第 1 章　绪　　论

随着信息技术的日益发展，计算机已经成为各行业中不可缺少的主要工具。为了编写一个出色的程序，必须分析对象与对象之间潜在的关系，这就是“数据结构与算法”这门学科形成和发展的背景。

“数据结构与算法”作为计算机科学与技术学科的一门重要基础课程，为后续专业课程学习提供了必要的基础知识和技能准备。

1.1　什么是数据结构

瑞士计算机科学家 Niklaus Wirth 提出了著名的公式“程序＝数据结构＋算法”，该公式说明了数据结构和算法对于程序设计是至关重要的，同时也说明了数据结构与算法的关系是密切的。程序可以看作是计算机指令的组合，用于控制计算机的工作流程，完成一定的逻辑功能，实现某种任务。算法是程序的逻辑抽象，是解决一些客观问题的过程。数据结构是对现实世界中数据及其关系的某种映射，数据结构既可以表示数据本身的物理结构，也可以表示计算机中的逻辑结构。下面通过例子来了解什么是数据结构。

【例 1.1】图书馆的图书查询系统。当用户使用图书馆的查询系统时，用户会发现查询系统给出了很多查询条件，可以通过索引号、图书名称、作者姓名和出版社等信息进行查询。每一本书都有唯一的索引号，不同的图书之间可以有相同的名字，或相同的作者名，或相同的出版社。为了方便用户查询，在计算机内部是通过建立数据库表来实现图书信息存储的。如表 1-1 所示，共有四张索引表。其中，表（a）是所有的图书信息，表（b）是按照书名进行索引的表格，表（c）是按照作者姓名索引的表格，表（d）是按照出版社索引的表格。每个索引表的每一行就是一个最简单的线性数据结构。

表 1-1　图书索引表

索引号	图书名称	作者姓名	出版社	
001	工科数学	张三	P1	…
002	大学英语	李四	P2	…
003	工科数学	王五	P2	…
004	英语写作	李四	P3	…
⋮	⋮	⋮	⋮	⋮

(a)

工科数学	001, 003, …
大学英语	002, …
英语写作	004, …
⋮	⋮

(b)

张三	001, …
李四	002, 004, …
王五	003, …
⋮	⋮

(c)

P1	001, …
P2	002, 003, …
P3	004, …
⋮	⋮

(d)

【例 1.2】家族谱的存储。A 姓家族谱如图 1-1 所示，其中“'”为配偶。A_1 和 A_2 是 A 的孩子，A_{11} 是 A_1 的孩子。若将 A 姓家族谱存储在计算机中，可以选择“树”这种数据结构进行存储。

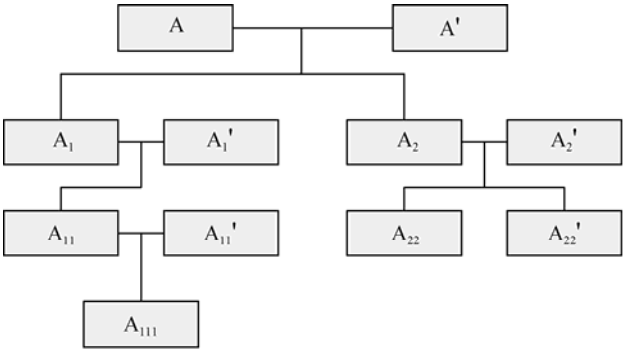

图 1-1　A 姓家族谱

【例 1.3】道路选择问题。如图 1-2 所示，ABCDE 分别代表 5 个城市，边的数字代表从一个城市到达另一个城市的花费。现在一个人想要从城市 A 到达城市 C，要求经过不同的城市，可以选择的路径见表 1-2。

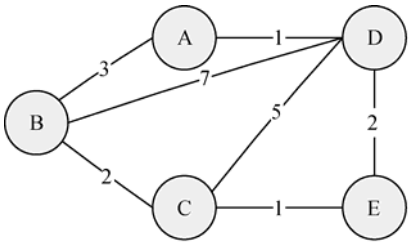

图 1-2　城市道路图

表 1-2　路径花费表

路径	花费
A→B→C	5
A→B→D→C	15
A→B→D→E→C	13
A→D→C	6
A→D→B→C	10
A→D→E→C	4

从表 1-2 的路径花费上可以看出，如果从城市 A 到达城市 C，最好的选择是 A→D→E→C，因为花费最小为 4。这种数据结构称之为“图”。

数据结构（Data Structure）描述的是按照一定的逻辑关系组织起来的待处理数据的表示及相关操作，涉及到数据之间的逻辑关系、数据在计算机中的存储和数据之间的操作（运算）。

1.1.1 数据的逻辑结构

数据的逻辑结构（Logical Structure）是从具体问题中抽象出来的数学模型，体现了事物的组成和事物之间的逻辑关系。

从集合论的观点来看，数据的逻辑结构由数据结点（Node）和连接两个结点的边（Edge）组成。一个逻辑结构可以用一个二元组(K, R)进行表示。其中，K 是由有限个结点组成的集合，R 是一组定义在集合 K 上的二元关系 r，其中的每个关系 r(r∈R)是 K×K 上的二元关系（Binary Relation），用来描述数据结点之间的逻辑关系。例如图 1-1 中的家族关系。

1．结点的数据类型

数据结构的重点是研究数据之间的内在结构关系，所以可以把组成结构的那些元素看作数据结点。结点数据类型可以是基本的数据类型，也可以是复杂的数据类型。

在程序设计语言中常使用 5 种基本数据类型。

- 整数类型（integer）：该类型规定了整数所能表示的范围，由于计算机中一般使用 1-4 字节来存储，所以整数类型的缺陷在于限制了存储字节所能表示的范围；
- 实数类型（real）：计算机的浮点数数据类型所能表示的数据范围和精度是有限的，一般使用 4-8 字节来存储浮点数；
- 布尔类型（boolean）：取值为真（true）或假（false），在 C++语言中一般使用 0 表示假，非 0 表示真；
- 字符类型（char）：用单个字节表示 ASCII 字符集中的字符，字符类型不包括汉字符号。
- 指针类型（pointer）：该类型表示机器内存地址，即表示指向某一内存单元的地址。

复合类型是由基本数据类型组合而成的数据结构类型。例如，在程序语言中的类类型、结构体类型等都属于复合数据类型。复合数据类型本身又可以参与定义更为复杂的结点类型。总之，结点的类型不限于基本的数据类型，可以根据实际需要来灵活定义。

2．结构的分类

讨论逻辑结构(K, R)的结构分类，一般以关系集 R 的分类为主。本书中所讨论的关系 R 集合仅包含一个关系 r，用 r 的性质来刻画数据结构的特点，进行分类。

- 线性结构（Linear Structure）：这种结构是程序设计中最常用的数据结构。线性结构中的关系 r 称为线性关系，也称前驱关系。结点集合 K 中的每个结点在关

系 r 上最多只有一个前驱结点和一个后继结点。如图 1-3 所示。A 是结点集合 K 在关系 r 上唯一的开始结点，A 没有前驱结点，但是具有后继结点 B；结点 B 和结点 C 既有前驱结点又有后继结点；结点 D 为终止结点，只存在前驱结点 C，而不具有后继结点。

- 树形结构（Tree Structure）：简称树结构或层次结构，其中关系 r 称为层次关系。如图 1-4 所示的树形结构图，结点 A∈K 在关系 r 中没有前驱结点，该节点被称为树跟（Root）。结点 C、结点 D 和结点 E 在关系 r 中具有唯一的前驱结点，但是没有后继结点，称这样的结点为叶子（leaf）。除了根结点和叶子结点以外的结点有且只有唯一的前驱结点，但可以具有多个后继结点，例如结点 B。
- 图结构（Graph Structure）：有时候也成为网络结构，Internet 的网页链接关系就是一个非常复杂的网络结构。图结构的关系 r 对于集合 K 中结点的前驱或后继数目不加任何约束。图 1-2 就是一个典型的图结构。

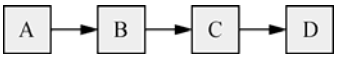

图 1-3　线性结构表示

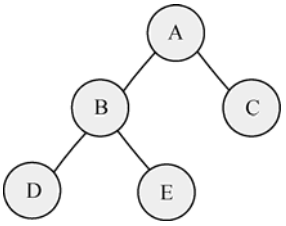

图 1-4　树形结构

从数学上看，线性结构和树形结构的主要区别是“每个结点是否具有一个直接后继”，而树形结构和图结构的主要区别是“每个结点是否仅仅从属于一个直接前驱”。以上几种数据结构分类揭示出数据之间的相互关系，给出关系本身的一种性质。这对于理解数据结构以及设计算法都是至关重要的，后续章节将对以上的数据结构进行详细的讲解。

1.1.2　数据的存储结构

数据的存储结构要解决各种逻辑结构在计算机中物理存储表示的问题。计算机主存储器的特性为其存储提供了一种具有非负整数地址编码的、在存储空间上相邻的单元集合，其基本的存储单元是字节。计算机指令具有按地址随机访问存储空间内任意单元的能力，访问不同地址所需要的访问时间相同。数据存储的主要任务是利用主存储器的“空间相邻”和“随机访问”两个特性，利用地址空间相邻关系来表达数据结构的结点，每个结点通常被存储在一片连续地址的紧凑存储区域内。

对于逻辑结构(K, r)而言，其数据的存储结构就是建立一种逻辑结构到物理结构的映射。一方面，需要建立一个从结点集合 K 到存储器 M 的映射：K→M，其中每一个结点 k∈K 都对应一个唯一的连续存储区域 c∈M 。另一方面，还要把每一个关系元组

$<k_i,k_j>\in r$（其中$k_i,k_j\in K$是结点）映射为相应的存储单元的地址间的关系（顺序关系或指针的地址指向关系）。

下面介绍4种常用存储映射的方法：顺序方法、链接方法、索引方法和散列方法。

1．顺序方法

顺序存储方法把一组结点存放在一片地址相邻的存储单元中，结点间的逻辑关系用存储单元间的自然关系来表达。顺序存储法为使用整数编码访问数据结点提供了便利。程序设计语言内提供的数组是顺序方法的一个具体实例。

顺序存储结构通常也被成为紧凑存储结构。其紧凑性是指其存储空间除了存储数据本身之外，没有存储其他附加的信息。紧凑性可以用“存储密度”来度量：所存储的“数据”占用的存储空间和该结构（包括附加信息）占用的整个存储空间大小之比。显然，存储密度太小的存储结构的空间效率比较低。

除线性结构外，对于部分非线性数据结构也可以采用顺序存储方法。例如，树形结构也可以采用顺序方式来存储，前提是同时存储一些附加信息来表示结点之间的逻辑关系。

2．链接方法

链接方法是在结点的存储结构中附加指针域来存储结点间的逻辑关系。链接方法中数据结点由两部分组成：数据域存放结点本身的数据，指针域存放指向其后继结点的指针。

根据应用的需要，结点的指针域也可存储多个指针来表达一个结点同时链接多个结点的情况。例如，非线性结构中一个结点可能会有多个后继结点的情况。

链接方法适用于那些需要经常增删结点而动态变化的数据结构。链接方法的缺陷在于：为了访问结点集K中的某个结点，必须使用指向该结点的指针。当不知道结点指针时，为了在结点集K中寻找某个符合条件的结点，就要从链头开始沿着链接结点的链索，通过逐个结点的比较来搜索。

3．索引方法

索引法是顺序存储的一种推广，通过建造一个由数据域Z映射到存储地址域D的索引函数Y：Z→D，把数据索引值z映射到结点的存储地址$d\in D$，从而形成一个存储了一组指针的索引表，每个指针指向存储区域的一个数据结点。索引表的存储空间是附加在结点存储空间之外的，每一个元素就是指向相应数据结点的指针，即结点存储单元的开始地址。作为一种存储机制，索引的主要作用是提高检索的效率。当数据量很大的，对数据的检索可能涉及大量读/写磁盘的操作，会影响效率。通过索引则可以降低读/写的数据量，根据检索码确定了被检索数据的存储地址之后再进行相应的读/写。

4．*散列方法*

作为索引法的一种延伸和扩展，散列法利用散列函数进行索引值的计算，然后通过索引表求出结点的指针地址。其主要思想是根据结点的关键码值来确定其存储地址，利用一种称为散列函数（Hash Function）的关系把关键码的值映射到存储空间的地址，然后把结点存入此存储单元中。

散列技术的关键问题是如何选择和设计恰当的散列函数、构造散列表、研究构造散列表存储的碰撞解决方案等。

一个散列函数把一个给定的关键码映射为一个小于 K 的非负整数，K 的大小取决于具体的应用。散列函数应该满足一些重要的散列性质：散列函数计算出的地址尽可能均匀地分布在构造的散列表地址空间，散列函数的计算应该简单化，以便提高地址计算速度。

在一个具体的应用中，可以根据需要选用以上 4 种存储方法或其组合。例如，树形结构的子结点表示方法就是顺序和链接的结合。另外，一个逻辑结构可以有多种不同的存储方案，在选择存储方法时，还要综合考虑定义在其上的运算及其算法的实现。

1.2　算法与算法设计

1.2.1　算法的概念

粗略地说，算法（Algorithm）是对特定问题求解过程的描述，是指令的有限序列，即为解决某一特定问题而采取的具体而有限的操作步骤。程序是算法的一种实现，计算机按照程序逐步执行算法，实现对问题的求解。

求解最大公因子的辗转相除法，以及求解联立线性方程组的主元素消去法都是算法的典型例子。一个求解问题通常用该问题的输入数据类型和该问题所要求解的结果（算法的输出数据）所应遵循的性质来描述。以求最大公因子算法为例，它的输入是整数类型，输入数据是任意给定的两个正整数 n 和 m，而算法的输出则是既能整除 n 又能整除 m 的所有公因子中的最大的非负整数。对于求解联立线性方程组，它的输入数据是方程的系数矩阵和方程等式右侧的常数向量，而其输出结果数据是方程变元的一组取值，它们代入方程应该满足所给的等式。

算法一般具有以下性质：

1）算法的通用性。算法通用性的含义是对于那些符合输入数据类型的任意输入数据，都能根据算法进行问题求解，并保证计算结果的正确性。

2）算法的有效性。算法是有限条指令组成的指令序列，其中每一条指令都必须能够被人或机器所确切执行。指令的类型应该明确规定，仅限于若干明确无误的指令动作，是一个有限的指令集。

3）算法的确定性。算法每执行一步之后，对于它的下一步应该有明确的指令。下

一步的动作可以是条件判断、分支指令、顺序执行一条指令或者指示整个算法的结束等。算法的确定性就是要保证每一步之后都有关于下一步动作的指令，不能缺乏下一步指令（被锁住）或仅仅有模糊不清的指令。

4）算法的有穷性。算法的执行必须在有限步内结束。也就是说，算法不能含有死循环。注意，算法由有限条指令所组成这一事实本身并不能保证算法执行的有穷性，在设计算法时，应该关注算法的结束条件。

以上的基本性质涉及算法的输入、输出、通用性、可行性、确定性和有穷性等多个方面，这些方面有助于对算法这个概念的确切理解。

1.2.2 算法设计

算法设计与算法分析是计算机科学的核心问题。算法设计是求解问题时必须考虑的，其任务是对各类具体问题设计求解的方法和过程，常用的算法设计方法有穷举法（Enumeration）、回溯法（Back track）、分治法（Divide and Conquer）、递归法（Recursion）、贪心法（Greedy）和动态规划法（Dynamic Programming）等。下面对各种方法给予简单的介绍。

1．穷举法

穷举法也称为枚举法，其基本思想是将问题空间的所有求解对象一一列举出来，然后逐一加以分析、处理，并验证结果是否满足给定的条件。穷举完所有对象后，问题将最终得以解决。穷举法具有以下特点：

- 对象应该是有限的，有明显的穷举范围；
- 有穷举规则，可按照某种规则列举对象；
- 一时找不出解决问题的更好途径。

一般而言，对一个问题空间进行全局的穷举，往往很浪费时间，效率上难以满足要求，但在问题的局部采用穷举法还是很有效的。

2．回溯法

回溯法也称为试探法，该方法的基本思想是将问题的候选解按照某种顺序逐一枚举和检验，来寻找一个满足预定条件的解。当发现当前候选解不可能是解时，就回退到上一步重新选择下一个候选解。如果当前候选解还不满足问题规模要求，但是满足所有其他要求，那么继续扩大当前候选解的规模，并继续试探。如果当前候选解满足包括问题规模在内的所有要求时，该候选解就是问题的一个解。在回溯方法中，放弃当前候选解，寻找下一个候选解的过程成为回溯。扩大当前候选解的范围，以继续试探的过程称为向前试探。

3．分治法和递归法

分治法的设计思想很朴素，其要点是在遇到一个难以直接解决的大问题时，将其分割成一些规模较小的子问题，以便各个击破，分而治之，然后把各个子问题的解合并起来，得出整个问题的解。分治法是一种自顶向下的设计方法。

如果原问题可以分割成若干子问题，这些子问题都可解，并且可以利用这些子问题的解求出原问题的解，那么这种分治法是可行的。由分治法产生的子问题往往是原问题的较小规模，在这种情况下，反复应用分治手段，可以使问题的规模不断缩小，最终缩小到容易直接求解的规模，这自然会导致递归过程的产生。分治法和递归法如同一对孪生兄弟，经常同时应用在算法设计之中，并由此产生很多高效算法。

4．贪心法和动态规划法

贪心法的基本思想是从问题的初始状态出发，依据某种贪心标准，通过若干次的贪心选择而得出最优值（或较优解）。贪心法并不是从整体上考虑问题，它所做出的选择只是在某种意义上的局部最优解，寄希望于由局部最优解构建全局最优解。选择能产生问题的最优解的最优度量标准是使用贪心法的核心问题。

比贪心法更为一般的动态规划法通常也用于求解具有某种最优性质的问题。当一个问题的解可以看成一系列判定的结果时，可以利用动态规划方法设计其求解算法。在这类问题中，可能会有许多可行解，希望从中找出具有最优值的解。动态规划与分治法类似，其基本思想也是将待求解问题分解为若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。与分治法不同的是，适合于用动态规划求解的问题，经分解得到的子问题往往不是互相独立的。若用分治法来解这类问题，则分解得到的子问题题目太多，有些子问题被重复计算很多次。如果能够保存已解决的子问题答案，而在需要时利用这些已求的的答案，就可以避免大量的重复计算，节省时间。一般用一个表来记录所有已解决的子问题的答案。不管子问题以后是否被用到，只要它被计算过，就将其结果填入表中，这就是动态规划法的基本思路。

可以选择和组合这些算法设计方法，根据问题的资源约束和要求，设计出相关数据结构和算法来求解问题。运用算法的常见方式有以下 4 种：

1）算法的组合。

2）经典算法的变形和推广。

3）研究困难问题的特殊情况。

4）探索新的算法。

1.3 算 法 分 析

这一节将介绍算法分析的基础知识，重点是分析执行算法时计算机所必须使用的时空资源。解决同一个问题总是存在着多种算法，而算法设计者需要在所花费的时间和所使用的空间资源两者之间采取折中，采用某种以空间资源换取时间资源的策略。通过算法分析，可以判断所提出的算法是否可行。本节将引入增长率函数、算法的增长率估计以及算法复杂度等概念，并引入渐进分析方法。

1.3.1 算法的渐进分析

一般情况下，用来表示数据规模和时间关系的函数都相当复杂。计算这样的函数时通常只考虑大的数据，而那些不能显著改变函数量级的部分都可以忽略掉，其结果就是原函数的一个近似值，这个近似值在数据规模很大时会足够接近原值。这种方法称为渐进算法分析（Asymptotic Algorithm Analysis）方法，简称渐进分析。例如，下面的这个函数：

$$f(n)=n^2+100n+\log_{10}n+1000$$

在 n 的取值很小时，例如为 1 时，最后的常数项 1000 对函数值的贡献最大；当 n=10 时，第 2 项（100n）和最后的 1000 具有相同的贡献；当 n 达到 100 时，前两项的贡献相同；但当 n 大于 100 后，第 2 项的贡献就小于第 1 项，而且 n 越大，第 2 项对函数值而言就越微不足道。由于第 1 项是二次增长的，当 n 取值很大时，函数的取值主要依赖于第 1 项的贡献。

渐进分析是对资源开销的一种不精确的估计，它提供的是对算法所需资源开销进行评估的简单化模型，以便把注意力集中在最重要的部分。应该注意的是，并非所有的情况下都可以忽略常数部分。当算法要解决的问题规模很小时，各项系数和常数项就会起到举足轻重的作用，如同上面的函数所示。因此，用于上万个数的排序算法也许并不适用于仅对 10 个数的排序。

1．大 O 表示法

渐进分析最常用的表示方法是估计函数的增长趋势，采用由 Paul Bachmann 于 1894 年引入的大 O 表示法。假设 f 和 g 为从自然数到非负实数集的两个函数。

【定义 1】如果存在整数 c 和正整数 N，使得对任意的 n≥N，都有 f(n)≤cg(n)，则称 f(n)在集合 O(g(n))中，或简称 f(n)是 O(g(n))的。

该定义说明了函数 f 和 g 之间的关系，即可以表达成函数 g(n)是函数 f(n)取值的上限(Upper Bound)，也可以说函数 f 的增长最终至多趋同于 g 的增长。

因此，大 O 表示法提供了一种表达函数增长率上限的方法。换言之，若某种算法在 O(g(n))中只是表明了该算法最多会差到何种程度。当然，一个函数增长率的上限可能不

止一个。例如，一个在集合 O(n)中的函数也一定在集合 $O(n^2)$中，同时也在集合 $O(n^3)$中。大 O 表示法给出了所有上限中最小的那个上限。

下面列出了大 O 表示法所具有的某些有益特性，可在计算算法效率时使用。

1）如果函数 f(n)是 O(g(n))的，g(n)是 O(h(n))的，则 f(n)是 O(h(n))的。

2）如果函数 f(n)是 O(h(n))的，g(n)是 O(h(n))的，则 f(n)+g(n)是 O(h(n))的。

3）函数 an^k 是 $O(n^k)$的，符号 a 表示不依赖于 n 的任意常数。

4）若 f(n)=cg(n)，则 f(n)是 O(g(n))的。

5）对于任何正数 a 和 b，且 b≠1，函数 $\log_a n$ 是 $O(\log_b n)$的。即任何对数函数无论底数为何值，都具有相同的增长率。

6）对任何正数 a≠1，都有 $\log_a n$ 是 $O(\log_2 n)$的。本书把 $\log_2 n$ 简写为 logn。

常见的上限 g(n)有以下若干种，表达式中的符号 a 是不依赖于 n 的任意常数，$\text{rate}_{n\to\infty} f(n)$ 表示 n 趋于无限大时函数 f(n)的极限：

1）g(n)=1，常数函数，不依赖于数据规模 n。

2）g(n)=logn，对数函数，它比线性函数 n 增长慢。

3）g(n)=n，线性增长，随着时间规模 n 而增长。例如：

$$\text{rate}_{n\to\infty}\ (\text{n 个 a 相加}) = \text{rate}_{n\to\infty}\ (n\times a) = O(n)$$

4）$g(n)=n^2$，二阶增长，例如：

$$\text{rate}_{n\to\infty}(1+2+3+\cdots+n) = \text{rate}_{n\to\infty}(n\times(n+1)/2) = O(n^2)$$

5）g(n)=nlog(n)，其增长率的阶数低于二阶，但高于一阶线性。例如：

$$\text{rate}_{n\to\infty}\sum_{i=1}^{\log n}\sum_{j=1}^{n} a = O(n\log(n))$$

这种 nlog(n)型的渐进式一般出现在树形数据结构算法中。

6）$g(n)=a^n$，指数增长，随问题规模 n 快速增长。

这种指数型的渐进式往往出现在递归定义的函数计算中。值得注意的是，指数增长的渐进式比任何高次的多项式函数如 n^3、n^4 都增长得更快。

这些渐进式的函数曲线，在 n 很大的时候其大小差别非常大。举例来说，令时间单位为μs，在 n 等于 1000 时，对于线性增长率，时间开销 T(n)=n 为 1000μs，即 1ms；而对二次增长率 $T(n)=n^2$,其时间开销是 10^6μs，即 1s；对于三次增长率的 $T(n)=n^3$ 而言，其时间开销就是 1000s(约 16 分钟)。对于指数型的增长率 $T(n)=2^n$ 所花费的时间约为 10^{286} 年，而迄今地球的年龄还不超过 10^{10} 年。具有指数增长率的算法简称指数爆炸型算法，在应用时需要谨慎使用。

2．Ω 表示法

大 O 表示法给出了函数增长率的上限。与此相对，还有Ω表示法，Ω读做“欧米伽”(Omega)。其定义如下：

【定义 2】如果存在整数 c 和正整数 N，使得对所有的 n≥N，都有 f(n) ≥cg(n)，则

称函数 f(n)在集合Ω(g(n))中，或简称 f(n)是Ω(g(n))的。

此定义说明了 cg(n)是函数 f(n)取值的下限(Lower Bound)，也可以说函数 f(n)的增长最终至少是趋同于函数 g(n)的增长。

大 O 表示法和Ω表示法的唯一区别在于不等式的方向不同。正如大 O 表示法，Ω表示法是在函数增值率的所有下限中最大的下限。

3．Θ表示法

大 O 表示法和Ω表示法描述了某一函数增长的上限和下限。当上、下限相同时，则可以用Θ（读做“希塔”）表示法。如果一个函数既在集合 O(g(n))中又在集合Ω(g(n))中，则称其为Θ(g(n))。即存在正常数 c_1、c_2 以及正整数 N，使得对于任意的正整数 n>N 有下列两不等式同时成立：

$$c1g(n) \leqslant f(n) \leqslant c2g(n)$$

此定义具体给出了下限估计和上限估计两种估计式。例如：

$$f(n)=100\times n^2+5\times n+500$$

令 $g(n)=n^2$，此时存在常数 $c_1=100$，$c_2=105$，N=10，当 n>N 时，

$$c1g(n) \leqslant f(n) \leqslant c2g(n)$$

成立。因此可以说 f(n)为Θ(n^2)。

4．渐进分析的实例

通过对算法所需要的时间和空间代价的估算，渐进分析方法可以衡量算法的复杂度。实际上，大多数情况下，算法的时间复杂度是根据算法执行过程中需要实施的赋值、比较等基本运算数目的多少来衡量的。

先从一个简单的例子开始分析。下面是一段对数组中的各个元素求和的代码：

```
for (i=sum=0; i<n; i++)
    sum+=a[i];
```

其中，主要的操作为赋值运算，因此该算法的时间代价主要体现在赋值操作的数目上。在循环开始之前有两次赋值，分别对 i 和对 sum 进行；循环进行了 n 次，每次循环中执行两次赋值，分别对 sum 和 i 进行更新操作。总共 2+2n 次赋值操作，其渐进复杂度为 O(n)。

在循环中嵌套循环，其复杂度会相应增大。例如，在这段代码依次求出给定数组的所有子数组中各元素之和：

```
for (i=0;i<n;i++){
    for (j=1,sum=a[0];j<=i;j++)
        sum+=a[j];
}
```

循环开始前，有一次对 i 的赋值操作。之后，外层循环共进行了 n 次，每个循环中

包含一个内层循环以及对 i、j、sum 分别进行赋值的操作；每个内层循环执行 2 个赋值操作，分别更新 sum 和 j；共执行 i 次(i=1,2,⋯,n−1)。因此，整个程序总共执行的赋值操作为

$$1+3n+\sum_{i=1}^{n-1}2i=1+3n+2(1+2+\cdots+n-1)=1+3n+n(n-1)$$

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

一般情况下，循环中嵌套循环会增加算法的复杂度，但也并非总是如此。例如，在上面的实例中，如果只对每个子数组的前 5 个元素求和，则对应的代码可采用下面的方式：

```
for (i=4;i<n; i++)
for (j=i-3,sum=a[i-4];j<=i;j++)
    sum+=a[j];
```

此时，外层循环进行 n−4 次。对每个 i 而言，内层循环只执行 4 次，每次的操作次数与 i 的大小无关：8 次赋值操作。外加对 i 的初始化，整个代码总共进行 O((1)+8(n−4))=O(n)次赋值操作。尽管存在嵌套循环，但算法的整体时间复杂度依然呈线性增长。

另外，对于排序算法而言，其主要时间开销体现在比较和交换(或移动)等操作上，具体可参考第 6 章的算法分析。

1.3.2 最坏、最好和平均情况

对于某些算法，即使问题的规模相同，如果输入数据不同，其时间复杂度也不同。换言之，算法的渐进分析往往无法独立于输入数据的状态而进行。这是因为算法实际执行的操作往往依赖于算法中分支条件的走向，而这些分支走向又取决于输入数据的取值。为此，在算法进行渐进分析时，会根据输入数据的取值分情况进行，以便确切了解各种算法所适用的情况以及能否在规定的响应时间内完成。

例如，求一个数组的所有有序子数组中最长的一个。在数组[1, 7, 1, 2, 3, 5, 1, 12, 6]中，这个最长有序子数组为[1, 2, 3, 5]，长度为 4。可用以下代码实现：

```
for(i = 0, length = 1; i < n - 1; i++)
{
    for(j1 = j2 = k = i; k < n - 1 && array[k] < array[k + 1]; k++, j2++);
    if(length < j2 - j1 - 1)
        length = j2 - j1 + 1;
}
```

这段代码的时间代价和数组 array 中元素的实际取值状态相关。根据这些元素的初始状态可以分成以下几种情况：

1）如果数组 array 的所有元素是以降序方式输入的，那么外层循环执行 n–1 次，每次内层循环只执行一次，整个的时间开销为 O(n) 。

2）如果数组 array 的所有元素是以升序方式输入的，那么外层循环执行 n−1 次，对于每一个 i，内层循环执行 n−1−i 次，整个的时间开销是 $O(n^2)$。

3）在大多数情况（即平均情况）下，数组的元素是无序的，既不按照升序也不按照降序输入，下面介绍一种如何计算平均情况下的算法复杂度。

一般而言，计算平均情况的复杂度应该考虑算法的所有输入情况，确定针对每种输入情况下所需的操作数目。在简单情况下，如果每种输入出现的概率相同，可把针对每种输入的操作数目求和，再除以输入的总数目来得到平均的开销。但是，每种输入的出现概率并非是相等的，此时分析平均情况下的复杂度就需要把每种输入出现的概率作为权值加以考虑，即 $C_{avg}=\sum_i p(input_i)steps(input_i)$。

此处，假设可以事先得知输入的概率分布情况。$p(input_i)$ 为第 i 种输入的出现概率，$steps(input_i)$ 为算法处理第 i 种输入时所需的操作或步骤数。当然，所有的概率均为非负数，且满足 $\sum_i p(input_i)=1$。

尽管平均情况的复杂度是算法在输入规模为 n 时的典型表现，但平均情况的分析并不总是可行的，因为这需要了解算法的实际输入在所有可能的输入集合中的分布情况。例如，上面例子中获取数组元素的平均分布情况并不是很容易。

以从一个规模为 n 的一位数组中找出一个给定的 K 值为例（假设该数组中有且仅有一个元素的值为 K），顺序检索法将从第一个元素开始，依次检查每一个元素，直到找到 K 为止。

最佳情况下，数组中第一个元素为 K，此时只需要检查一个元素即可。

最差情况下，K 是数组的最后一个元素，此时算法需要检查 n 个元素才能找到 K。

平均情况下，如果 K 出现在数组中的每个位置上的概率相等，即 K 在每个单元中的概率均为 $\frac{1}{n}$，平均需要 $\frac{1+2+\cdots+n}{n}=\frac{n+1}{2}$ 次比较才能找到。

概率不相等时，例如出现在第 1 个位置的概率为 1/2，第 2 个位置上的概率为 1/4，而出现在其他位置的概率相等，即

$$\frac{1-1/2-1/4}{n-2}=\frac{1}{4(n-2)}$$

则查找 K 平均需要

$$\frac{1}{2}+\frac{2}{4}+\frac{3+\cdots+n}{4(n-2)}=1+\frac{n(n+1)-6}{8(n-2)}=1+\frac{n+3}{8}$$

次比较，比等概率的 $\frac{n+1}{2}$ 快将近 4 倍。

由此可见，第 1 种情况是最佳的，所需的步骤最少；第 2 种情况所需的步骤最多，是最差的情形；第 3 种则介于最佳和最差之间。那么分析一种算法时，应该研究最佳、最差还是平均情况呢？一般而言，最佳情况发生的概率太小，并不能作为算法性能的代

表，但它可以帮助算法设计者或使用者了解某个算法在何种情况下使用。而最差情况可以让人了解一个算法至少能做多快，这点在实时系统中尤其重要。例如，在空运处理系统中，一个绝大部分情况下能管理 n 架飞机的算法，如果不能在规定时间内管理 n 架飞机，则该算法是不可接受的。此外，对于多数算法而言，最坏情况和平均情况的时间开销的公式虽然不同，但是往往只是常数因子大小的区别，或者常数项大小的区别。

上面 3 种情况的复杂度都相对比较简单，可以得到很精确的函数关系，但复杂度与数据规模之间的关系并非总是一目了然。尤其是平均情况的复杂性分析往往需要复杂的计算，此时便可以使用前面介绍的大 O、Ω 和 Θ 等渐进分析法。

1.3.3 时间和空间资源开销

对于空间开销，也可以实行类似的渐进分析方法，不过，很多常见算法所使用的数据结构是静态的存储结构。所谓静态的含义是，算法所使用的存储空间在算法执行过程中并不发生变化。一旦输入数据和问题规模确定，它的数据结构大小也就确定下来。对于这种静态数据结构，空间开销的估计往往是容易的，它们或者与所涉及的问题规模成正比（空间开销为线性增长），或者不随问题的规模而增大（空间开销为常数）。本书的后续章节中对这类使用静态数据结构的算法一般将仅限于讨论时间开销。当然，在算法运行过程中有时会有空间开销的增大或缩小。对于这种情况，空间开销的分析和估计是十分必要的。

在算法设计分析中，还涉及到一个“时间资源的折中原理”。这个原理可以简述如下：对于同一个问题求解，一般会存在多种算法，而这些算法在时空开销上的优劣往往表现在“时空折中”的性质。所谓的“时空折中”，是指为了改善一个算法的时间开销，往往可以通过增大空间开销为代价，设计出一个新算法。例如，为了从公司黄页数据表中快速查找公司电话（假设每个公司的名称、地址和电话都存储在数据表中），可以附加一个散列式索引表，其散列函数可以将公司名称快速变换为索引值，通过这个索引值可以读出一个指针，指向存储该公司电话的存储单元。这种散列法增加了索引表的空间开销，但节省了时间，用散列函数的简单计算代替了原本耗时的在黄页中逐项进行公司名称的比较操作，时间开销从线性增长改善为常数时间，而空间开销方面则增添了一个线性增长的存储空间。在设计算法时，经常采用这种以空间换时间的办法。为此，就需要对原有存储结构做出修改，或者设计出全新的、更适合逻辑数据结构使用的存储方案。当然，有时也可以牺牲计算机的运行时间，通过增大时间开销来换取存储空间的节省，例如树的顺序存储。

1.4 数据结构的选择和评价

在求解一个问题时，怎么去设计或者选择数据结构呢？对于初学者来说，主要遵循以下原则：

1）仔细分析所要解决的问题，特别是求解问题所涉及的数据类型和数据间的逻辑关系。

2）在算法设计之前往往先进行数据结构的初步设计。

3）注意数据结构的可扩展性，包括考虑当输入数据的规模发生改变时，数据结构是否能够适应。同时，数据结构应该适应求解问题的演变和扩展。

4）数据结构的设计和选择也要比较算法的时空开销的优劣。

数据结构的选择和评价是复杂的，需要考虑的因素也不限于以上的几条，一般要具体情况具体分析。本书会针对具体问题进一步阐述和检验这些原则。

习 题

1．简述逻辑结构与存储结构的关系。

2．度量一个算法的执行时间通常有几种方法？各有何优缺点？

3．分析下面函数的时间复杂度。

```
void func(int n)
{
    int i=1,k=100;
    while (i<n)
{
    k++;i+=2;
}
}
```

4．设 n 是偶数且有程序段：

```
for (i=1;i<=n;i++)
    if(2*i<=n)
        for (j=2*i;j<=n;j++)
            y=y+i*j;
```

则 y=y+i*j 的执行次数是多少？要求列出计算公式。

5．将下列函数按它们在 n→∞时的无穷大阶数，从小到大排列。

n , $n-n^3+7n^5$, $n\log n$, $2^{n/2}$, n^3 , $\log n$, $n^{1/2}+\log n$, $(3/2)^n$, $n!$, $n^2+\log n$