的概念，例如排序问题，哈密尔顿回路问题等，而算法是为问题而建立的。排序问题就有各种算法，如气泡排序算法、选择排序算法、快速排序算法、归并排序算法等。虽然算法是为问题而建立，但算法不能为一般的问题而运行或执行，算法的运行只能针对该问题的一个实际情况（称为实例）。如排序算法运行时必须要知道有多少个数被排序，其中每一个数的值又是什么。

由上述算法的定义可以看到，算法有一个特性即有穷性（或称有限性），算法定义不仅要求序列中的指令是有限的，不仅要求每条指令的执行只包含有限的工作量，更重要的是要求算法的整个指令序列的执行会在有限时间内结束。

一个算法的运行所需时间的长短、空间的多少具有极为重要的意义。本书不仅讨论算法设计的技巧，还讨论算法对时间、空间的需求。

对于算法有两方面的要求：

(1)要求算法易于理解、易于编程（在计算机上实现）、易于调试。

(2)要求算法高效，节省时间与空间。

但是，这两方面的要求相互矛盾。一个易于理解、编程、调试的算法往往效率很低。反之，一个高效率算法的思路往往不易理解，不易于编程、调试。具体选择算法时应两方面兼顾，综合考虑。如果需要一个反复多次执行的程序，则要将算法的高效性放在易理解、易编程性的前面。虽然在理解、编程、调试上要多耗时间，但可以得到高效算法，且因多次运行，节省的时间将是大量的。反之，如需要一个仅运行一两次的程序，则不必很讲究算法的高效率，选择一个易编、易调的算法即可。

以排序问题为例，气泡法最直观，由此写出的程序最短，但这样编出的程序运行时所要的时间最长。快速排序法所需时间比气泡法少，但其思路不那么直观易懂。堆排序法更快，但思路更不易懂。

算法(或由此写成的程序)运行所需时间与两个因素有关：

(1)问题实例的大小。例如给 1000 个数排序一般比给 1000000 个数排序要快得多。

(2)即使问题实例的大小相同，算法运行的时间还与实例的具体情况有关。例如给 1000 个几乎有序的数排序比给 1000 个杂乱无章的数排序要快。

如用 n 表示问题实例的大小，则算法运行所需时间不仅与 n 有关(即不仅仅是 n 的函数)。下面用 x 和 $|x|$ 分别表示实例和实例的大小。从而有如下定义。

算法 A 的时间复杂度用 $T_A(n)$ 表示，定义为

$$T_A(n)=\max\left\{m \left| \begin{array}{l}\text{存在实例 } x,|x|=n,\text{且算法 A}\\ \text{对实例 x 运行所需时间为 m}\end{array}\right.\right\}$$

由以上定义可知，时间复杂度 $T_A(n)$ 就是算法 A 对大小为 n 的所有实例运行所需时间的最大者(即大小为 n 的实例的最坏情况所需的时间)。显然，时间复杂度是 n 的函数。

同样地，可以定义算法 A 的空间复杂度 $U_A(n)$ 为

$$U_A(n)=\max\left\{s \left| \begin{array}{l}\text{存在实例 } x,|x|=n,\text{且算法 A}\\ \text{对实例 x 运行所需内存单元数为 s}\end{array}\right.\right\}$$

由于超大规模集成电路技术的发展，使得计算机可能含有数千个甚至更多的处理器。这种含有不止一个处理器的计算机称为并行计算机。因此，一个问题的计算工作量可由多个处理器分担、并行进行。如算法 A 是在并行机上运行的，则要考虑 A 对处理器数目的需求。因此算法 A 的处理器复杂度 $V_A(n)$ 定义为

$$V_A(n)=\max\left\{p \left| \begin{array}{l}\text{存在实例 } x,|x|=n,\text{且算法 A}\\ \text{对实例 x 运行所需处理器数为 p}\end{array}\right.\right\}$$

本书主要讨论时间复杂度。下文中复杂度即指时间复杂度。

设问题 P 有两个算法 A_1 和 A_3，其复杂度分别为 $1000n$ 和 2^n，单位为毫秒(ms)。当实例大小 n 为 10，A_1 需要 10 秒，A_3 要 1 秒；但当 n 为 60 时。A_1 要 60 秒，A_3 要 10^{15} 秒，合数千万年。详见表 1.

表 1　不同复杂度的算法对时间的需求表

复杂度 (ms)	实例大小一定时所需的时间(ms)				
	2	10	60	1024	2048
$1000n$	2000	10^4	6.10^4	10^6	2.10^6
$10n^2$	40	1000	$3.6\cdot10^4$	10^7	4.10^7
2^n	4	10^3	10^{18}	10^{300}	10^{600}

不同复杂度的算法在一定时间内对问题实例大小的限制见表 2.

表 2　复杂度对问题实例大小的限制表

复杂度 (ms)	规定时间内可处理的最大实例		
	1 秒	1 分	1 小时
1000n	1	60	3600
$10n^2$	10	77	600
2^n	10	16	22

由表 2 可以看出，算法 A_1 在 1 秒内可处理的最大实例的大小为 1，在 1 小时内可以处理的实例大小为 3600. 然而当时间从 1 秒增加到 1 小时，算法 A_3 可处理的实例大小仅从 10 增大为 22。

现考虑把一个算法移植到另一个运行速度更快的计算机上。若新计算机速度为原计算机速度的十倍，则在同样时间内新的机器可完成的工作是原机器的十倍，因此在同样时间内可处理更大的问题实例。在新机器上不同的算法对问题实例大小的限制是不同的，详见表 3。由表 3 可见复杂度为 1000n 的算法 A_1 可处理的问题实例大小增长为十倍，复杂度为 $10n^2$ 的算法可处理的问题实例大小增长为 3.16 倍，然而复杂度为 2^n 的算法 A_3 可处理的问题实例大小的增长不是成倍的，其增长为一个固定值 3。换言之，如原可以处理的实例的大小为 1000，在新机器上（其速度增长了十倍）可处理的实例大小仅为 1003，这样投资买新机器就很不经济了。

表 3　不同计算机对问题实例大小的限制

复杂度	原机器可处理的问题实例大小	新机器可处理的问题实例大小
1000n	S	10S
$10n^2$		3.16S
2^n		S+3

从上述的三张表可以看到算法的复杂度意义重大。

［例 1］　问题 P 有算法 A_1，复杂度 $T_1(n)=n^3$（毫秒），现将算法改善为 A_2，复杂度 $T_2(n)=n^2$（毫秒），则原先要 1 小时运行时间的问题实例现在需要运行时间多少？

实例的大小为 n，则

$$n^3=3600\cdot 1000$$

因此

$$n=153$$

用算法 A_2 则需时间

$$n^2=153^2=23489\text{ 毫秒}$$
$$=23\text{ 秒}$$

如原来算法的复杂度 $T_1(n)=2^n$，则

$$2^n=3600\cdot 1000$$

问题实例大小

$$n=\log_2(3600\cdot 1000)$$
$$=22$$

用算法 A_2 则需时间

$$n^2=22^2=484\text{ 毫秒}$$
$$=0.4\text{ 秒}$$

由此可见改进算法复杂度意义重大。

算法的复杂度往往是一个很复杂的表达式，有时为了表示复杂度随 n(问题实例的大小)而增长的速率，常用到“大 O”和“大 Ω”记号。虽然用大 O 和大 Ω 记号将复杂度表达式简化了，但最本质的东西被保留了。因此可用来表示算法复杂度的数量级。

定义 1 两个非负函数 T(n)和 f(n)定义在非负整数域上，如存在正数 C 及正整数 N，使

$$T(n)\leqq C\cdot f(n)\ (n\geqq N)$$

则称 T(n)的阶小于或等于 f(n)的阶，记作

$$T(n)=O(f(n))$$

显而易见 $C\cdot f(n)$是 T(n)的上界。

显然，按上述定义有

$$3n^2+500=O(3n^2)$$
$$3n^2+4n+500=O(3n^2)$$

可见 T(n)如用大 O 记号，则 T(n)中的低次项可忽略，又

$$3n^2=O(n^2)$$
$$4n^3=O(n^3)$$

可见，如 T(n)是个多项式，用大 O 记号时，只与 T(n)的最高项次数有关。

[**例 2**] 用大 O 记号表示 $2n^3+3n^2$，因为

$$2n^3+3n^2\leqq 5\cdot n^3(n\geqq 1)$$

所以

$$2n^3+3n^2=O(n^3)$$

此时 C=5 N=1

容易证明，如果

$$T(n)=O(n^k)$$

则

$$T(n)=O(n^j)\ (j>k)$$

因此 n^k 和 n^j 都是 T(n)的上界，所以在用大 O 记号表示函数 T(n)时，要选择尽量低的幂。

定义 2 两个非负函数 T(n)和 g(n)定义在非负整数上，如存在正数 C，使

$$T(n)\geqq C\cdot g(n)$$

对无穷多的 n 取值成立。则记为

$$T(n)=\Omega(g(n))$$

要注意的是，g(n)并不是 T(n)的下界。

[例 3]

$$T(n)=\begin{cases}n, & n\text{ 为奇数}\\ n^2/100, & n\text{ 为偶数}\end{cases}$$

显然有

$$T(n)=\Omega(n^2)$$

只要令 $C=\frac{1}{100}$ 但是 $g(n)=n^2$ 不是 T(n)的下界。

容易证明，如果

$$T(n)=\Omega(n^K)$$

则

$$T(n)=\Omega(n^j)\ (j<K)$$

因此在用大 Ω 记号表示函数 T(n)时，要选择尽量高次的幂。

[例 4] $T(n)=n^3-2n^2$，则令 $C=1/3$，$g(n)=n^3$，有

$$n^3-2n^2\geqq\frac{1}{3}\cdot n^3(n\geqq 3)$$

对无穷多的 n 值成立，因此

$$n^3-2n^2=\Omega(n^3)$$

[例 5] 考虑例 2 中的函数，有

$$2n^3+3n^2=\Omega(n^3)$$

§1.2 算法分析

分析、评价算法时应从其正确性、复杂度、简单性、最优性、可读性及可修改可扩展性等方面加以考虑。

1. *正确性* 正确性应作为分析算法的标准之一是不言而喻的。一个不正确的算法无论有什么样的“优点”，永远不会为人们所采用。一个算法是正确的是指：在给定有效的输入数据后，算法经过有穷时间的计算能给出正确的答案。算法的正确性是可以证明的，但稍复杂的算法的正确性的证明就是一件极为耗时的工作。数学归纳法常被用来证明算法的正确性。为了证明算法的正确性，必须对输入的有效性及输出的正确性作严格的定义。为证明一个大型程序的正确性，可以试图将这个程序分解成一些较小的程序段，通过证明所有这些较小的程序段是正确的来证明整个程序是正确的。

2. *复杂度* 时间复杂度。显然算法的时间复杂度取决于算法运行时执行的工作量。因此如何度量算法的工作量就是很重要的了。这种度量要便于比较同一个问题的不同算法。

用算法运行所需时间作度量是不方便的。因为该时间随计算机不同而变化；用算法执

行的指令数或语句数目作度量也不方便，因为这取决于所用的语言。理想的度量方法不仅应与具体的计算机及使用的程序语言无关，而且应能告诉算法的效率。并且，我们还希望这种度量方法既精确又通用。

为了分析一个算法的复杂度，可以选取一种或几种基本运算，这个算法运行时只计基本运算的执行次数。当然不同的问题应选取不同的基本运算。例如，排序问题、搜寻问题的基本运算是比较，而矩阵乘法问题的基本运算应是实数乘法和实数加法。

如果一个问题的基本运算是乘法和加法，则降低这个算法复杂度的主要途径是减少乘法，因为乘法所需时间比加法多得多，致力于减少加法收到的效果小；不仅如此，在减少加法时，可能会增加乘法，结果适得其反，复杂度反而增大了。但是，在减少乘法次数时，有时不得不增加加法次数，可是这仍然可能改善复杂度。

只要正确地选择基本运算，算法的基本运算执行次数就表示了算法工作量，代表了算法的复杂度，同时很容易比较同一问题的几种不同算法的优劣。显而易见，如果选择的基本运算越多，则对工作量的度量越精确（但分析所需的工作量也就越大），因此可以通过选择基本运算集合来满足对分析精度的要求。

在衡量算法的复杂度时，需要对问题实例的大小进行量化。当然，不同的问题的量化方法是不一样的。例如，排序问题实例的大小是要排序的值的个数，搜索问题实例的大小是要搜索的表中的值的个数，线性方程组实例的大小可以是未知数的个数，矩阵乘法问题实例的大小是两个矩阵的行数、列数，图问题实例的大小是图的节点数和边的条数。当问题实例大小固定时，算法的工作量仍不确定，工作量还与输入的具体情况有关。从而一种可行的解决办法是分别计算算法的平均工作量和最坏情况工作量。

计算平均工作量，也就是计算大小为 n 的所有问题实例的工作量的平均值。设 B_n 是大小为 n 的问题实例的集合，I 是一个实例，且其大小为 n，即 $I \in B_n$，P(I) 是实例 I 出现的概率，t(I) 是算法对实例 I 运行时所需要的基本运算的执行次数，则平均工作量

$$A(n)=\sum_{I\in B_n} P(I)\cdot t(I)$$

但 P(I) 不是很容易确定的，因此平均工作量 A(n) 不易计算。

最坏情况工作量 T(n) 定义为

$$T(n)=\max_{I\in B_n} t(I)$$

由此定义可见，T(n) 就是我们定义的算法复杂度。T(n) 的计算比 A(n) 容易得多，并且还很有用，因为 T(n) 给出了算法工作量的上界。

[**例 6**] 考虑搜索问题，设要搜索的值 X 存在于被搜索的表中。如使用顺序搜索算法，则这种算法的平均工作量

$$A(n)=\frac{n+1}{2}$$

最坏情况工作量为

$$T(n)=n$$

但对有些问题而言，没有平均情况与最坏情况之差别。

[例 7] 考虑矩阵乘法问题，只要输入大小确定，一种算法运行时的平均工作量与最坏情况工作量是一样的，与输入的具体情况无关。

空间复杂度与时间复杂度相似。内存空间用来存放程序、常量、变量、输入数据、结果及中间结果。对算法进行分析就可以知道需要多少内存单元。所需内存空间大小不仅决定于问题实例的大小，还决定于特定的输入情况，因此对空间复杂度也要分别考虑平均情况和最坏情况。

3. 简单性 如上节所述，简单性往往与高效性矛盾。

4. 最优性 算法 A 是最优的是指：这个问题的所有算法中，A 执行的基本运算次数最少。证明一个算法是否是最优的一个方法如下所述。先从理论上证明一个问题所需基本运算次数的下界，如一个算法执行的基本运算的次数就是这个下界的值，则该算法是最优的。空间最优性也可同样分析。

5. 可读性 要求算法易于理解，便于分析。

6. 可修改可扩展性 如问题 P 的一个算法是 A，为了解答一个与 P 相似的问题 P′，希望对 A 稍作改动就可正确运行，如算法 A 满足这一点，则说 A 的可修改性好。反之，如要解决问题 P′，不得不对 A 作多处重大修改，那还不如重新写一个算法，则说 A 的可修改性不好。同样地，有时希望扩大算法 A 所解决的问题的范围，即要给 A 增加一些功能，如只要对 A 稍作修改即可，则称 A 的可扩展性好。反之，A 的可扩展性不好。

§1.3 复杂度分析

为了确定一个完整算法的时间复杂度，可以对算法(或由该算法写成的程序)的各部分进行分析，得到各部分的时间复杂度。再总和各部分的分析结果，就可以得到整个程序的时间复杂度。在开始分析之前，先要确定两件事情：

(1)问题实例大小的量化

(2)基本运算的选择

对构成算法的各种成分执行时所需工作量的分析：

(1)读入一个简单变量，所需时间为常量，记为 O(1).

(2)读入一个数组的所有元素，所需时间与该数组中元素个数成正比，复杂度为元素个数乘一个常量，记为 O(m)，其中 m 为数组元素的个数。FORTRAN 语言中有这样的语句

```
READ(5,112)(((X(I,J,K),K=1,5),J=1,4),I=1,3)
```

这个语句的执行是从输入设备号为 5 的设备上按格式语句 112 读入数组 X，X 为三维数组，共有 60 个元素。

(3)写一个简单变量，所需时间也是常量，记为 O(1).

(4)写一个数组所需时间与数组中含有的元素个数成正比，复杂度为元素个数乘一个常量，如元素个数为 m，可记为 O(m). 例如，语句

```
WRITE(6,106)(TOTAL(I),I=1, 10)
```

将一个含有 10 个元素的一维数组打印出来。

(5)写一个表达式的值，如下述语句

WRITE((A+B)/2)

将表达式

(A+B)/2

的值写出来，执行这种语句所需时间可分两种情况讨论：

(i)如表达式不含有函数调用，则所需时间为常量，记为 O(1).

(ii)如表达式含有函数调用，则所需时间为函数调用所需时间再加上一个常量。

(6)给一个简单变量赋值，且不含函数调用，则所需时间为常数，记为 O(1).

(7)给一个简单变量赋值，但是赋值语句右端包含了函数调用，则所需时间为函数调用所需时间再加上一个常数。

(8)给复杂结构变量赋值(例如 PASCAL 语言可以给一个数组、一个记录赋值)所需时间与该结构中所含纯量数目成正比。

(9)顺序执行的程序段(或算法成分)所需的时间为该程序段中各语句所需时间的和，其大 O 表示形式只与其中耗时最多的语句有关，整个程序段的复杂度的大 O 形式即为这个语句的大 O 形式。

(10)选择执行的成分，如 IF 语句的执行时间决定于 THEN 子句、ELSE 子句耗时多的子句及逻辑表达式所需时间的和。

(11)循环执行的成分，各种循环语句的执行时间为循环体执行时间与循环次数的乘积再加上检验循环条件所需的时间。

(12)对于非递归分程序(如 FORTRAN 语言不允许递归)运行所需时间，可以首先确定不调用任何过程的分程序(这样的分程序必然存在，不然就存在递归)，然后计算这些分程序所需时间，再逐次计算那些只调用复杂度已知的过程的分程序的复杂度，最终可以得到整个分程序的时间复杂度。

(13)递归分程序的时间复杂度不是很容易计算的。这一节的剩余篇幅都用于这个问题。先按递归分程序列出关于复杂度的递归方程，然后解这个方程。

[**例 8**] 考虑下述排序算法，问题实例的大小是 n(在此为要排序的数值的个数)，基本运算为比较。其过程如下：

```
PROCEDURE SORT(VAR A: ARRAYTYPE; N: INTEGER);
{过程 SORT 将数组 A 排序,A 是含有 N 个元素的一维数组};
  VAR  I, J: INTEGER;
  BEGIN
     FOR J:=1 TO N-1 DO
        FOR I:=J+1 TO N DO
           IF A[I]<A[J] THEN SWAP(I,J);
     PRINT(A)
  END
```

其中 SWAP(I,J)是一个过程，其功能为交换元素 A[I]和元素 A[J]的位置；PRINT(A)是另一个过程，它将排序后的数组 A 的元素依次打印出来。过程 SORT 有一个双层循环语句。将外层循环语句的循环控制变量 J 的值及循环体执行的比较操作的次数列表如下：

J	比较操作的执行次数
1	N—1
2	N—2
⋮	⋮
N—1	1

因此总的比较次数为

$$(N-1)+(N-2)+\cdots+1=N(N-1)/2$$

过程 SORT 的时间复杂度

$$T(N)=N(N-1)/2$$

现将该算法改写为递归形式：

```
PROCEDURE  SORT2 (J, N: INTEGER);
```

{A 是一个有 n 个元素的一维数组，SORT2将 A 数组的从第 J 个元素起的子数组排序}

```
VAR  I: INTEGER;
BEGIN
  IF  J=N THEN {子数组只有一个元素}                          (1)
     THEN WRITE(A[N])
  ELSE
     BEGIN  FOR I:=J+1 TO N DO                               (2)
              IF A[I]<A[J] THEN SWAP(I,J);
            WRITE (A[J]);                                    (3)
            SORT2(J+1,N)                                     (4)
     END
END
```

SORT2是个递归过程，递归调用发生在SORT2的结束部分，因此称为尾部递归。SORT2的非递归形式就是过程 SORT. 要给一个有 n 个元素的数组排序，须用如下语句

```
SORT2(1, N)
```

当 J=N 时，即子数组只有一个元素，此时所需工作为打印一个数 A[N]，所需时间为常数，可用 O(1)表示；不然则执行 ELSE 子句，其中语句(2)需要 N—J 次比较，语句(3)的工作是打印一个数，所需时间用 O(1)表示，语句(4)是递归调用语句，如用 T(n)表示 SORT2(1, N)的工作量，则语句(4)SORT2(2, N)的工作量就是 T(n—1)，因此得到如下的递归表达式(n 为要排序的元素个数)

$$T(N)=\begin{cases}1 & ,N=1\\ T(N-1)+(N-1)+1 & ,N>1\end{cases} \qquad (*)$$

则

$$
\begin{aligned}
T(N) &= T(N-1) + (N-1) + 1 \\
&= T(N-2) + [(N-2) + 1] + [(N-1) + 1] \\
&= T(N-3) + [(N-3) + 1] + [(N-2) + 1] + [(N-1) + 1] \\
&= \cdots\cdots \\
&= T(2) + [2 + 1] + \cdots + [(N-3) + 1] + [(N-2) + 1] + \\
&\quad + [(N-1) + 1] \\
&= T(1) + [1 + 1] + [2 + 1] + \cdots + [(N-3) + 1] + \\
&\quad + [(N-2) + 1] + [(N-1) + 1] \\
&= 1 + (1 + 1) + [2 + 1] + \cdots + [(N-3) + 1] + \\
&\quad + [(N-2) + 1] + [(N-1) + 1] \\
&= 1 + 2 + \cdots + (N-3) + (N-2) + (N-1) \\
&\quad + \underbrace{1 + 1 + \cdots\cdots + 1}_{N\text{个}} \\
&= \frac{N(N-1)}{2} + N
\end{aligned}
$$

在计算 SORT 的复杂度时略去了语句 PRINT 的时间。该时间是打印 N 个数的时间(即为 N),这就是算法 SORT2和算法 SORT 的复杂度相差 n 的原因,恰恰说明了两个算法有同样的复杂度。

[例 9] 考虑归并排序算法,其过程如下

```
PROCEDURE MERGESORT (I, J: INTEGER);
{此过程对 A 数组的第 I 到第 J 个元素的子数组进行归并排序}
VAR  M: INTEGER;
BEGIN
  IF  I <>J THEN
    BEGIN M:=(I+J)DIV 2;
          MERGESORT (I, M);
          MERGESORT (M+1, J);
          MERGE
    END
END
```

如要对一维数组 A 进行排序,则要用如下的调用语句

MERGESORT (1,N)

设 N 是2的整次幂,即 $\log_2 N$ 是整数。过程 MERGESORT 调用过程 MERGE,其功能为将两个已排序的子数组进行归并,具体如下:

```
ALGORITHM  MERGE
{一维数组 A 和 B 各含有 M 及 N 个元素,A、B 两数组的尾部还各有一个计算机允许的很大的值,但是实际上不可能出现在数组中的值,MERGE 将这两个数组归并到数组 C 中}
```

```
I:=1; J:=1; K:=0;
WHILE K< M+N DO
    IF  A[I]<B[J] THEN
        BEGIN  K:=K+1;
               C[K]:=A[I];
               I:=I+1
        END
    ELSE BEGIN K:=K+1;
               C[K]:=B[J];
               J:=J+1
         END
```

显然在 WHILE 循环语句中 IF 语句要执行 M+N 次，其中 THEN 子句和 ELSE 子句的内容相同，因此 WHILE 语句，乃至 MERGE 过程的复杂度为 c·(M+N)，其中 c 是循环体执行一次所需时间。过程 MERGESORT 当 I=J 时则不做任何工作，从而 MERGESORT 的复杂度的递归关系式为：

$$T(n) = \begin{cases} 0, & n = 1 \\ 2T(\frac{n}{2}) + c \cdot n, & n > 1 \end{cases} \qquad (**)$$

其中 c·n 是过程 MERGE 所需时间。因此

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + cn \\ &= 2(2T(\frac{n}{4}) + c \cdot \frac{n}{2}) + cn \\ &= 2^2T(\frac{n}{4}) + cn + cn \\ &= 2^2T(\frac{n}{2^2}) + cn + cn \\ &= 2^3T(\frac{n}{2^3}) + cn + cn + cn \\ &= \cdots\cdots \\ &= 2^kT(\frac{n}{2^k}) + \underbrace{cn + \cdots + cn + cn + cn}_{K\text{个}} \\ &= 2^k \cdot O + kcn \\ &= kcn \end{aligned}$$

由 $\frac{n}{2^k} = 1$

得 $k = \log_2 n$

所以

$$T(n) = k \cdot cn = c \cdot n \cdot \log_2 n$$

用大O记号则有

$$T(n) = O(n \log_2 n)$$

例8和例9中的(*)式和(**)式称为递归方程。递归方程的一般形式为

$$T(n)=\begin{cases} C_1 & ,n=1 \\ C_2 & ,n=2 \\ \vdots & \\ C_K & ,n=K \\ a_1T(n-1)+a_2T(n-2)+\cdots+a_KT(n-K)+d(n) & ,n>K \end{cases}$$

其中 $C_1, C_2 \cdots C_K, a_1, a_2 \cdots a_K$ 是常数。

§1.4 展开法

上一节例8和例9的方法可用于解决一类特殊的递归方程。这种方法称为展开法，是一种经常使用的方法。

设递归算法A是这样解决问题的：当问题实例大小 $n=1$ 时，算法A可以直接给出解答，需要时间为c；当 $n>1$ 时，则将问题实例分解为a个小实例，这a个实例的大小都是 n/b；得到了这a个小实例的解以后，再将这a个解合成原问题的完整的解。分解与合成需要的时间为 $d(n)$. 如用 $T(n)$ 表示问题实例大小为n时所需的时间，则据此可以写出如下递归方程

$$T(n)=\begin{cases} c & ,n=1 \\ aT(\frac{n}{b})+d(n) & ,n>1 \end{cases}$$

从而有

$$T(\frac{n}{b}) = aT(\frac{n}{b^2}) + d(\frac{n}{b}),$$

$$T(\frac{n}{b^i}) = aT(\frac{n}{b^{i+1}}) + d(\frac{n}{b^i}), \qquad (n > 1)$$

因此

$$\begin{aligned} T(n) &= aT(\frac{n}{b}) + d(n) \\ &= a^2T(\frac{n}{b^2}) + ad(\frac{n}{b}) + d(n) \\ &= a^3T(\frac{n}{b^3}) + a^2d(\frac{n}{b^2}) + ad(\frac{n}{b}) + d(n) \\ &= \cdots\cdots \\ &= a^KT(\frac{n}{b^K}) + a^{K-1}d(\frac{n}{b^{K-1}}) + \cdots\cdots \end{aligned}$$