第一部分

预 备 知 识

第 1 章　数据结构和算法

得克萨斯州达拉斯市方圆 500 英里[①]之内有多少个人口超过 250 000 人的城市？一个公司里有多少人年收入超过 100 000 美元？用不超过 1000 英里长的电缆是否能把所有电话用户都连起来？现在就要正确回答类似这样的问题是不可能的，因为这里还没有给出回答这些问题所必须要了解的信息。必须以能及时找到答案的方式来有效地组织信息，才能回答上述问题。

信息表示是计算机科学的基础。大多数计算机程序的主要目标与其说是完成运算，倒不如说是存储信息和尽快地检索信息。因此，研究数据结构和算法就成为了计算机科学的核心问题。本书的目的就是帮助读者理解怎样组织信息，以便支持高效的数据处理。

本书有三个主要目的。第一个目的是介绍常用的数据结构，这些数据结构形成了一个程序员基本数据结构工具箱(toolkit)。对于许多问题，工具箱里的数据结构是理想的选择。

第二个目的是引入并加强“权衡”(tradeoff)的概念，每一个数据结构都有相关的代价和效率的权衡。本书通过列举出不同的数据结构，并指出它们应用于实际问题时的代价和效率来讨论“权衡”的概念。

第三个目的是讲解如何评估一个数据结构或算法的有效性。只有通过这样的分析，才能确定对一个新问题最合适的数据结构是工具箱中的哪一个。这种技术也使得程序员能够判断自己或别人发明的新数据结构的价值。

一个问题通常有多种解法，应该选择哪一种呢？计算机程序设计的核心有两个目标(有时它们互相冲突)：

1. 设计一个容易理解、编码和调试的算法；
2. 设计一个能有效利用计算机资源的算法。

在理想情况下，最终的程序都能实现这两个目标，有时把这样的程序称为是“完美的”。本书给出的算法和程序实例在这种意义上来说是趋于完美的。与目标 1 有关的问题不是本书的目的，它们主要涉及的是软件工程原理。而本书主要讲的是与目标 2 有关的问题。

怎样度量效率呢？第 3 章将给出估算一个算法或者一个计算机程序效率的方法，称为算法分析(algorithm analysis)。算法分析还可以度量一个问题的内在复杂程度。以后的章节中对算法的时间代价进行衡量时，都会用到算法分析方法。利用这种方法可以清楚地看到在解决同一个问题时不同算法在效率上的差异。

这一章提出与选择和使用数据结构相关的话题，通过这些话题为后面的内容做准备。本章将先审视设计者为要完成的任务选择一个合适的数据结构所经历的过程，然后讨论在程序设计中如何进行抽象，最后将探索问题、算法和程序三者之间的关系。

---

① 1 英里 = 1.609 千米。

1.1　数据结构的原则

1.1.1　学习数据结构的必要性

有人可能认为，随着计算机功能的日益强大，程序的运行效率变得越来越不那么重要。毕竟，处理器的速度和内存容量还在不断提升。为什么那些效率的问题不能使用将来的硬件来解决呢？

人们不断开发出性能更强大的计算机，历史上也能看到许多使用计算机技术来解决的复杂问题，例如更复杂的用户界面、更大的问题规模，或者以前因为计算复杂度太大无法计算的问题。更复杂的问题需要更大的计算量，这就使得对高效率程序的需求更加明显。糟糕的是，工作越复杂就越偏离人们的日常经验。当今的计算机科学家必须学习和具备彻底理解隐藏在高效率程序设计后面的一般原理的能力，因为在日常生活经历中并不会用到进行程序设计才需要的这些能力。

简单地说，一种数据结构就是一类数据的表示及其相关的操作。即使是存储在计算机中的一个整数或者一个浮点数，也是一个简单的数据结构。更一般而言，人们认为“数据结构”是一组数据项的组织或者结构。存储在数组中的一个有序整数表就是这种结构的一个例子。

如果有足够的空间来存储一组数据项，总会有可能在这个数据项集合中查找出指定的数据项、打印数据项或将这些数据项处理成任何期望得到的顺序，或者更改任何特定数据项的值。因此，就有可能对任何数据结构执行所有必要的运算。然而，选择不同的数据结构可能会产生很大的差异：同样一个程序，选择一种数据结构可能在几秒钟内就运行完毕，而选择另一种数据结构则可能需要几天时间才能完成运行。

一个算法如果能在所要求的资源限制(resource constraint)内将问题解决好，则称这个算法是有效率的(efficient)。例如，一个资源限制是：可用来存储数据的全部空间——可以分为内存空间限制和磁盘(外存)空间限制——和允许执行每一个子任务所需要的时间。一个算法如果比其他已知算法所需要的资源都少，那么这个算法也可以称为是有效率的，而不管该算法是否有其他特殊要求。一个算法的代价(cost)是指这个算法消耗的资源量。一般来说，代价是由一个关键资源(例如时间)来评估的，这意味着这个算法满足其他资源限制。

毋庸置疑，人们编写程序是为了解决问题。在选择数据结构解决特定问题时，头脑中有这个不言而喻的道理是具有重要意义的。只有通过预先分析问题来确定必须要达到的性能目标，才有希望挑选出正确的数据结构。水平不高的程序设计人员往往忽视了这一分析过程，而直接选用某一个他们习惯使用的、但是与问题不相称的数据结构，结果设计出一个低效率的程序。相反，当使用简单的设计能达到性能目标时，选用复杂的数据表示来改进这个程序也是没有道理的。

当为解决某一问题而选择数据结构时，应该完成以下几步：

1. 分析问题以确定必须支持的基本操作。基本操作的实例包括向数据结构中插入一个数据项、从数据结构中删除一个数据项和查找指定的数据项。
2. 衡量每种基本操作会遇到的资源限制。
3. 选择最接近这些代价的数据结构。

根据这三个步骤来选择数据结构，实际上贯彻了一种以数据为中心的设计观点。先定义数据和对数据的操作，然后确定数据的表示方法，最后是数据表示的实现。

某些重要的操作，例如查找、插入和删除数据记录的资源限制通常决定了数据结构的选择过程。对于这些操作相对重要性的争论焦点集中在以下三个问题中。无论什么时候，只要选择数据结构，就应该仔细考虑这三个问题：

- 开始时将所有数据项都插入数据结构，还是与其他操作混合在一起插入？静态应用(数据在一开始就被载入内存中并且不会改变)与动态应用相比，通常只需要一些简单的数据结构就可以得到比较高效率的性能。
- 数据项可以删除吗？如果可以，这会使实现更加复杂。
- 所有数据项是否按照某一个已经定义好的顺序排列？或者是否允许查找特定的数据项？“随机访问”搜索通常需要更复杂的数据结构。

1.1.2 代价与效益

每一个数据结构都把代价与效益联系在一起。如果有人说某个算法在所有情况下都比其他算法好，这通常在实践上是不正确的。如果一个数据结构或算法在各方面都比另一个要好，那么差的那一个肯定很久没人使用了。本书提到的几乎每一个数据结构和算法，都有一些例子，用来说明它在什么地方才是最好的选择，其中有些例子是出人意料的。

一个数据结构需要一定的空间来存储它的每一个数据项，需要一定的时间来执行单个基本操作，也需要一定的程序设计工作。每一个问题都有可利用的空间和时间的限制。问题的每一个解决方案都利用了一定比例的相关基本操作，数据结构的选择过程必须考虑到这一点。只有对问题的特性仔细分析之后，才能得到执行这项任务的最好的数据结构。

例 1.1 一个银行必须支持与顾客相关的多种交易，但这里人们只考虑一个简单的情况：顾客可以新开账户、注销账户，以及从账户上存款和取款。可以从两个层面考虑这个问题：(1)银行用来与顾客交互的物理基础结构和工作处理流程的需求，(2)管理账户的数据库系统需求。

一般的顾客进行新开账户和注销账户的次数通常远少于从账户上取款和存款的次数。在新开账户或注销账户时，顾客可以等候许多分钟，而在进行具体的每一笔账户交易业务时，例如存款或取款，他们就不愿意进行长时间的等待。

银行的实际做法是提供两级服务：人工出纳或自动取款机(ATM)可以使顾客查询账面余额和进行诸如存款和取款之类的账面更改，而特殊服务一般只提供(在限定的几小时内)进行开户和销户的业务。出纳和 ATM 交易仅花费很短的时间，而开户和销户操作则可能花费很长的时间(或许在顾客眼里最长达到一个小时)。

从数据库方面来看，ATM 交易业务并不会过多地更改数据库。如果简单地假设只是钱增加或减少了，则这种交易业务只是简单地改变了存储在账户记录里的值。向数据库中添加一个新账户可以允许花费几分钟的时间，而注销一个账户则可以没有时间限制，因为从顾客的角度来看，所关心的是钱都回到了自己的手中(等同于取款)。从银行的角度来看，账户可以在营业时间之后，或者是在月末处理账户时才删除。

在考虑管理账户的数据库系统所使用的数据结构时可以看出，一个不太考虑删除代价、查找效率极高并且具有适度插入效率的数据结构，应该符合上述问题所要求的资源限制。通

过唯一的账号很容易取得账户记录(有时称之为“精确匹配查询”)。符合这些要求的数据结构就是在 9.4 节描述的散列表。散列表具有非常快的精确匹配搜索速度。当修改操作不影响记录长度时，记录可以很快地修改。散列表也支持新记录的高效率插入，还能支持高效率删除，不过删除得太多会导致其他操作性能降低。然而，散列表可以定期重组，以便把系统还原到最高效率状态。这种重组应该脱机进行，以免影响 ATM 业务。

例 1.2　有一家公司正在开发一个数据库系统，该系统包含了美国城镇的信息。因为美国有数以千计的城镇，所以该系统应该允许用户根据城镇名字来查找某个地方的信息(这是精确匹配查询的另一个例子)。用户还应该能用诸如地理位置或人口数量这样的城镇特性的某个值或者某个范围值来查找与之相匹配的所有地方，这样的查询称为“范围查询”(range query)。

一个合理的数据库系统必须能够足够快地给出用户查询结果，查询所需要的时间应该保证在一般用户能够忍受的范围内。对于一个精确匹配查询，秒级等待是可以容许的。如果数据库要提供对范围查询的支持，即满足查询条件的结果可以是许多城市，那么整个查询操作花费的时间可以长一些，甚至可以到分钟级。要满足这个要求，就必须支持成批处理在查询结果范围内的城市信息，而不是一个接一个地依次处理每个城市的信息，从而高效率地处理范围查询。

在前面的一个例子中建议使用的散列表对本例的城市数据库系统就不太合适，因为它不能执行有效率的范围查询。虽然 10.5.1 节介绍的 B^+ 树支持大型数据库，也支持数据记录的插入和删除，并支持范围查询，但是数据库一旦建立后就不能再改变了，例如作为商品销售的、存放在 CD-ROM 上或通过 Web 访问的电子地图程序。而在 10.1 节中描述的简单线性索引表要更为有效一些。

1.2　抽象数据类型和数据结构

前面使用了术语“数据项”和“数据结构”，却没有给出它们的确切定义。这一节将介绍术语，并描述选择数据结构的三个步骤中的设计过程。

类型(type)是一组值的集合。例如布尔类型由 **true** 和 **false** 这两个值组成。整数也构成一个类型，一个整数是一个简单的类型，因为它的值不包含子结构。一个银行账户记录一般包含多项信息，例如姓名、地址、账号和余额，这样的记录是聚合类型(aggregate type)或组合类型(composite type)的一个例子。数据项(data item)是一条信息或者其值属于某个类型的一条记录，数据项也可说成是数据类型的成员(member)。

数据类型(data type)是指一个类型和定义在这个类型上的一组操作。例如，一个整数变量是整数数据类型的一个成员，而加法是定义在整数数据类型之上的操作的一个例子。

数据类型的逻辑概念与它在计算机程序中的实现有很重要的区别。例如，线性表数据类型有两种传统的实现方式：链表(linked list)和数组(array-based list)。因此，可以在链表或者数组之间选择一种来实现线性表数据类型。但是术语“数组”(array)是一个模棱两可的概念，因为它既可以指一种数据类型，又可以指一种实现方式。“数组”在计算机程序设计中常用来指一块连续的内存空间，每一个内存空间存储一个固定长度的数据项。从这个意义上来说，数组是一个物理数据结构。然而，数组也能够表示一个由一组(通常是结构相同的)数据项组

成的逻辑数据类型，每一个数据项由一个特定的索引号(即数组中的下标)来标识。这样来看，数组可以采用多种不同的方法来实现。例如，12.2 节中描述了用来实现一个稀疏矩阵(只有极少非零元素的大型二维矩阵)的数据结构，其实现就和传统的占用连续内存空间的数组大不相同。

抽象数据类型(abstract data type，ADT)是指数据结构作为一个软件构件的实现。ADT 的接口利用一个类型和这个类型上的一组操作来定义，每一个操作由它的输入和输出定义。一个 ADT 并不指定数据类型是如何实现的，这些实现细节对于 ADT 的用户是隐藏的，并且通过封装(encapsulation)这个概念来阻止外部对它的访问。

数据结构(data structure)是 ADT 的实现。在诸如C++ 之类的面向对象语言中，ADT 和它的实现共同组成了“类”(class)。同 ADT 联系在一起的每一个操作均由一个成员函数(member function)或方法(method)来实现。定义数据项所需要的存储空间的变量称为“数据成员”(data member)。“对象”(object)是类的一个实例，即它在一个计算机程序的执行期间创建，并占用一些存储空间。

术语“数据结构”常指存储在计算机内存中的数据。与其相关的术语文件结构(file structure)常指外存储器(如磁盘驱动器、CD)中数据的组织。

例 1.3 整数的数学概念和施加到整数的运算构成一个数据结构。C++ 的变量类型 **int** 就是对这个抽象整型的一种物理实现。**int** 变量类型加上定义在 **int** 变量之上的操作，就构成了一个 ADT。遗憾的是，由于 **int** 变量有一定的取值范围，所以它对这个抽象整型的实现并不完全正确。如果无法接受这些限制，就必须引进其他的 ADT“整型”定义，并对相关操作采用新的实现方法。

例 1.4 一个整数线性表的 ADT 应包含下列操作：

- 把一个新整数插入到线性表的结尾；
- 如果线性表为空，则返回 **true**；
- 重新初始化线性表；
- 返回线性表中当前整数的个数；
- 删除线性表中特定位置上的整数。

通过上述描述，每个操作的输入/输出都清晰可见，但是线性表的实现还未详细说明。

对于使用了同一个 ADT 的两个应用程序，可能存在一个应用程序使用的 ADT 特殊成员函数比另一个多的情况，或者这两个应用程序对不同的操作有不同的时间需求。正是由于应用程序的需求存在差异，才使得一个给定的 ADT 或许有多个实现形式。

例 1.5 对于基于磁盘的大型数据库应用，两种普通的实现方法是散列表(见 9.4 节)和 B^+ 树(见 10.5 节)。这两种方法都支持高效率的记录插入和删除操作，也都支持精确匹配查询。然而，在精确匹配查询方面，散列表比 B^+ 树更为有效。另一方面，B^+ 树能够执行范围查询，而散列表在范围查询方面则非常低效。因此，如果数据库应用局限于精确匹配查询则首选散列表。反过来，如果应用程序要求支持范围查询，则 B^+ 树成为首选。尽管执行效率不一样，但是这两种实现方法都可以解决类似的问题：更改和检索大量的记录。

ADT 的概念甚至还有助于在处理非计算应用问题时，集中于关键问题。

例 1.6　驾驶汽车的主要操作有控制方向、加速和刹车。几乎所有汽车都通过转动方向盘控制方向，踩油门加速，踩车闸刹车。汽车的这种设计可以看成是具有“控制方向盘”、“加速”和“刹车”操作的一个 ADT。两辆汽车可以利用截然不同的方式实现这些操作，但是大多数司机都能够驾驶许多不同的汽车，因为 ADT 提供了一致的操作方法，不需要司机去了解任何特定发动机或驱动设计的特别之处，其差异被有意地隐藏起来。

任何成功的计算机科学家都懂得一个重要原理：将复杂的问题抽象化。ADT 的概念就是这样的一个例子。计算机科学的主题是问题的复杂性和处理它的技术。为了解决复杂性，人们首先给一个物体或概念集合赋予一个称号(label)，然后用这个称号代替其实体来执行有关操作。有一位心理学家称这样的一个称号为隐喻(metaphor)。一个特定的称号可能与其他信息或者其他称号有关，这个集合被依次分配给一个称号，从而形成概念和称号的层次。称号的这种层次结构能使人们重视主要问题而忽略不必要的细节。

例 1.7　称号“硬盘驱动器”指在某种类型的存储设备上处理数据的硬件集合。称号“CPU”指控制计算机指令执行的硬件。这两个称号及其他一些称号合起来从属于称号“计算机”。由于即使小型家用电脑也要由数百万个部件组成，所以在弄清楚计算机是怎样进行工作之前，一些抽象的形式是十分必要的。

考虑一个实现和处理 ADT 的复杂计算机程序。ADT 通过一种特定的数据结构在程序的某个部分得以实现，而在设计使用 ADT 的那部分程序时，只关心这个数据类型上的操作，而不关心数据结构的实现。因此，在思考一个复杂程序时，如果不能将它简化，就没有希望理解或者实现它。

例 1.8　考虑设计一个简单的存储在硬盘上的数据库系统。通常，这类程序中存储在磁盘上的记录是通过一个缓冲池来访问的(见 8.3 节)，而不是直接访问。变长记录会使用一个内存管理器(见 12.3 节)寻找磁盘中的合适位置来存放这个记录。多重索引结构(见第 10 章)的典型用途就是支持不同的记录访问方式。因此，会定义很多类，每个类有其职责和访问权限。用户的数据库查询操作会通过查找索引结构来实现。这个索引结构通过查询缓冲池来寻找记录。如果一个记录被插入或删除，该请求会传到内存管理器，内存管理器会与缓冲池交互，从而获得对磁盘文件的访问。对于一个编程者来说，要把这些复杂逻辑都存放在大脑中不太可能。设计和实现的唯一办法是使用抽象和隐喻。在面向对象的编程中，这种抽象是通过类之间的关系来实现的。

数据项有逻辑形式(logical form)和物理形式(physical form)两个方面。利用 ADT 给出的数据项的定义是它的逻辑形式，数据结构中对数据项的实现是它的物理形式。图 1.1 说明了数据类型的逻辑形式和物理形式之间的关系。实现一个 ADT 时，是处理相关数据项的物理形式。在程序中的其他地方使用 ADT 时，则涉及相关数据类型的逻辑形式。本书的一些章节侧重于对于给定数据结构的一种或各种物理实现，而另一些章节则用逻辑 ADT 来实现高层任务中的数据类型。

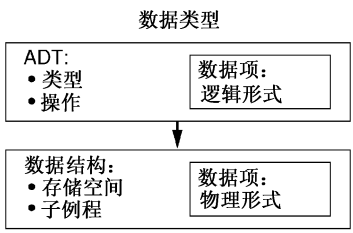

图 1.1　数据项、抽象数据类型和数据结构的关系

ADT 定义了数据类型的逻辑形式，数据结构是实现数据类型的物理形式。

例1.9 某个特定的C++集成环境可能会提供包含有一个线性表类的库函数。线性表的逻辑形式由定义该类的公共函数集和函数的输入/输出来定义。这可能是一个程序员对于该线性表的实现所能了解的所有情况,也是需要知道的全部内容。在类的内部,可能存在线性表的多种物理实现形式。4.1节描述了线性表的几种物理实现。

1.3 设计模式

比ADT更高层的抽象是对描述程序设计的抽象,即对象和类的相互关系。经验丰富的软件设计者懂得学习和重用设计模型来组合软构件,这样的技术称为设计模式(design pattern)。

对于一个重复发生的问题,设计模式使之具体化,并概括其重要的设计理念。设计模式最基本的目标是更快地将知识从有经验的设计者向新手程序员传递。另一个目标是让程序员相互进行更高效的交流。在讨论设计问题的时候,基于同一个与主题相关的技术词汇表,这样沟通起来会更加方便。

当一个特定的设计问题在一些情况中反复出现时,设计模式就应运而生。设计模式是为了解决实际问题的。设计模式有一点类似于模板:描述一个解决方案的框架,以及解决一个给定问题的具体细节。设计模式又有一点类似于数据结构:都有代价和收益,让使用者可以做出权衡。因此,一个给定的设计模式对于不同的应用可以有不同的变化,以做出当前情况下的权衡。

1.3.1 享元模式

享元(Flyweight)设计模式用于解决以下问题。假设有一个包含很多对象的应用。一些对象是相同的,即它们包含的信息和担任的角色一样。但是,这些对象会从不同的地方被访问到,而且从概念上来说它们也不一样。由于有太多的冗余信息,所以希望能够通过共享空间来减少内存消耗。以文本排版为例。字母“C”可以用一个对象来表示,该对象描述了这个字母的笔画和边界框。然而,作者并不想对文本中的每个“C”都建立一个对象。解决方法是对所有的“C”对象建立单一的对象。文本中每一个需要“C”的地方,比如字体、大小、字形等信息,都引用这个单一对象。对这个“C”对象的多个引用实例就称为享元。

采用树结构来描述页面中的文本排版。树的根结点表示整个页面。页面中有许多子结点,每一个表示一列。列结点的子结点表示行,行结点的子结点则表示每个字母。这种对字母的表示就是享元。享元包含对共享信息的引用和当前实例的特殊附加信息。例如,每个“C”的实例包含了一些共享信息,比如字体大小和笔画,同时附加了出现位置等与实例有关的特殊信息。

13.3节的PR四分树结构中将使用享元模式,用来存储结点对象。PR四分树中的许多叶结点代表空区域,因此唯一存储的信息就是表示空。这些相同的结点可以通过享元模式对一个类进行引用来节省内存,从而提高效率。

1.3.2 访问者模式

假设有一个简单的文本处理程序,采用一个对象树来描述页面布局,很可能会对上面的每个结点做一些特定的操作。本书的5.2节将讨论树的遍历,也就是按照特定顺序访问一棵

树中每个结点的过程。这个文本处理程序可能需要统计表示这个页面的对象树中结点的个数，在调试时还可能需要打印所有结点的列表。

为了实现上述两个功能，可以在每个功能中实现单独的遍历功能，但是一个更好的办法是编写一个通用的遍历函数，然后将需要进行的操作传入，以便在每个结点上执行，这种组织方式就称为访问者(Visitor)模式。访问者模式将在 5.2 节(树的遍历)和 11.3 节(图的遍历)用到。

1.3.3　组合模式

在处理一些有层级关系的对象和一组操作的关系时，有两种基本的方法。首先考虑传统的面向过程的方法。假设有一个描述页面组成的基类及其多层子类，用来表示具体的页面元素(页面、列、行、图表、字符等)，并且有一些可以应用到这些对象组合上的动作(例如将若干对象绘制到屏幕上)。在面向过程的设计中，每个动作被实现为一个方法，该方法接受一个基类的指针作为参数。对于每个动作，相应的方法都会遍历所有的对象，依次访问每个对象，每个动作的方法都会包括一些类似 **switch** 语句的结构，用来判断一个对象具体是哪个子类(页面、行、字符等)，并根据判断结果执行具体动作。可以借助访问者模式来简化代码，这样就只需要编写一次遍历代码，并将每个动作的行为写到访问者的动作中。但是，这样的访问者程序段需要包含逻辑判断，从而区分对象所属子类。

在页面编辑程序中，对于页面上的各种对象，只有有限的若干种动作需要执行。例如，完全绘制对象和只绘制对象轮廓这两种动作，如果需要增加一种可以应用到全部对象上的新动作，则并不需要修改已有动作的任何代码。然而在这个应用中，很少需要添加新的动作。另一方面，该应用存在很多种对象类型，并且会经常创建新的对象类型。遗憾的是，每添加一种新的对象类型，就需要将已有的每一种动作的代码都修改一遍。而且随着对象的增多，这些动作中的 **switch** 语句将会变得越来越长。

这里尝试另一种设计方法。在这种方法中，每种对象自身包含对各种可能动作的处理。每种子类都有执行各种动作的代码(如绘制全部内容和绘制轮廓)。之后如果要对一组对象执行一个操作，只需要调用这组对象中的第一个对象并且指定动作(即调用该对象的一个方法)。在这个表示页面布局的例子中，那些包含其他对象的对象(例如一个行对象会包含若干个字符对象)会调用每个子对象的相应方法。如果先要添加一种新的动作，就需要修改每种对象的代码，不过相对而言这种情况是很少见的。然而，添加一种新的对象(在该应用中，这比添加一种新绘制函数要常见得多)变得容易很多。只需要实现新添加的这种对象，执行每种动作的代码就可以了。

上面所说的第二种设计方法，称为组合(Composite)模式。在本书的 5.3.1 节中将看到组合模式的具体使用实例。

1.3.4　策略模式

在设计模式的最后一个例子中，将封装一系列可替换的方法，这些可变的方法会在一个更大的行为中执行。继续采用之前讨论的文字处理的例子，假设这个应用支持若干种输出设备，每种设备执行实际的绘制工作时，需要各自的代码。也就是说，文字对象都将被拆散成一些像素点或线，但是绘制这些点和线的机制根据设备的不同而不同。我们并不希望这些绘

制功能被编写到那些对象本身的类中。反之,可以给具体负责绘制的子程序传递一个方法或者一个类,而后者了解在具体的输出设备上绘制的细节。换言之,希望使用合适的“策略”来负责每个对象绘制的细节。因此,称这个方法为策略(Strategy)模式。

第 7 章将进一步讨论策略模式。届时会把一个类传给一个排序函数,这个类称为比较器(Comparator),该类了解对具体的待排序记录值应该如何比较大小。这样,排序函数在对记录排序时并不需要知道这个类型的记录是如何实现的。

了解设计模式的最大挑战之一,是有时两个设计模式之间只有细微的差别。例如,可能无法区分组合模式和访问者模式。组合模式处理的是将遍历的控制权交给树的结点还是树本身的问题。两种设计模式都利用了访问者模式,这将需要封装每个结点所执行的活动,以避免遍历功能的重复开发。

然而,策略模式是不是也在做相同的事情呢?访问者模式和策略模式的区别更加细微,它们的区别主要是在意图上。在这两个设计模式中,都会将一段行为予以封装,并作为一个参数传递。策略模式着重考虑的是将一大段过程中的某个可替换的子过程予以封装,以便几种不同的子过程可以相互替换。而访问者模式着重考虑的是将需要在一组对象上都要进行的动作予以封装,所以只需要实现一个通用的访问所有对象的方法,就可以完成一些不同的任务。

1.4 问题、算法和程序

程序设计人员总是需要与问题、算法和计算机程序打交道。这是三个不同的概念。

问题(problem):从直觉上讲,问题无非是一个需要完成的任务,即对应一组输入,就有一组相应的输出。问题的定义不能包含有关怎样解决问题的限制。只有在问题被准确定义并完全理解后,才能研究问题的解决方法。然而,问题的定义应该包含对任何可行方案所需资源的限制。对于计算机要解决的任何一个问题,总有一些直接的或者间接的资源限制。例如,任何计算机程序只能使用可用的主存储器和磁盘空间,而且必须在合理的时间内完成运行。

从数学角度来看,可以把问题看做函数。函数(function)是输入(即定义域,domain)和输出(即值域,range)之间的一种映射关系。函数的输入可以是一个值或者一些信息,这些值组成的输入称为函数的参数(parameter)。参数的一组选定值称为问题的一个实例(instance)。例如,一个排序函数的输入参数是整数数组。一个特定的整数数组,具有给定的数组长度,并且数组每个位置上的值也是确定的,这就是排序问题的一个实例。不同的实例可能产生相同的输出,但是对于问题的任何一个实例,只要把它作为给定的输入,则每次函数计算得到的输出必须相同。

这种将问题看做类似数学函数的概念可能并不符合人们对计算机程序行为的直观想法。众所周知,在两种不同的情况下,给程序输入同样的值可能得到两个不同的输出结果。例如,在 UNIX 命令行中键入命令“`date`”可以得到当前的日期。即使给出的是同样的命令,不同的日子所得到的日期也不同。然而,日期程序的输入显然比运行这个程序所键入的命令要多,日期程序执行的是一个函数。也就是说,在任意一个确定的日子里,对于一个完全确定的输入,正确运行日期程序只能得到唯一的结果。对于所有的计算机程序,输出完全由程序

的整个输入决定，即使是“随机数生成器”，也完全由它的输入决定(尽管一些随机函数生成系统从表面上看是一个不受用户控制的物理过程，是从一个物理随机数发生器接收随机输入)。程序和函数的关系将在 17.3 节进一步研究。

算法(algorithm)：算法是指解决问题的一种方法或者一个过程。如果将问题看做函数，那么算法就是把输入转化为输出。一个问题可以用多种算法来解决，一个给定的算法解决一个特定的问题(如计算一个特殊函数)。本书涉及了许多问题，对于其中几个问题给出了不止一种算法。对于重要的排序问题，差不多给出了 12 种算法。

知道一个问题有多种解法的好处在于：对于问题一个特例或问题的一类特殊输入，解法 *A* 可能比解法 *B* 有效；而对于另一个特例或者问题的另一类特殊输入，解法 *B* 可能又比解法 *A* 更有效。例如，有的排序算法适合于数目较少的序列(如果需要多次进行这个操作，那么这种算法是非常重要的)，有的算法适合于数目较多的序列，而有的算法则适合于可变长度的字符串。

一个算法应该包含以下几条性质：

1. 正确性(correct)。也就是说，它必须完成所期望的功能，把每一次输入转化为正确的输出。注意每一个算法都实现了某些功能，因为这些算法都把一组输入转化为一组输出(即使输出可能就是程序崩溃)。关键问题是，一个给定的算法是否实现了所期望的功能。
2. 具体步骤(concrete step)。一个算法应该由一系列具体步骤组成。“具体”意味着每一步所描述的行为对于必须完成算法的人或机器是可读的、可执行的。每一步必须在有限的时间内执行完毕。因此，算法就好像提供了通过一系列步骤来解决问题的“工序”，其中的每一步都是力所能及的，是否能够完成每一步，依赖于谁或者什么来执行这个工序。例如，烹饪书中关于小甜饼的制作方法对于指导一位厨师是足够具体的，但是对于一个自动小甜饼加工工厂却是不够的。
3. 确定性(no ambiguity)。下一步(通常是指算法描述中的下一步)应执行的步骤必须明确。选择语句(例如C++ 中的 **if** 和 **switch** 语句)是任何算法描述语言的组成部分，它允许对下一步执行的语句进行选择，但是选择过程必须是确定的。
4. 有限性(finite)。一个算法必须由有限步组成。如果一个算法的描述是由无限步组成的，人们就不可能将它写出来，也不可能将它作为计算机程序来实现。大多数算法描述语言(包括自然语言或“伪码”)均提供一些实现重复行为的方法(如循环)，例如 C++ 中的 **while** 和 **for** 循环结构。循环结构具有简短的描述，但是实际执行的次数由输入来决定。
5. 可终止性(terminate)。算法必须可以终止，即不能进入死循环。

程序(program)：一个计算机程序被认为是使用某种程序设计语言对一个算法的具体实现。本书中几乎所有的算法都给出了全部程序或者部分程序。当然，由于使用任何一种现代计算机程序设计语言都可以实现同样的一个算法，所以可能有许多程序都是同一个算法的实现(尽管一些程序设计语言可能使得编程人员的工作容易一些)。本书在下面的内容中为了简化表达，常常混用“算法”和“程序”，尽管它们实际上是互相独立的两个概念。定义一个算法时，必须提供足够多的细节，以便必要时转化为程序。

算法必须可终止,这意味着不是所有的计算机程序都是算法。操作系统是一个程序,而不是一个算法。然而,可以把操作系统的各种任务看成是一些单独的问题(每个都有相应的输入和输出),每一个问题由一部分操作系统程序通过某个算法来实现,得到输出结果后便终止。

以上内容可总结为:问题是一个函数,或者是从输入到输出的一个映射。算法是一个能够解决问题的、有具体步骤的方法。算法步骤必须无二义性。算法必须正确、长度有限,必须对所有输入都能终止。程序是算法在计算机程序设计语言中的实现。

1.5 深入学习导读

第一部数据结构和算法方面的权威性著作是 Donald E. Knuth 所著的系列丛书 *The Art of Computer Programming*,其中第 1 卷和第 3 卷是有关数据结构的研究[Knu97,Knu98]。Robert Sedgewick 所著的 *Algorithms*[Sed11]采用现代百科全书的方式组织,如果已经掌握了数据结构和算法的基本原理,这本书比较容易理解。Udi Manber 所著的 *Introduction to Algorithms: A Creative Approach*[Man89]是一部优秀的、可读性较强的高级著作,它介绍了算法、算法设计和算法分析。Cormen、Leiserson 和 Rivest 合著的 *Introduction to Algorithms*[CLRS90]采用了百科全书式的组织方法,内容比较先进。Steven S. Skiena 所著的 *The Algorithm Design Manual*[Ski10]则提供了许多用在 Web 上的数据结构和算法的实现方法。

所有现代程序设计语言可以实现同样的算法(准确地讲,应该是任何一个对于某种程序设计语言是可计算的函数,对于其他任何具备标准功能的程序设计语言也是可计算的),这是可计算性理论的一条重要结论。James L. Hein 所著的 *Discrete Structures, Logic, and Computability*[Hei09]很早就在这方面进行了介绍。

许多人致力于计算机科学中的问题求解,实际上这正是它吸引许多人进入这个领域的地方。George Pólya 所著的 *How to Solve It*[Pól57]被认为是提高问题求解能力的经典著作。如果你想成为一名好学生(同样也是一名好的问题解决者),可以参考 Folger 和 LeBlanc 合著的 *Strategies for Creative Problem Solving*[FL95],Marvin Levine 所著的 *Effective Problem Solving*[Lev94],Arthur Whimbey 和 Jack Lochhead 合著的 *Problem Solving & Comprehension*[WL99],以及 Zbigniew 和 Matthew Michaelewicz 合著的 *Puzzle-Based Learning*[MM08]。

Julian Jaynes 所著的 *The Origin of Consciousness in the Breakdown of the Bicameral Mind*[Jay 90],仔细讨论了如何利用隐喻解决复杂性问题(complexity)。与计算机教育和编程相关的文献有 Dan Aharoni 的"Cogito, Ergo Sum! Cognitive Processes of Students Dealing with Data Structures[Aha00],其中讨论了如何从编程情境思维转变为更高级的(更具设计导向的)不受编程限制的思维。

数据结构可以满足高层次程序设计的需要,许多人学习数据结构是为了编写出更好的程序。要想使程序正确、有效地运行,首先这个程序对于自己和合作者应该是可理解的。Kernighan和 Pike 合著的 *The Practice of Programming*[KP99]介绍了怎样养成良好的程序设计风格。Frederick P. Brooks 所著的经典之作 *The Mythical Man-Month: Essays on Software Engineering*[Bro 95]介绍了编写大程序的困难,该书十分出色,而且富有娱乐性。

最后,要成为一个成功的C++ 程序设计人员,手边应该有几本好的参考书。最标准的

C++ 参考书是 Bjarne Stroustrup 所著的 *C++ Programming Language*[Str00]，以及 Ellis 和 Stroustrup 合著的 *Annotated C++ Reference Manual*[ES90]，后一本书提供了更深层次的内容。几乎没有哪一个C++ 程序员不会阅读 Stroustrup 的书，因为他的书对该语言进行了准确的描述，并包含大量有关面向对象设计原则的内容。遗憾的是，该书没怎么介绍C++ 编程。由 Patrick Henry Winston 所著的 *On to C++*[Win94]是一本介绍C++ 语言基本内容的好书，而由 Deitel 所著的 *C++ How to Program*[DD08]则是一本很好的讲解C++ 编程的教材。

在掌握了程序的编写结构之后，接下来就应该熟悉程序设计。在任何学科中，要学会一流的设计都是一件困难的事情，而优秀的面向对象软件设计又是最难学的设计之一。对于初学设计者，可以通过学习已知和好用的设计模式来跳过这个学习过程。设计模式的经典参考书是 *Design Patterns*：*Elements of Reusable Object-Oriented Software*[GHJV95]，该书由 Gamma、Helm、Johnson 和 Vlissides 合著(遗憾的是，这是一本非常难理解的书，因为概念非常难)。很多网站也讨论了设计模式，并且有设计模式的相关书籍的指南。值得一读的另外两本讨论面向对象软件设计的书是由 Dennis Kafura 所著的 *Object-Oriented Software Design and Construction With C++*[Kaf98]和 Arthur J. Riel 所著的 *Object-Oriented Design Heuristics*[Rie96]。

1.6　习题

本章与本书其他章节的习题的不同之处在于，本章大部分习题在后面的章节中都得到了解答。但是请不要到后面的章节中寻找答案，这些习题的目的就是请读者思考一些后面将要讨论的问题，并利用当前所知道的知识尽可能地回答。

1.1　从以前用过的程序中找出一个慢得无法接受的程序，找出使程序运行速度变慢的个别操作和使程序运行得足够快的其他基本操作。

1.2　大多数程序设计语言有内建的整数数据类型。在正常情况下这种表示方法有固定的长度(表示一个整数所用的位数)，因此限制了整型变量的大小。请给出一种无长度限制(除计算机可用内存的限制之外)的整数表示方法，这样存储的整型变量大小就不受限制了。请用这种表示方法简要地说明如何实现加、乘、指数操作。

1.3　为字符串定义一个 ADT，要求包含字符串的一般操作，每一个操作定义为一个函数，每一个函数由它的输入、输出来定义。然后定义字符串的两个不同的物理实现。

1.4　为一个整数线性表定义一个 ADT。首先确定该 ADT 应该提供的功能，这可以参见例 1.4。然后在C++ 中利用抽象类声明来定义该 ADT，并写出函数、函数参数和返回类型。

1.5　简要概括整型变量在计算机上是如何表示的(如果对这些不熟悉，可以在计算机科学入门教材或者“数字逻辑”教材中查阅“反码”和“补码”的定义)。为什么整数的这种表示方法符合 1.2 节中所定义的数据结构？

1.6　为二维整数数组定义一个 ADT，详细准确地说明可以在此数组上完成的操作。然后试着将它用于一个 1 000 行、1 000 列的数组，其中非零元素远少于 10 000 个。为这个数组设计两种不同的实现方法，使它们比使用标准的二维数组所需要的 100 万(1 000 × 1 000)个位置的实现方法节省空间。

1.7　假设要实现一个排序算法，编写一个能实现通用功能的程序。也就是说，不用实现定义记录或者关键码的类型。描述设计这样一个简单排序算法的步骤(比如插入排序，或其他熟悉的排序算法)，以及如何支持通用性。

1.8　假设要实现一个在数组中顺序查找的简单方法。希望这个查找算法越通用越好。也就是说，算法需要支持任意记录和关键码类型。请通过对查找函数进行通用化修改来支持这个目标。假设

这个函数会在同一个程序中的不同数据类型上被使用多次，并且要查找的关键码也有可能是不同的。例如，一个学生的成绩记录可能会通过邮编、姓名、补助或成绩来查找。

1.9 每一个问题都有一个算法吗？

1.10 每一个算法都可以写出一个C++ 程序吗？

1.11 设计一个在家用计算机上运行的拼写检查程序，它能快速处理少于 20 页的文献。假设这个程序有一个大约 20 000 个单词的以 ASCII 码形式存储的字典。这个字典必须实现的原语操作是什么？每一个操作的合理时间限制是多少？

1.12 假设需要设计一个包含美国城镇信息的数据库服务系统，具体功能在例 1.2 中已经描述过。建议给出两个可能的实现方法。

1.13 假设有一组记录，按照每个记录都包含的一些关键码字段排序。给出两种不同的方法搜索有特定关键码值的记录，你认为哪一种更好，为什么？

1.14 怎样比较两种对一组整数进行排序的算法？特别是

(a)作为比较两种排序算法的基础，使用哪种代价度量比较合适？

(b)在这些代价度量方式下，使用哪种测试方法来判断这两种算法的性能？

1.15 编译器和文本编辑器的一个普遍问题是判断一个字符串中的圆括号(或其他括号)是否平衡并恰好匹配。例如，字符串“((())())()”中的圆括号平衡且恰好匹配，但是字符串“)()(”中的圆括号不平衡，字符串“())”中的圆括号不匹配。

(a)给出一个算法，当字符串中的圆括号恰好平衡且匹配时返回 **true**，否则返回 **false**。提示：从左到右扫描一个合法的字符串，保证任何时候所遇到的右圆括号不比左圆括号多。

(b)给出一个算法，如果字符串中圆括号不平衡或者不匹配，则返回字符串中第一个非法圆括号的位置。也就是说，如果发现一个多余的右圆括号，则返回其位置；如果有多个左圆括号，则返回第一个多余的左圆括号的位置；如果字符串平衡且恰好匹配，则返回 -1。

1.16 一个图由顶点集和边集组成，每条边连接两个顶点，任意一对顶点间只能连接一条边。至少给出两种不同的方法，表示由图的顶点和边定义的连接关系。

1.17 设想自己是一家大公司的送货员，要处理 1 000 张发货单，每张发货单为一页纸，在右上角写有号码。这些发货单必须按号码从小到大进行排序。尽可能多地写出完成这些发货单排序功能的不同实现方法。

1.18 如何对 1 000 个整数的数组按值从小到大进行排序？至少写出 5 种不同的实现方法。不必使用 C++ 或伪码来编写算法，只需要用语句来描述就可以了。

1.19 考虑一个寻找(未排序)数组中最大值的算法。然后再考虑在数组中寻找次大值的算法。哪个更难实现？哪个需要更长的运行时间(利用比较的次数来衡量)？现在，再考虑在数组中寻找第三大的值。最后，考虑在数组中寻找中位数的算法。这些问题中的哪个问题最难？

1.20 一个无序数组支持常数时间内的插入算法，即在末尾插入一个新的元素。遗憾的是，寻找值为 X 的元素需要在无序数组中进行顺序查找，这个算法平均需要遍历数组中一半元素。另一方面，一个有 n 个元素的有序数组可以在 $\log n$ 时间内进行二分法检索。遗憾的是，在这样一个有序数组中插入元素却需要很多时间。如何设计一个数据结构，使得插入和查找都能在 $\log n$ 时间内完成？

第 2 章　数学预备知识

本章介绍本书所使用的数学符号、背景知识和方法，主要用于复习和参考。在后面的章节中遇到不熟悉的符号或数学方法时，可以学习本章的相关部分。

2.7 节介绍的估计(estimate)对于许多读者来说可能比较陌生。估计不是一种数学方法，而是一种以前可能没有遇到过的工程上的通用技巧。估计对于计算机科学家的设计工作有很大用途，因为对于任何可能方案，如果它的估计资源需求超出了问题的资源限制，就可以立即放弃，将这些时间用于分析更有希望的解决方案。

2.1　集合和关系

数学意义上的集合概念在计算机科学上有广泛的用途。集合理论的符号和技术不仅用来描述和实现算法，与集合有关的抽象还有助于算法的清晰和简化。

集合(set)是由互不相同的成员(member)或者元素(element)构成的一个整体。成员取自一个更大的范围，称为基类型(base type)。集合的每个成员或者是基类型的一个基本元素(primitive element)，或者它本身也是一个集合。集合中没有重复的概念。取自基类型的每个值要么在集合内，要么不在集合内。例如，集合 **P** 由整数 7、11、42 组成，此时 **P** 的成员是 7、11 和 42，基类型是整型。

图 2.1 给出了表示集合和它们之间关系的常用符号。下面给出使用这种表示方法的实例。首先定义两个集合 **P** 和 **Q**：

$$\mathbf{P} = \{2, 3, 5\}, \qquad \mathbf{Q} = \{5, 10\}$$

则有 $|\mathbf{P}| = 3$(因为 **P** 有 3 个成员)，$|\mathbf{Q}| = 2$(因为 **Q** 有 2 个成员)。**P** 和 **Q** 的并(union)，记为 $\mathbf{P} \cup \mathbf{Q}$，是由 **P** 或 **Q** 中的元素组成的集合，为 $\{2,3,5,10\}$。**P** 和 **Q** 的交(intersection)，记为 $\mathbf{P} \cap \mathbf{Q}$，是由既在 **P** 中又在 **Q** 中的元素组成的集合，为 $\{5\}$。**P** 和 **Q** 的差(difference)，记为 $\mathbf{P} - \mathbf{Q}$，是由在 **P** 中但不在 **Q** 中的元素组成的集合，为 $\{2,3\}$。注意：总有 $\mathbf{P} \cup \mathbf{Q} = \mathbf{Q} \cup \mathbf{P}$，$\mathbf{P} \cap \mathbf{Q} = \mathbf{Q} \cap \mathbf{P}$，但通常 $\mathbf{P} - \mathbf{Q} \neq \mathbf{Q} - \mathbf{P}$，本例中 $\mathbf{Q} - \mathbf{P} = \{10\}$。注意集合 $\{3, 2, 5\}$ 和集合 **P** 是没有区别的，因为集合没有顺序的概念。同样，集合 $\{2, 3, 2, 5\}$ 与集合 **P** 也没有区别，因为集合没有重复元素的概念。

一个集合 **S** 的幂集(powerset)是指由 **S** 的所有可能子集组成的集合。例如，如果集合 $\mathbf{S} = \{a, b, c\}$，则 **S** 的幂集为

$$\{\emptyset, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\}$$

没有顺序的一组元素(像集合那样)，但是有重复的元素，这样的元素组称为“元包”(bag)[①]。为了区分元包和集合，使用方括号来括住元包的元素。例如，元包[3, 4, 5, 4]和元包[3, 4, 5]是

① 在数学中，与这里的元包相对应的对象有时也称为多重表(multilist)。但是在本书将术语“多重表”指为可以包含子表的表(见 12.1 节)。

不同的，但集合{3，4，5，4}与集合{3，4，5}没有区别，而元包[3，4，5，4]与元包[3，4，4，5]却是相同的。

{1，4}	由元素1和4组成的集合
{x | x是一个正数}	使用set形式定义的集合 示例：所有正整数集合
$x \in \mathbf{P}$	x是集合**P**的一个元素
$x \notin \mathbf{P}$	x不是集合**P**的一个元素
$\emptyset$	空集
$\lvert\mathbf{P}\rvert$	基数：集合**P**的大小或者集合**P**中元素的数目
$\mathbf{P} \subseteq \mathbf{Q}, \mathbf{Q} \supseteq \mathbf{P}$	集合**P**包含在集合**Q**中 集合**P**是集合**Q**的一个子集 集合**Q**是集合**P**的一个超集
$\mathbf{P} \cup \mathbf{Q}$	并集： 所有出现在集合**P**或者集合**Q**的元素
$\mathbf{P} \cap \mathbf{Q}$	交集： 所有既出现在集合**P**又出现在集合**Q**的元素
$\mathbf{P} - \mathbf{Q}$	差集： 所有只出现在集合**P**但不在集合**Q**的元素

图 2.1　集合的表示方法

序列(sequence)是指一个具有顺序的元素组，并且可以含有重复值的元素。序列有时也称为元组(tuple)或者向量(vector)。在一个序列中，存在着第 0 个元素、第 1 个元素、第 2 个元素这样的概念。采用一对尖括号括住元素来表示一个序列。例如，〈3，4，5，4〉是一个序列。注意序列〈3，5，4，4〉与序列〈3，4，5，4〉是有区别的，而这两个序列都与序列〈3，4，5〉不相同。

在集合 **S** 上的关系(relation) R 是指由 **S** 生成的有序对组成的集合。例如，如果 **S** 为 $\{a, b, c\}$，则

$$\{\langle a,c\rangle, \langle b,c\rangle, \langle c,b\rangle\}$$

是一个关系，并且

$$\{\langle a,a\rangle, \langle a,c\rangle, \langle b,b\rangle, \langle b,c\rangle, \langle c,c\rangle\}$$

是一个不同的集合。如果元组$\langle x, y\rangle$在关系 R 中，可以使用中缀表示法 xRy 来表示。实际上，人们常常使用到一些关系，例如在自然数上的小于运算符“ <”，有序对〈1，3〉和〈2，23〉满足该关系，但是〈3，2〉或〈2，2〉就在该集合中。往往不用有序对来表示这个关系，而通常使用中缀表示法来表示，例如写为 1 <3。

现在定义关系的如下属性。如果 R 是集合 **S** 上的一个二元关系，则有

- 如果对于所有的 $a \in \mathbf{S}$ 都有 aRa，则称 R 是自反的(reflexive)。
- 对于所有的 a，$b \in \mathbf{S}$，如果 aRb，则 bRa，就称 R 是对称的(symmetric)。
- 对于所有的 a，$b \in \mathbf{S}$，如果 aRb 且 bRa，则 $a = b$，就称 R 是反对称的(antisymmetric)。
- 对于所有的 a，b，$c \in \mathbf{S}$，如果 aRb 且 bRc，则 aRc，就称 R 是传递的(transitive)。

例如，对于自然数，“ <”是自反对称的和传递的(因为不可能同时满足 bRa 和 aRb)，“≤”是自反的、反对称的和传递的，“ =”是自反的、对称的(同时又是反对称的)和传递的。对于人来说，“同胞”关系是对称的和传递的。如果定义一个人是他自己的同胞，那么“同胞”关系就是自反的；如果定义一个人不是他自己的同胞，那么这个关系就不是自反的。

如果集合 **S** 上的关系 R 是自反的、对称的和传递的，则称 R 是一个等价关系(equivalence relation)。等价关系可以用来把一个集合划分成一些等价类(equivalence class)。如果两个元素 a 和 b 相互是等价的，就写成 $a \equiv b$。集合 **S** 的一个划分(partition)是由子集组成的集合，这些子集之间互不相交，所有子集的并集就是 **S**。在集合 **S** 上的等价关系把该集合划分为一些子集，每个子集中的元素是等价的。可以参考 6.2 节关于如何表达集合中的等价关系的讨论。11.5.2 节介绍了一个不相交集合的应用。

例 2.1　对于整数，“=”是一个等价关系，它把每个元素划分为一个独立的子集。也就是说，对于任何整数 a，满足以下三条：

1. 有 $a = a$ 和 $a \equiv a$；
2. 如果 $a = b$ 那么 $b = a$；
3. 如果 $a = b$ 且 $b = c$，那么 $a = c$。

当然，对于不同的整数值 a、b 和 c，不会出现 $a = b$、$b = a$ 或 $c = a$ 的情况。所以，“=”关系的对称性和传递性是无法验证的(显然不会出现这种情况)。但是，并没有违背对称性和传递性，因此“=”关系是对称的和传递的。

例 2.2　如果定义一个人可以是他自己的同胞，那么同胞关系就是划分由人组成的集合的一个等价关系。

例 2.3　可以使用取模函数(该函数将在下一节定义)来定义一个等价关系。对于整数集合，可以使用取模函数来定义这样一个二元关系：x 和 y 是该关系的成员当且仅当 $x \bmod m = y \bmod m$。因此，对于 $m = 4$，$\langle 1, 5 \rangle$ 属于该关系，因为 $1 \bmod 4 = 5 \bmod 4$。可以看到这样的取模定义了整数上的一个等价关系，并且这个关系可以用来把整数划分成 m 个等价类。这个关系是等价关系，因为：

1. 对于所有 x，都有 $x \bmod m = x \bmod m$；
2. 如果 $x \bmod m = y \bmod m$，则 $y \bmod m = x \bmod m$；并且
3. 如果 $x \bmod m = y \bmod m$ 并且 $y \bmod m = z \bmod m$，则 $x \bmod m = z \bmod m$。

如果一个二元关系是反对称的和传递的，那么这个关系就称为一个偏序(partial order)[①]。定义了偏序的集合称为部分有序集(partially order set)或偏序集(poset)。如果一个集合的两个元素 x 和 y 在给定的关系下有 xRy 或 yRx，则称 x 和 y 是可比的(comparable)。如果偏序中的每一对不同元素都是可比的，则该偏序称为全序(total order)或线性序(linear order)。

例 2.4　对于整数，关系“<”和“≤”各自都是一个偏序。“<”运算是一个全序，因为对任意一对整数 x 和 y 且 $x \neq y$，都有 $x < y$ 或 $y < x$。同样，“≤”也是一个全序，因为对任意一对整数 x 和 y 且 $x \neq y$，都有 $x \leqslant y$ 或 $y \leqslant x$。

例 2.5　对于整数的幂集，子集运算是一个偏序(因为这是反对称的和传递的)。例如，$\{1, 2\} \subseteq \{1, 2, 3\}$，然而集合 $\{1, 2\}$ 和 $\{1, 3\}$ 对于子集运算不是可比的，因为它们互不为对方的子集。因此，子集运算不是定义在整数集合上的幂集的一个全序。

---

① 不是所有书籍都这样定义偏序。作者本人在各种文献中至少见过三种不同的定义。本书选择的定义使得“<”和“≤”都能定义整数上的偏序，因为这样显得最为自然。

2.2 常用数学术语

计量单位:本书使用以下计量单位表示法,字节缩写为"B",位缩写为"b",千字节(2^{10} = 1 024字节)缩写为"KB",兆字节(2^{20}字节)缩写为"MB",十亿字节(2^{30}字节)缩写为"GB",毫秒(1 毫秒为1/1000 秒)缩写为"ms"。数字与以 2 为底数的缩写单位之间不应该有空格。因此,硬盘的大小为 25 兆字节(1 兆字节 = 2^{20}字节)就记为"25 MB"。如果单位是以 10 为底数缩写的,那么它与数字之间应该有空格。因此,2 000 位记为"2 Kb",而"2 Kb"代表2 048位。2 000 毫秒记为"2 000 ms"。本书中在计算大容量存储空间时一般都使用 2 作为底数,很少使用 10 作为底数。

阶乘函数(factorial function):阶乘函数 $n!$ 是指从 1 到 n 之间所有整数的连乘,其中 n 为大于 0 的整数。因此,$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$。特别是 $0! = 1$。阶乘函数随着 n 的增大而迅速增长。由于直接计算阶乘函数非常耗时,所以有时候使用一个公式来做近似计算是非常有用的。Stirling 近似公式 $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$,其中 $e \approx 2.718\ 28$(e 是自然对数的底数)①。可以看到当 $n!$ 增长得比 n^n 慢时(因为 $\sqrt{2\pi n}/e^n < 1$),对任意常数 c,它都增长得比 c^n 快。

排列(permutation):一个序列 **S** 的排列就是把这个序列 **S** 的成员按照一定的顺序组织起来。例如,整数 1 到 n 的排列可以把这些数字按照任意顺序放置。如果一个序列有 n 个不同的成员,那么这个序列就有 $n!$ 种不同的排列。因为排列中的第一个成员有 n 种选择方法,对于每一个选定的第一个成员,第二个成员有 $n-1$ 种选择方法,以此类推。有时候需要得到某个序列的一个随机排列(random permutation),也就是说,$n!$ 种可能排列中的任意一种被选中的概率都相同。下面是一个产生随机排列的简单C++ 函数。序列的 n 个值存储在数组 **A** 的元素 **A**[0]到 **A**[**n** − 1]中,函数 **swap**(**A**, **i**, **j**)在数组 **A** 中交换元素 **i** 和元素 **j** 的值。函数 **Random**(**n**)返回一个 0 到 $n-1$ 之间的整数(有关 **swap** 和 **Random** 的更多信息请见附录),程序如下:

```
// Randomly permute the "n" values of array "A"
template<typename E>
void permute(E A[], int n) {
  for (int i=n; i>0; i--)
    swap(A, i-1, Random(i));
}
```

布尔变量(Boolean variable):布尔变量是一个只能取值为 **true** 或 **false** 的变量(C++ 中的 **bool** 型变量)。这两个值常常分别与值 1 和值 0 对应,这样的规定并没有任何特殊的理由。在实际编程中把 0 与 **false** 完全对应起来是不太妥当的,因为它们在逻辑上是两个不同类型的对象。

逻辑表达式(logic notation):有时会使用一个符号或逻辑表达式。$A \Rightarrow B$ 表示"A 蕴含 B"或者"如果 A,则 B"。$A \Leftrightarrow B$ 表示"A 当且仅当 B"或"A 等价于 B"。$A \vee B$ 表示"A 或 B"(在逻

① 符号"≈"表示"约等于"。

辑表达式和布尔表达式中都有用)。$A \wedge B$ 表示“A 和 B”。$\sim A$ 和 $\bar{A}$ 都表示“非 A”，即当 A 是布尔变量时表示对 A 的否定。

取下整和取上整(floor and ceiling)：实数 x 的取下整函数(记为$\lfloor x \rfloor$)返回不超过 x 的最大整数。例如，$\lfloor 3.4 \rfloor = 3$，与 $\lfloor 3.0 \rfloor$的结果相同。实数 x 的取上整函数(记为$\lceil x \rceil$)返回不小于 x 的最小整数。例如，$\lceil 3.4 \rceil = 4$，与$\lceil 4.0 \rceil$的结果相同，而$\lceil -3.4 \rceil = \lceil -3.0 \rceil = -3$。

取模运算符(modulus operator)：取模(mod)函数返回整除后的余数。有时在数学表达式中用 $n \bmod m$ 表示，在C++ 中取模运算符的表示为 **`n % m`**。从余数的定义可知，$n \bmod m$ 得到一个整数 r，满足 $n = qm + r$，其中 q 为一个整数，且$|r| < |m|$。因此，$n \bmod m$ 的结果一定在 0 到 $m - 1$ 之间，这里 n 和 m 都是正整数。例如，$5 \bmod 3 = 2$，$25 \bmod 3 = 1$，$5 \bmod 7 = 5$，$5 \bmod 5 = 0$。

还有一种计算 q 和 r 的方法。最常见的取模函数的数学定义是 $n \bmod m = n - m\lfloor n/m \rfloor$。这样，$-3 \bmod 5 = 2$。Java 和C++ 编译器通常会使用当前处理器的指令来进行整数计算。许多计算机采用截尾操作，也就是 $n \bmod m = n - m(\text{trunc}(n/m))$。在这个定义下，$-3 \bmod 5 = -3$。

遗憾的是，对于许多应用这不是用户所期望的结果。例如，许多散列系统会以散列表长为模，对关键码进行取模来计算散列值，期望的结果是一个满足条件的合法下标值，而不是一个负数。散列函数的实现必须保证结果永远是正的，或者当结果为负时加上散列表长。

2.3　对数

以 b 为底 y 的对数(logarithm)定义为使得 b 的某次幂等于 y 的那个指数，记为 $\log_b y = x$。因此，如果 $\log_b y = x$，则 $b^x = y$，并且 $b^{\log_b y} = y$。

编程人员经常使用对数，它有两个典型的用途。

例 2.6　*许多程序需要对一些对象进行编码，那么表示 n 个不同的编码至少需要多少位呢？答案为$\lceil \log_2 n \rceil$位。例如，如果要存储 1 000 个不同的编码，至少需要$\lceil \log_2 1\,000 \rceil = 10$位(10 位可以产生 1 024 个不同的可用编码)。*

例 2.7　*考虑从一个按值由低到高排序的数组中查找指定值所使用的二分法检索算法。二分法检索首先与中间元素进行比较，以确定下一步是在上半部分进行查找还是在下半部分进行查找，然后继续将适当的子数组分半，直到找到指定的值(二分法检索将在 3.5 节详细描述)。一个长度为 n 的数组被逐次分半，直到最后的子数组中只有一个元素。那么，二分法检索一共需要分多少次呢？答案是$\lceil \log_2 n \rceil$次。*

本书中用到的对数几乎都是以 2 为底的，这是因为数据结构和算法总是把事情一分为二，或者用二进制位来存储编码。本书中的所有 $\log n$，要么是表示 $\log_2 n$，要么表示渐近函数而不关心底数的具体值。任何不以 2 为底的对数都会把底数清楚地写出来。

对于任意正数 m、n、r 及任意正整数 a 和 b，对数有下列性质：

1. $\log(nm) = \log n + \log m$
2. $\log(n/m) = \log n - \log m$
3. $\log(n^r) = r \log n$
4. $\log_a n = \log_b n / \log_b a$

前两个性质表明两个数相乘(或相除)的对数,等于两个数分别取对数再相加(或相减)①。性质 3 是对性质 1 的推广。性质 4 表明对于变量 n 和任意两个整数变量 a 和 b,$\log_a n$ 与 $\log_b n$ 只相差常数因子 $\log_b a$,而与 n 的值无关。本书中的大多数运行时间分析都忽略常数因子。性质 4 表明这种分析与对数的底数无关,因为它们对整体开销只是改变了一个常数因子。注意 $2^{\log n} = n$。

在讨论对数时,容易与指数相混淆。性质 3 说明,$\log n^2 = 2 \log n$。那么对数的平方应该如何表示呢?应该记为$(\log n)^2$,也习惯记为 $\log^2 n$。同样,n 的对数的对数应该记为 $\log \log n$。

还有一种用在很少见的情况下的特殊表示法,这种情况即在得到一个小于等于 1 的值之前,应该对一个数进行多少次 log 运算,记为 $\log^* n$。例如,$\log^* 1\,024 = 4$,因为 $\log 1\,024 = 10$,$\log 10 \approx 3.33$,$\log 3.33 \approx 1.74$,$\log 1.74 < 1$,这正好是 4 次 log 运算。

2.4 级数求和与递归

大多数程序都具有循环结构。分析循环程序的运行时间开销时,需要把每次循环执行的时间累加起来,这就是一个级数求和(summation)的例子。简单地讲,级数求和就是把函数在一定范围内取的值加起来,一般采用下面的"$\sum$"表示法:

$$\sum_{i=1}^{n} f(i)$$

这个记号表示对作用于某个(整数)范围内的值 i 的函数 $f(i)$ 之值求和,表达式的参数和初值写在 $\sum$ 符号的下面。这里,记号 $i=1$ 表明参数是 i,其初值是 1。Σ 符号的上面是表达式 n,表示参数 i 的最大值。即当 i 从 1 变到 n 时对 $f(i)$ 的值求和,也可以写成:$f(1) + f(2) + \cdots + f(n-1) + f(n)$。嵌在文本句子中的时候,求和表示法也被写为 $\sum_{i=1}^{n} f(i)$。

给出一个级数求和,希望能用一个具有相同结果的代数表达式来代替,称为"闭合形式解"(closed form solution),用"闭合形式解"替换级数的过程称为级数求解。例如,级数求和 $\sum_{i=1}^{n} 1$ 就是把数值"1"累加 n 次(注意 i 从 1 变到 n)。由于 n 个 1 的和为 n,所以这个级数求和的闭合形式解为 n。下面给出本书中出现的所有级数求和公式及其闭合形式解。

---

① 这些性质是形成计算尺的基础。两个数相加可以看成把两个长度连接起来,测量它们的总长度,而相乘却不是那么容易实现的。但是,如果先把这些数转化为它们的对数,然后相加,最后对得到的结果取反对数,就可以得出相乘的结果(这正是对数的性质 1)。计算尺计算出来的是数的对数长度,只需要滑动木条把这些长度累加起来,最后对累加的结果取反对数,就可以得到正确结果。

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{2.1}$$

$$\sum_{i=1}^{n} i^2 = \frac{2n^3+3n^2+n}{6} = \frac{n(2n+1)(n+1)}{6} \tag{2.2}$$

$$\sum_{i=1}^{\log n} n = n\log n \tag{2.3}$$

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \quad 0<a<1 \tag{2.4}$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1} \quad a \neq 1 \tag{2.5}$$

作为式(2.5)的特例，有

$$\sum_{i=1}^{n} \frac{1}{2^i} = 1 - \frac{1}{2^n} \tag{2.6}$$

和

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1 \tag{2.7}$$

作为式(2.7)的推论，有

$$\sum_{i=0}^{\log n} 2^i = 2^{\log n+1} - 1 = 2n - 1 \tag{2.8}$$

最后，

$$\sum_{i=1}^{n} \frac{i}{2^i} = 2 - \frac{n+2}{2^n} \tag{2.9}$$

从 1 到 n 的倒数之和称为调和级数(Harmonic Series)，记为 $\mathcal{H}_n$，它的值介于 $\log_e n$ 到 $\log_e n+1$之间。确切地说，当 n 增长时，级数趋向于

$$\mathcal{H}_n \approx \log_e n + \gamma + \frac{1}{2n} \tag{2.10}$$

γ 是欧拉(Euler)常数，其值为 0.5772…。

这些等式中的大多数容易由数学归纳法证明(见 2.6.3 节)，但是数学归纳法不能帮助推出闭合形式解，它只能证明一个闭合形式解是不是正确的。推出闭合形式解的方法将在 14.1 节进行讨论。

递归算法的运行时间最易于用递归表达式来表示，因为它包括了运行递归调用的时间。递归关系(recurrence relation)用一个表达式定义了一个函数，这个表达式包括其本身的一个或者多个(更小的)实例。一些经典的实例包括阶乘函数的递归定义：

$$n! = (n-1)! \cdot n \quad n>1; \quad 1! = 0! = 1$$

另一个标准的递归例子是 Fibonacci 序列：

$$\mathrm{Fib}(n) = \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2), \quad n>2; \quad \mathrm{Fib}(1) = \mathrm{Fib}(2) = 1$$

从这个定义可以得到 Fibonacci 序列的前 7 个数是

$$1, 1, 2, 3, 5, 8, 13$$

这个定义包含两部分：Fib(n)的一般定义，初始情况 Fib(1)与 Fib(2)。同样，阶乘函数的定义也包括递归部分和初始情况。

递归关系经常用来计算递归函数的开销。例如，如果输入为 n，当 $n=0$ 或 $n=1$(初始情况)时，2.5 节的函数 **fact** 所要求的乘数应为 0，并且等于用 $n-1$ 调用 **fact** 的开销加 1。这可以用下面的递归公式来定义：

$$\mathbf{T}(n) = \mathbf{T}(n-1) + 1 \qquad n > 1; \quad \mathbf{T}(0) = \mathbf{T}(1) = 0$$

就级数求和而言，希望用它的闭合形式解来代替递归关系。一种方法是展开(expand)递归，即对于出现的每一个 **T**，在等号右边都用 **T** 的定义来替换。

例 2.8 如果按 $\mathbf{T}(n) = \mathbf{T}(n-1) + 1$ 展开递归，则得到：

$$\begin{aligned} \mathbf{T}(n) &= \mathbf{T}(n-1) + 1 \\ &= (\mathbf{T}(n-2) + 1) + 1 \end{aligned}$$

可以按照这样展开任意多步，目标是要找到某种模式，以便能够利用求和运算来重写递归。在这个例子中，可以注意到：

$$(\mathbf{T}(n-2) + 1) + 1 = \mathbf{T}(n-2) + 2$$

并且如果再展开，则有

$$\mathbf{T}(n) = \mathbf{T}(n-2) + 2 = \mathbf{T}(n-3) + 1 + 2 = \mathbf{T}(n-3) + 3$$

然后由 $\mathbf{T}(n) = \mathbf{T}(n-i) + i$，可以得到：

$$\begin{aligned} \mathbf{T}(n) &= \mathbf{T}(n-(n-1)) + (n-1) \\ &= \mathbf{T}(1) + n - 1 \\ &= n - 1 \end{aligned}$$

不能随便猜测一个形式但却不证明这是正确的闭合形式解。为了完成这个过程，必须使用归纳证明来验证所获得的这个闭合形式解是正确的(见例 2.13)。

例 2.9 下面是稍微复杂一点的递归：

$$\mathbf{T}(n) = \mathbf{T}(n-1) + n; \quad T(1) = 1$$

将这个递归展开几步后，可以得到：

$$\begin{aligned} \mathbf{T}(n) &= \mathbf{T}(n-1) + n \\ &= \mathbf{T}(n-2) + (n-1) + n \\ &= \mathbf{T}(n-3) + (n-2) + (n-1) + n \end{aligned}$$

进一步可以将这个递归写为下面的形式：

$$\begin{aligned} \mathbf{T}(n) &= \mathbf{T}(n-(n-1)) + (n-(n-2)) + \cdots + (n-1) + n \\ &= 1 + 2 + \cdots + (n-1) + n \end{aligned}$$

这等价于求和算式 $\sum_{i=1}^{n} i$，对于这个算式，前面已经描述过它的闭合形式解。

寻找递归关系的闭合形式解将在 14.2 节讨论。在第 14 章之前，递归关系会在本书中频繁使用，其所对应的闭合形式解和推出的过程会在使用的时候给出。

2.5　递归

如果一个算法调用自己来完成它的部分工作，就称这个算法是递归的(recursive)。这种方法要想取得成功，必须在比原始问题规模更小的问题上调用自己。总而言之，一个递归算法必须有两个部分：初始情况(base case)和递归部分。初始情况只处理可以直接解决而不需要再次递归调用的简单输入。递归部分包含对算法的一次或者多次递归调用，每一次的调用参数都在某种程度上比原始调用参数更接近初始情况。下面给出一个计算 $n!$ 的C++ 递归函数。对一个较小的 n，4.2.4 节将给出这个函数的执行过程。

```
long fact(int n) {       // Compute n! recursively
  // To fit n! into a long variable, we require n <= 12
  Assert((n >= 0) && (n <= 12), "Input out of range");
  if (n <= 1)  return 1; // Base case: return base solution
  return n * fact(n-1);  // Recursive call for n > 1
}
```

函数的前两行构成初始情况。如果 $n \leqslant 1$，那么初始情况计算出问题的一个解函数。如果 $n > 1$，**fact** 调用一个知道如何得到 $(n-1)!$ 的函数。当然，能够计算 $(n-1)!$ 的刚好是函数 **fact** 本身。但是应该知道这样一个算法到底需要多少代价。递归算法的设计总能使用下面的方法实现。首先写出初始情况，然后考虑通过组合一个或多个较小但是类似子问题的结果来解决原问题。如果编写的算法是正确的，那么当然可以依靠它来递归地解决规模更小的子问题。成功的秘诀在于：不要担心递归方法是如何解决子问题的。只要简单地接受它能正确地解决子问题，而且使用子问题的求解结果就能正确地解决原问题。还有什么方法能比这样更简单呢?

递归方法在日常生活的问题求解中没有类似的概念。因为它需要使用一种新的思维方式来考虑问题，所以这个概念很难掌握。为了有效地使用递归，必须强制自己尽量不用非递归方法来思考问题。只要在不超出递归调用的范围内分析递归过程，子问题就会迎刃而解。

阶乘函数的递归实现看上去并不需要那么复杂，因为用一个 **while** 循环就可以达到同样的效果。下面给出的另一个实例基于著名的“河内塔”(Tower of Hanoi)问题。它的实现有多个递归调用，它不是那么容易就能够使用 **while** 循环改写的。

河内塔问题首先给出 3 根柱子和 n 个圆盘，所有圆盘均在最左边的柱子上(记为柱 1)。各个圆盘之间大小不同，按照大的圆盘在下面的顺序依次往上堆放，如图 2.2(a)所示。问题是要通过一系列步骤把这些圆盘从最左边的柱子上移到最右边的柱子上(记为柱 3)。每一步只能把某个柱子最上面的一个圆盘移到另一个柱子的上面，圆盘移到哪根柱子上不受限制，但是任何一个圆盘都不能放到比它小的圆盘的上面。

怎样解决这个问题呢? 如果不努力去考虑细节，这个问题是非常容易的。只要考虑所有圆盘都必须从柱 1 移到柱 3 上，因此必须首先把最下面(最大)的圆盘移到柱 3 上。要达到这个目的，柱 3 必须是空的，而且柱 1 上只能有最下面的一个圆盘，因此其余的 $n-1$ 个圆盘只能在柱 2 上，如图 2.2(b)所示。这该如何实现呢? 假设 X 是一个函数，可以把柱 1 上面的 $n-1$ 个圆盘移动到柱 2 的上面，然后把柱 1 最下面的一个圆盘移动到柱 3 上；最后，再用函数 X 把其余的 $n-1$ 个圆盘从柱 2 移动到柱 3 上即可。在这两种情况中，“函数 X”只不过是一个调用更小问题的河内塔函数而已。

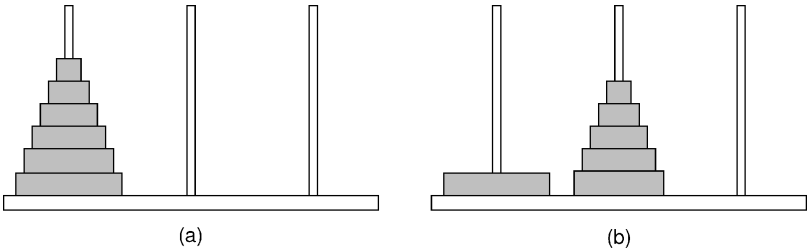

图 2.2 河内塔示例。(a)这个问题有 6 个圆盘时的初始状态;(b)得到解的过程中必须经过的一个中间步骤

成功的秘密在于河内塔算法为人们实现了这些操作。不必关心河内塔的子问题如何解决这些细节,只要做好两件事,问题就迎刃而解了。第一,必须有一个初始情况(如果只有一个圆盘该怎么做),以便递归过程不会永远进行下去。第二,对河内塔问题的递归调用只能用来解决更小的问题,而且只有一种正确的形式(一种满足河内塔问题初始定义的形式,假定对柱子适当地重命名)。

下面给出河内塔递归算法的一种实现,函数 **move(start, goal)** 把柱 start 最上面的圆盘移动到柱 goal 上。如果函数 **move** 的功能是打印它的参数值,那么递归调用 **TOH** 的结果将给出解决此问题的圆盘移动序列。

```
void TOH(int n, Pole start, Pole goal, Pole temp) {
  if (n == 0) return;              // Base case
  TOH(n-1, start, temp, goal); // Recursive call: n-1 rings
  move(start, goal);               // Move bottom disk to goal
  TOH(n-1, temp, goal, start); // Recursive call: n-1 rings
}
```

把递归作为一种主要用于设计和描述简单算法的工具,对于不熟悉它的编程人员是很难接受的。递归算法通常不是解决问题最有效的计算机程序,因为它包含递归函数调用,比其他替代选择诸如 while 循环等,所花费的代价更大。但是,递归通常提供了一种能合理有效地解决第 3 章所讨论的问题的算法(但不总是,请参看习题 2.11)。在需要的时候,可以对递归方法进行修改,以便更快地实现算法,这部分内容将在 4.2.4 节进一步地讨论。

很多数据结构是自然递归的。在这种情况下,可以定义为由自相似的部分组成。很多树结构都是这样的例子。因此,处理这样的数据结构的算法通常也是用递归来描述的。许多搜索和排序算法是基于"分治法"(divide and conquer)策略的,即通过这样做来找到解决方法:把问题分解成较小的(类似)子问题,再解决这些子问题,然后组合子问题的解以形成对原问题的解。这个过程通常用递归来实现。因此,递归在本书中扮演了一个重要的角色,并且本书也给出了很多的递归函数例子。

2.6 数学证明方法

解决任何问题都需要两个步骤:调查与推导(investigation and argument)。学生习惯于在教材和讲座中学习推导。但是如果想在学校中成绩突出(包括在以后的生活中),一个人应当需要在调查和推导两个部分都做得很好,并且能理解这两个部分的区别。为解决问题,必须做好调查,这意味着要努力探索直到找到解决方法。然后,要给客户一个答案(无论这个"客

户”是收作业和试卷老师，或是收报告的老板），还需要做好推导，即把解决方法阐述得清晰而简洁。推导技能包括良好的写作技巧——清晰而有逻辑地论证的能力。

熟悉标准的证明方法有助于这两个步骤，知道如何去写一个好的证明过程也很有帮助。首先，它很清楚地表达了你的思考过程，从而阐述了你的解释。然后，如果使用任何一种标准的证明方法，例如反证法和数学归纳法，那么你和读者都能很容易地理解这个结构。这能减少理解证明过程的复杂度，因为他们不需要从零开始来解析这些结构。

这部分简要介绍本书中最常用的三种证明方法：(1)推理法或直接证明法；(2)反证法；(3)数学归纳法。

2.6.1　直接证明法

一般来说，直接证明法只是一个“逻辑解释”。一个直接证明通常是作为推理演绎的一个论据。这是简单的逻辑推理。写出来一般是：“如果……那么……”，还可以用逻辑表达式写为“$P \Rightarrow Q$”。即使不想使用逻辑表达式，仍然能利用逻辑的基础理论来得到所需的论据。比如，如果希望证明 P 和 Q 是等价的，可以先证明 $P \Rightarrow Q$ 再证明 $Q \Rightarrow P$。

在一些领域中，证明是一系列从头到尾的状态变化。可以这样看待形式论断逻辑(formal predicate logic)：通过使用一组“逻辑规则”，把一个或一组公式变换到一个目标公式，类似于基础微积分中采用符号操作来求解积分问题，以及高中的几何证明题。

2.6.2　反证法

推翻一个定理或者命题的最简单方法就是找出一个反例。然而，支持一个定理的实例再多也不足以证明这个定理的正确性。反证法(proof by contradiction)是一种类似于使用反例进行证明的方法。要使用反证法证明一个定理，首先假设这个定理是错误的，然后找出由这个假设导致的逻辑上的矛盾。如果寻找矛盾的逻辑是正确的，那么唯一解决矛盾的方法就是纠正前面所做的关于定理错误的假设，即定理是正确的。

例 2.10　下面是使用反证法的简单证明。

定理 2.1　没有最大的整数。

证明：反证法。

第 1 步，反面假设：假设存在一个最大整数，记为 B。

第 2 步，由该假设导出矛盾：考虑 $C = B + 1$，因为 C 是两个整数的和，所以 C 也是整数，而且 $C > B$，因此导出矛盾。在推理过程中的唯一漏洞就是开始时的假设是错误的，从而得出结论：定理是正确的。

一个相关的证明技巧是反证法。可以通过 $\sim Q \Rightarrow \sim P$(非 Q 推出非 P)来证明 $P \Rightarrow Q$。

2.6.3　数学归纳法

数学归纳法(proof by mathematical induction)对许多定理都适用。数学归纳法还提供了一种考虑算法设计的有用方法，因为它促使人们从简单的子问题入手考虑问题的求解。数学归纳法有助于证明递归函数是否得到了正确的结果。理解递归是理解数学归纳法的一个重大步骤，反之亦然，因为它们在本质上是相同的。

在算法分析中,数学归纳法最重要的一个用处是作为测试一个假设的方法。正如 2.4 节中讲到的,在为一个求和或者递归寻找闭合形式解的时候,或许会首先猜测或者获取一个指定的公式就是正确解的证据。如果该公式确实正确,通常用数学归纳法就能很容易地加以证明。

假设 Thrm 是一个要证明的定理,用一个正整数参数 n 来表达 Thrm。数学归纳法表明,如果下面两个条件为真,那么对于参数 n 的任何值,Thrm 都是正确的(对于 $n \geqslant c$, c 是一个较小的常量):

1. 初始情况(Base Case):$n = c$ 时 Thrm 成立。
2. 归纳步骤(Induction Step):如果 $n-1$ 时 Thrm 成立,则 Thrm 对于 n 也成立。

证明初始情况通常很容易,只需要用一些较小的值(如 1)代替定理中的 n,然后应用一些简单的代数或简单的逻辑来证明定理即可。证明归纳步骤有时容易有时难。强归纳法(strong induction)的归纳步骤有所变化,即

2a. 归纳步骤:如果对于所有满足条件 $c \leqslant k < n$ 的 k,Thrm 都成立,则 Thrm 对于 n 也成立。

这种强归纳法,保证归纳过程的每一步都是正确的(即初始情况对于不同的值都成立),然后得出一个完满的证明。

把构成归纳法的两个条件合起来,表明 $n = 2$ 时 Thrm 成立是对事实 $n = 1$ 时 Thrm 成立的扩展,再把这个事实和条件 2 或 2a 结合起来就得出 $n = 3$ 时 Thrm 也成立,以此类推。因此,只要证明了这两个条件,就有 Thrm 对所有的 n 值都成立。

数学归纳法如此强大(而且许多初学者都觉得神秘)的原因是:可以充分利用 Thrm 对所有小于 n 的值都成立的假设来帮助证明 Thrm 对 n 也成立。这个假设称为归纳假设(induction hypothesis)。有了这个假设,归纳步骤的证明要比直接处理定理容易一些。建立在归纳假设之上,可以利用这些额外的归纳信息来求解问题。

递归和归纳法有相似之处,它们都是由一个或者多个初始情况终止的。递归函数能够调用自己而得到此问题更小的实例下的解。同样,归纳法证明依靠归纳假设的事实来证明定理。归纳假设并不是凭空出现的,只有定理正确时假设才是正确的,因此这在证明的情况中可信的处理方式。使用归纳假设与递归中调用子问题是一样的原理。

例 2.11 下面是一个用数学归纳法证明的例子,求前 n 个自然数的和 $\mathbf{S}(n)$。

定理 2.2 $\mathbf{S}(n) = n(n+1)/2$。

证明: 数学归纳法。

1. 检查初始情况。$n = 1$ 时,和显然为 1,公式的值为 $n = 1$, $1(1+1)/2 = 1$。因此,对于初始情况公式成立。
2. 提出归纳假设。归纳假设为

$$\mathbf{S}(n-1) = \sum_{i=1}^{n-1} i = \frac{(n-1)((n-1)+1)}{2} = \frac{(n-1)(n)}{2}$$

3. 利用 $n-1$ 的归纳假设,说明结果对于 n 也是正确的。归纳假设说明由于 $\mathbf{S}(n-1) = (n-1)(n)/2$,并且由于 $\mathbf{S}(n) = \mathbf{S}(n-1) + n$,可以用 $\mathbf{S}(n-1)$ 来替换:

$$\begin{aligned}\sum_{i=1}^{n} i &= \left(\sum_{i=1}^{n-1} i\right) + n = \frac{(n-1)(n)}{2} + n \\ &= \frac{n^2 - n + 2n}{2} = \frac{n(n+1)}{2}\end{aligned}$$

所以，由数学归纳法得出：

$$\mathbf{S}(n) = \sum_{i=1}^{n} i = n(n+1)/2$$

请仔细观察本例的推导步骤。首先把 $\mathbf{S}(n)$ 转换为本问题的较小规模：$\mathbf{S}(n) = \mathbf{S}(n-1) + n$。这非常重要，因为求解 $\mathbf{S}(n-1)$ 时，可以运用归纳假设，用 $(n-1)(n) = 2$ 来代替 $\mathbf{S}(n-1)$。再次，证明 $\mathbf{S}(n-1) + n$ 等于原来那个定理的右边等式，这就只是简单的代数问题了。

例 2.12　下面给出另一个使用数学归纳法进行简单证明的例子，它说明了要为数学归纳法选择一个合适的变量。想要证明前 n 个正奇数的和为 n^2。首先要有一种描述第 n 个奇数的方法，即 $2n-1$。这样就可以得出一个级数求和的定理。

定理 2.3　$\sum_{i=1}^{n}(2i-1) = n^2$

证明：由 $n=1$ 的初始情况得 $1 = 1^2$，结论正确。归纳假设为

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

现在利用归纳假设来说明定理对于 n 成立。前 n 个奇数的和等于前 $n-1$ 个奇数的和加上第 n 个奇数。在下面的第 2 行，使用归纳假设来替换部分和(即第 1 行中括号的内容)。之后就是一些简单的代数变换：

$$\begin{aligned}\sum_{i=1}^{n}(2i-1) &= \left[\sum_{i=1}^{n-1}(2i-1)\right] + 2n - 1 \\ &= [(n-1)^2] + 2n - 1 \\ &= n^2 - 2n + 1 + 2n - 1 \\ &= n^2\end{aligned}$$

因此由数学归纳法得出 $\sum_{i=1}^{n}(2i-1) = n^2$。

例 2.13　这个例子展示可以用数学归纳法来证明一个递归关系的闭合形式解是正确的。

定理 2.4　递归关系 $\mathbf{T}(n) = \mathbf{T}(n-1) + 1$；$\mathbf{T}(1) = 0$ 有闭合形式解 $\mathbf{T}(n) = n - 1$。

证明：由 $n=1$ 的初始情况得 $\mathbf{T}(1) = 1 - 1 = 0$，结论正确。归纳假设为：$\mathbf{T}(n-1) = n - 2$。结合递归定义和归纳假设，可以立即得到对于 $n > 1$ 有

$$\mathbf{T}(n) = \mathbf{T}(n-1) + 1 = n - 2 + 1 = n - 1$$

因此，由数学归纳法证明该定理正确。

例 2.14　下面的例子使用了数学归纳法，但不涉及级数求和，它说明了初始情况的一种更为灵活的定义。

定理 2.5　用 2 分和 5 分的邮票可以构成任何票面金额的邮资(对于金额≥4)。

证明：首先注意定理所定义的问题是对于金额≥4 的情况成立，而对于 1、2 和 3 都不成

立。所以用4作为初始情况,4分的金额能由两张2分邮票组成。归纳假设金额为$n-1$时可以由2分和5分邮票组合而成。现在用归纳假设说明如何推出金额为n的组合。金额为$n-1$的组合中或者包含5分邮票,或者不包含。如果包含5分邮票,则用3个2分邮票来代替。如果不包含,那么它的组合中至少包含两张2分的邮票(因为金额至少是4,而且只含有2分邮票),这种情况下用一张5分邮票代替两张2分邮票。无论哪种情况,都可以得到金额为n的邮资可以由2分和5分邮票组成。因此,由数学归纳法推导出定理正确。

例2.15 下面是一个使用强归纳法的实例。

定理2.6 对于所有大于1的整数n,n能被某个素数整除。

证明: $n=2$为初始情况,2可以被素数2整除。归纳假设对于所有的值a,$2 \leqslant a < n$,a能被某个素数整除。为了证明定理对于n成立,下面要考虑两种情况。如果n是一个素数,那么n可以被它自己整除。如果n不是素数,则$n=a \times b$,其中a、b均小于n且大于1。由归纳假设a可以被某个素数整除,因而n也可以被这个素数整除。因此,由数学归纳法可知,定理成立。

数学归纳法的下一个例子证明了一个几何定理。它也说明了一种数学归纳法证明技术,即原本有n个物体,为了使用归纳假设,要任意去掉一个物体。

例2.16 "双着色"(two coloring)是指已知一个区域集及两种颜色,要为每一个区域分配一种颜色,使得有共享边的区域不同色。例如,国际象棋棋盘就是一个"双着色"的图。图2.3显示了有三条线的平面的"双着色",假设两种颜色为黑和白。

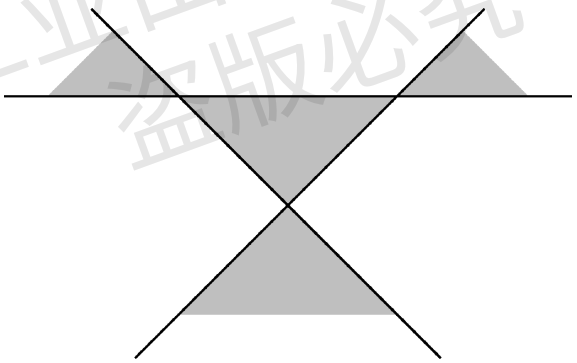

图2.3 对平面上由三条线形成的区域双着色

定理2.7 由平面上的n条直线形成的区域集可以实现"双着色"。

证明: 初始情况为平面上有一条直线。它将平面分为两个区域,一个区域着黑色,另一个区域着白色,这样就得到一个合法的"双着色"。归纳假设是$n-1$条直线形成的区域集可以被"双着色"。为了证明定理对n也成立,先考虑删去n条直线中的任意一条,由剩下的$n-1$条直线形成区域集。由归纳假设,这个区域集可以被"双着色"。现在把第n条直线放回来,它把平面分为两个半平面,每一个(互相独立)都实现了合法的"双着色"。但是,被第n条直线分割的区域却违反了"双着色"的规则。因此,把第n条直线一侧的所有区域的颜色都取反,现在被第n条直线分割的区域也符合"双着色"的规则了,因为这些区域在第n条直线一侧的部分为黑色,另一侧的部分为白色。因此,由数学归纳法可知,整个平面实现了"双着色"。

比较一下定理 2.7 和定理 2.5 的证明。对于定理 2.5，使用了金额为 $n-1$ 的一组邮票（由归纳假设必须有期望的性质），并由此“建立”一组金额为 n、满足限制条件的邮票，因而证明了存在一组金额为 n、满足限制条件的邮票。

对于定理 2.7，必须证明 n 条线的任意集合都满足限制条件。因此，策略是取 n 条线的一个任意集合，并且“缩小”该集合，以便得到一个满足限制条件的直线集合，因为得到的集合与归纳假设想匹配。然后，只需要反推原始的归纳过程满足限制条件即可。

对比一下，考虑如果试图建立从 $n-1$ 条直线的集合到 n 条直线的集合需要做的事情。很难证明 n 条直线的所有可能集合都被建立过程所覆盖。通过减少由 n 条线组成的一个任意集合中的直线数，就避免了这个问题。

这一节最后的例子展示了怎样使用数学归纳法来证明一个递归函数产生正确的结果。

例 2.17　证明函数 **fact** 确实计算阶乘。这个证明有两个不同的步骤，第 1 步证明该函数总能终止，第 2 步证明该函数返回正确的值。

定理 2.8　函数 **fact** 对任何值 n 都能终止。

证明：对于初始情况，当 $n \leqslant 0$ 时，**fact** 直接终止。归纳假设为：对于 $n-1$，**fact** 将终止。对于 n，有两种可能性。一种可能性是 $n \geqslant 12$，这时 **fact** 将直接终止，因为它在断言测试中失败。在另一种可能性下，即 $0 < n < 12$，**fact** 将做一次 **fact(n-1)** 的递归调用，根据归纳假设，**fact(n-1)** 必能终止。

定理 2.9　函数 **fact** 对 0 到 12 之间的任何值都计算阶乘。

证明：对于初始情况，当 $n=0$ 或 $n=1$ 时，**fact(n)** 都返回正确的值 1。归纳假设为：**fact(n-1)** 返回正确的值 $(n-1)!$。对于在合法范围内的任意值 n，**fact(n)** 返回 $n *$ **fact(n-1)**。由归纳假设，**fact(n-1)** $=(n-1)!$，并且因为 $n*(n-1)!=n!$，由此证明 **fact(n)** 产生正确的结果。

可以使用类似的过程来证明任何递归程序的正确性。通用的方法是：首先说明初始情况执行正确，然后使用归纳假设来表明递归步骤也产生正确的结果。除了正确性之外，还必须证明函数总能终止，可以使用数学归纳法来证明。

2.7　估计

可以从计算机科学训练中获得的最有用的生活技能之一，就是怎样进行快速估计。有时也称为“餐巾纸背面的计算”（back of the napkin calculation）或者“信封背面的计算”（back of the envelope calculation），这两个别名都表明只需要进行粗略的估计。估计技术是工程学课程的基本内容之一，但是在计算机科学中却常常被忽视。它不能代替对一个问题的严格细节分析，但是当严格的分析被保证实现时，可以用它来指示：如果最初的估计表明一个方法不可行，那么进一步的分析可能就没有必要了。

估计可以被形式化为以下三步：

1. 确定影响问题的主要参数；
2. 推导出一个与问题的参数有关的公式；
3. 选择参数值，由该公式得出一个估计解。

为了确保自己的估计是合理的,最好使用两种不同的方法进行估计。总体来说,如果想知道一个系统到底怎么样,可以直接对它进行估计,也可以估计系统的输入参数是什么(假设输入系统的参数,一定有相应的结果)。如果两种(相互独立的)方法得出同样的结果,那么你对于自己的估计能力一定会信心倍增。

估计时一定要保证计量单位的统一。例如,不要把英尺和英镑相加。一定要验证结果单位的正确性。时刻记住,一次计算的结果只与本次的输入参数有关。第3步输入的参数值越不确定,输出的值也就越不确定。然而,"信封背面的计算"通常只意味着得到一个大致正确或者50%正确的结果即可。因此在估计之前,应该规定一个误差允许的范围,如10%以内、50%以内等。一旦估计值落在误差允许范围内,就不用再管它。如果没有必要,就不用再费力去得到一个更精确的值。

例2.18 要存放共有100万页的书籍,需要多少个图书馆书架?我估计一本500页的书需要在图书馆书架上占1英寸①,因此100万页的书需要占200英尺②的书架空间。如果书架有4英尺宽,则需要50层。如果一个书架可以放5层书,则需要10个图书馆的大书架。为了得出这个结论,我需要估计每英寸的页数、图书馆书架的宽度和每个图书馆书架的层数。我的估计可能没有一个是准确的,但是我相信我的结论有50%正确的可能性。(写完此例后,我去图书馆查看了一些实际的书籍和书架。书架只有3英尺宽,但是一个书架有7层共21英尺的可存书宽度。因此在书架的容量方面,我只有10%的误差,基本上是正确的,这远比我所期望或需要的更高。我选的值一个太大,而另一个又太小,因此抵消了误差。)

例2.19 买一辆每加仑汽油可以行驶20英里③的汽车是否比买一辆每加仑行驶30英里但贵3 000美元的汽车更合算呢?普通汽车每年大约行驶12 000英里。如果油价是每加仑3美元,则低效率的汽车每年买汽油需要花1 800美元,而节油的汽车则需要花1 200美元。如果忽略诸如把3 000美元存入银行得到利息等问题,需要5年来弥补价格上的差异。此时,买主就必须决定是否价格是唯一的标准,是否5年的弥补时间可以接受。当然,车行驶的距离越长,弥补差异就越快,而且汽油价格的变化也会大大影响结果。

例2.20 当你在超市购物时,你是否估算过到收银台出口要付多少钱?有一种简单的估算方法,就是把购物篮中的每一项的价格都舍入到元,在你把每一件物品放到购物篮中时,在头脑中更新物品总额。这样得到的结果与实际需付款项之间只会有几元钱的差值。

2.8 深入学习导读

本章涉及的大多数问题都是离散数学的问题。有关这个领域的介绍请参见Susanna S. Epp所著的*Discrete Mathematics with Applications*[Epp10]。Graham、Knuth和Patashnik合著的*Concrete Mathematics: A Foundation for Computer Science*[GKP94],这是一本关于在计算机科学中有用的一些数学问题的高级处理方法的书籍。

IEEE Spectrum 1995年2月份期刊上的文章"Technically Speaking"[Sel95]讨论了本书中

① 1英寸=2.54厘米。
② 1英尺=0.305米。
③ 1英里=1.609千米。

用到的计算机存储单元的标准。Udi Manker 所著的 *Introduction to Algorithms* [Marn89]扩展了数学归纳法，从而使之成为一种设计算法的技术。

阅读 Eric S. Roberts 所著的 *Thinking Recursively* [Rob86]，可以得到有关递归的进一步知识。为了正确掌握递归，应该了解一下 LISP 或者其他程序设计语言，尽管没有必要编写一个 LISP 程序。特别是 Friedman 和 Felleisen 合著的“Little”系列(包括 *The Little LISPer* [FF89]和 *The Little Schemer* [FFBS95])能够指导读者如何考虑递归，也同时教授 LISP 语言，这是一本有趣的书。

Daniel Solow 所著的 *How to Read and Do Proofs* [Sol09]是一本有关书写数学证明的好书。为提高数学问题求解能力，可以参考 *The Art and Craft of Problem Solving* [Zei07]。Zeitz 也讨论了 2.6 节的三个证明技巧，以及求解问题中研究和推导的作用。

阅读 John Louis Bentley 撰写的 *The Back of the Envelope* 和 *The Envelope is Back* [Ben84, Ben00, Ben86, Ben88]的有关程序设计的姐妹篇著作，可以获得有关估计的技巧。James Gleick 所著的 *Genius*: *The Life and Science of Richard Feynman* [Gle92]，深刻地指出“信封背面的计算”对于计算机科学的重要性，它一点儿也不亚于原子弹的开发者对现代理论物理学的贡献。

2.9　习题

2.1　对于以下每个关系，解释该关系为什么满足或不满足自反、对称、反对称和传递性质。

(a)在自然人集合上的“兄弟”关系。

(b)在自然人集合上的“父子”关系。

(c)关系 $R=\{\langle x,y\rangle \mid x^2+y^2=1\}$，$x$ 和 y 是实数。

(d)关系 $R=\{\langle x,y\rangle \mid x^2=y^2\}$，$x$ 和 y 是实数。

(e)关系 $R=\{\langle x,y\rangle \mid x \bmod y=0\}$，$x$ 和 $y \in \{1,2,3,4\}$。

(f) 整数上的空关系 $\emptyset$(即没有任何有序对存在的关系)。

(g) 空集合上的空关系 $\emptyset$(即没有任何有序对存在的关系)。

2.2　对于以下每个关系，证明该关系是等价关系或者不是等价关系。

(a)对于整数 a 和 b，$a \equiv b$ 当且仅当 $a+b$ 是偶数。

(b)对于整数 a 和 b，$a \equiv b$ 当且仅当 $a+b$ 是奇数。

(c)对于非零有理数成员 a 和 b，$a \equiv b$ 当且仅当 $a \times b>0$。

(d)对于非零有理数成员 a 和 b，$a \equiv b$ 当且仅当 a/b 是一个整数。

(e)对于非零有理数成员 a 和 b，$a \equiv b$ 当且仅当 $a-b$ 是一个整数。

(f)对于非零有理数成员 a 和 b，$a \equiv b$ 当且仅当 $|a-b| \leqslant 2$。

2.3　指出以下关系哪些是偏序，并解释为什么是或为什么不是。

(a)父子关系。

(b)祖先关系。

(c)年纪大小关系。

(d)姐妹关系。

(e)$\{a,b\}$ 上的 $\{\langle a,b\rangle,\langle a,a\rangle,\langle b,a\rangle\}$。

(f)$\{1,2,3\}$ 上的 $\{\langle 2,1\rangle,\langle 1,3\rangle,\langle 2,3\rangle\}$。

2.4　在一个具有 n 个元素的集合上可以定义多少个全序？并加以解释。

2.5　为整数集合定义一个 ADT(记住集合没有重复元素和元素顺序的概念)。该 ADT 应该由可以在集合上控制成员、检查大小、检查元素是否存在等函数组成，每个函数用输入和输出来定义。

2.6 为整数元包定义一个ADT(记住元包可以包含重复元素,但没有元素顺序的概念)。该ADT应该由在元包上控制成员、检查大小、检查元素是否存在等函数组成,每个函数用它的输入和输出来定义。

2.7 为整数序列定义一个ADT(记住序列可以包含重复元素,并且支持每个成员有位置这个概念)。该ADT应该由在序列上控制成员、检查大小、检查元素是否存在等函数组成,每个函数用它的输入和输出来定义。

2.8 一个投资家将30 000美元放入证券市场,10年后账户有69 000美元。使用对数函数和反对数函数来表示计算年平均增长率的公式。然后使用该公式计算这支基金的年平均增长率。

2.9 不使用递归,改写2.5节的阶乘函数。

2.10 把2.2节用**for**循环编写的、产生随机排列的函数改写为一个递归函数。

2.11 下面是一个计算Fibonacci序列的简单递归函数。

```
long fibr(int n) { // Recursive Fibonacci generator
  // fibr(46) is largest value that fits in a long
  Assert((n > 0) && (n < 47), "Input out of range");
  if ((n == 1) || (n == 2)) return 1; // Base cases
  return fibr(n-1) + fibr(n-2);       // Recursion
}
```

这个算法的计算速度非常慢,调用**fibr**的总次数多于Fib(n)次。把它与下面的迭代算法进行比较:

```
long fibi(int n) { // Iterative Fibonacci generator
  // fibi(46) is largest value that fits in a long
  Assert((n > 0) && (n < 47), "Input out of range");
  long past, prev, curr;  // Store temporary values
  past = prev = curr = 1;    // initialize
  for (int i=3; i<=n; i++) { // Compute next value
    past = prev;             // past holds fibi(i-2)
    prev = curr;             // prev holds fibi(i-1)
    curr = past + prev;      // curr now holds fibi(i)
  }
  return curr;
}
```

函数**fibi**执行$n-2$次**for**循环。

(a)为什么?

(b)请解释为什么**fibr**比**fibi**慢得多。

2.12 将河内塔问题推广,初始状态每个圆盘可能在任何一个柱子上,只要没有大圆盘放在小圆盘的上面即可。编写一个递归函数解决这个问题。

2.13 修改2.5节河内塔的递归实现,返回解决问题所需要的移动步骤。

2.14 考虑下面的函数:

```
void foo (double val) {
  if (val != 0.0)
    foo(val/2.0);
}
```

这个函数通过每一次的递归调用向初始情况逼近。在理论上(即把**double**变量当成真正的实数),对于非零输入值**val**,这个函数是否能终止?在实际情况下(即计算机的实际实现)该函数是否能终止?

2.15 写出一个函数,对包含有n个不同整数值的数组,打印出该数组元素的所有排列组合。

2.16 写出一个递归算法,对由从1开始的前n个正整数值组成的集合,打印出该集合的所有子集。

2.17 两个正整数n和m的最大公约数(LCF)是能被n和m整除的最大整数。LCF(n,m)的最小值是1,最大值是m,假设$n \geqslant m$。两千年前,欧拉提出了一个高效的算法,算法基于$n \bmod m \neq 0$,$\mathrm{LCF}(n,m) = \mathrm{LCF}(m, n \bmod m)$。使用这个结论可以写出两个算法来寻找两个正整数的LCF。第一个版本使用迭代计算,第二个版本使用递归计算。

2.18　利用反证法证明素数的个数是无限的。

2.19　(a) 使用数学归纳法证明 $n^2 - n$ 总是偶数。

(b)请对 $n^2 - n$ 总是偶数这个问题给出一两句话的直接证明。

(c)证明 $n^3 - n$ 总能被 3 整除。

(d)$n^5 - n$ 是否总能被 5 整除，请解释你的答案。

2.20　证明 $\sqrt{2}$ 是无理数。

2.21　解释为什么下式成立：

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n} (n - i + 1) = \sum_{i=0}^{n-1} (n - i)$$

2.22　使用数学归纳法证明式(2.2)。

2.23　使用数学归纳法证明式(2.6)。

2.24　使用数学归纳法证明式(2.7)。

2.25　对于下面的级数，求和找出闭合形式解并证明(使用数学归纳法)这个解是正确的。

$$\sum_{i=1}^{n} 3^i$$

2.26　证明前 n 个偶数的和为 $n^2 + n$。

(a)通过假设前 n 个奇数的和是 n^2，进行证明。

(b)用数学归纳法进行证明。

2.27　计算 $\sum_{i=a}^{n} i$，a 是 1 到 n 之间的整数。

2.28　证明 $\mathrm{Fib}(n) < \left(\frac{5}{3}\right)^n$。

2.29　证明：当 $n \geqslant 1$ 时，

$$\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

2.30　下面的定理称为鸽笼原理(Pigeonhole Principle)。

定理 2.10　$n+1$ 只鸽子要在 n 个鸽笼中栖息，那么至少有一个鸽笼中有 2 只鸽子。

(a)用反证法证明鸽笼原理。

(b)用数学归纳法证明鸽笼原理。

2.31　考虑这样一个问题，在一个平面上有无限多条直线，这些直线中没有三条或者三条以上的直线在一个点相交，也没有两条直线平行。

(a)利用一个递归关系式表达 n 条直线形成的区域数目，并解释一下为什么你的递归关系式是正确的。

(b)通过扩展递归关系式，得到它的求和公式。

(c)给出求和公式的闭合形式解。

2.32　证明(用数学归纳法)：递归 $\mathbf{T}(n) = \mathbf{T}(n-1) + n$；$\mathbf{T}(1) = 1$ 有闭合形式解 $\mathbf{T}(n) = n(n+1)/2$。

2.33　展开下面的递归，以便找到一个闭合形式解，并用数学归纳法证明这个解是正确的。

$$\mathbf{T}(n) = 2\mathbf{T}(n-1) + 1 \qquad n > 0;\ \mathbf{T}(0) = 0$$

2.34　展开下面的递归，以便找到一个闭合形式解，并用数学归纳法证明这个解是正确的。

$$\mathbf{T}(n) = \mathbf{T}(n-1) + 3n + 1 \qquad n > 0;\ \mathbf{T}(0) = 1$$

2.35　假设有一个 n 位整数(以标准二进制表示)按照相同的概率在 0 到 $2^n - 1$ 之间取值。

(a)对于每一个比特位，它取 1 的概率是多少？取 0 的概率是多少？

(b)对于一个 n 位的随机整数,值为"1"的位平均有多少个?

(c)最左边一个为"1"的位所在位置的数学期望值是多少?也就是说,从最左边一位开始向右移动直到遇到第一个"1"时平均检查了多少位?

2.36 用公升度量,你的总体积有多大(也可以用加仑作为单位)?

2.37 一位艺术史学家有一个包含 20 000 幅全屏彩色图像的数据库。

(a)大约需要多少存储空间?存储这个数据库需要多少张 CD(一张 CD 的容量为 600MB)?请说出为推出结论你所做出的所有假设。

(b)现在假设你已经掌握了一种图像压缩技术,此时存储一幅图像只需要不压缩时所需存储空间的 1/10。如果压缩图像,整个数据库能放在一张 CD 上吗?

2.38 密西西比河每天的水流量大约是多少立方英里?请给出为了推出该结论你所做出的所有假设,不要查找答案或任何辅助事实。

2.39 分期付款购买房产时,你可以选择先付一些钱(称为"折扣点"),以获得一个较好的借款利率。假设你可以在两种 15 年期的抵押贷款中进行选择,一种为 8% 的利率,没有其他的费用,另一种为 $7\frac{3}{4}\%$ 的利率,但是要多预付房款总金额的 1%。如果选择低利率的抵押贷款,那么总房款 1% 的费用要多长时间才能平衡?另外,请更精确地估计一下,如果选择高利率的抵押,而且把等价的总房款 1% 的费用存入利率为 5% 的银行,那么考虑付款额和利息,需要多长时间才能够得到回报?不要使用纸或计算器来演算。

2.40 当你建造一个新房屋时,有时会收到"建造贷款",然后成为你信用卡账单中的一行。在建造后期,你把所有的建造贷款换成房屋抵押。在这段建造贷款的时期,你每个月只支付贷款利息。假设你的房屋建造计划于四月初开始,六个月后结束。假设总的建房开销是 300 000 美元,包括一开始以每月 50 000 美元增长的费用。建造贷款的利息为 6%。估计在建造贷款时期需要支付的总利息。

2.41 下面的问题用来测试有关计算机操作速度的知识。访问磁盘驱动器的时间通常是用毫秒(千分之一秒)或微秒(百万分之一秒)来度量的吗?RAM 访问一个字的时间是多于 1 微秒还是少于 1 微秒?如果计算机以最快速度不停地运行着,那么 CPU 一年中能执行多少条指令?不用纸或计算器,推出你的结论。

2.42 你家里所有的书加起来有 100 万页吗?你所在学校的图书馆的藏书总共有多少页?请说明你是怎样得到这个答案的。

2.43 本书中共有多少个单词?请说明你是怎样得到这个答案的。

2.44 100 万秒是多少小时?多少天?用心算回答这些问题。请说明你是怎样得到这个答案的。

2.45 美国有多少城市和小城镇?请说明你是怎样得到这个答案的。

2.46 从波士顿到旧金山,如果走路的话要走多少步?请说明你是怎样得到这个答案的。

2.47 一位男士开车去拜访他的亲戚。整个距离为 60 英里,他出发时的速度为 60 英里/小时。恰好行驶 1 英里后,他的旅行兴趣有所减少,所以立即减速到 59 英里/小时。再行驶 1 英里后速度降为 58 英里/小时。这样继续下去,每行驶 1 英里减速 1 英里/小时,直到走完全程。

(a)他到达亲戚家所需的时间是多少?

(b)如果速度随着距离均匀地连续减慢,行驶 1 英里正好总共减速了 1 英里/小时,那么他的旅行驾驶时间是多少?

第3章　算法分析

在完成了公司的合并计划之后，要花多长时间来处理公司的工资册呢？是否应该从 X 供应商或 Y 供应商那里购买新的工资管理程序？如果一个给定的程序执行得很慢，是因为该程序编得不好呢，还是因为它正在解决一个难解的问题？像这样的提问让人们想到了一个问题的困难程度，也会思考解决一个问题的两个或更多方法的相应效率。

本章将介绍算法分析的动机、基本符号和基本技术。将重点关注渐近算法分析(asymptotic algorithm analysis)，简称渐近分析(asymptotic analysis)。算法分析可以评估一个算法所消耗的资源。可以据此对解决同一个问题的两种或两种以上算法的代价加以比较，算法设计者也可以使用这种方法在真正实现算法之前判断一种算法是否会遇到资源限制问题。学习完本章之后，读者应该掌握以下知识点：

- 增长率(growth rate)的概念，即当问题的规模增大时，算法代价增长的速度。
- 增长率的上限和下限的概念，即怎么对简单程序、算法或问题的上下限做出估算。
- 能够区分一种算法(或程序)的代价和一个问题的代价。

本章最后还讨论了通过实验方式测算程序时间代价时可能遇到的一些实际问题，并介绍了通过代码调优来改善程序效率的一些原则。

3.1　概述

如何比较两种算法解决问题的效率呢？可以用源程序分别实现这两种算法，然后输入适当的数据运行，测算两个程序各自的开销。但是这种方法并不尽如人意。第一，编写两个程序来测算两种算法将花费较多的时间和精力，而至多只需要保留其中之一。第二，仅凭实验来比较两种算法，很有可能因为一个程序比另一个“写得好”，而使得算法的真正质量没有得到很好的体现。当程序员对算法有偏见时，尤其容易发生这种情况。第三，测试数据的选择可能对其中的一个算法有利。第四，你可能会发现即使是较好的那种算法也超出了预算开销，这意味着你不得不再重复一遍这样的过程——寻找一种新的算法，再编写一个程序实现它。但是，你又怎么知道存在能够满足预算开销的算法呢？很有可能这个问题就是很困难，对于任何一种算法的开销都不可能在预算之内。

有一种办法能够解决所有这些问题，那就是渐近分析。渐近分析可以估算出当问题规模变大时，一种算法及实现它的程序的效率和开销。这种方法实际上是一种估算方法，如果两个程序中的一个总是比另一个“稍快一点”，它并不能判断那个“稍快一点”的程序的相对优越性。但是在实际应用中，它被证明是很有效的，尤其是当科学家确定某种算法是否值得实现的时候。

运行速度通常是算法代价的一个关键方面，但是也不能片面地注重运行速度，而应该同时考虑其他因素，如运行该程序所需要的空间代价(包括内存和磁盘空间)。通常需要分析一

种算法(或者是实现该算法的一个程序实例)所花费的时间,以及一种数据结构所占用的空间。

许多因素都会影响程序的运行时间。有些因素与程序的编译和运行环境有关,例如计算机主频、总线和外部设备等。如果与其他用户共享计算机(或网络)资源,有时会使程序慢得像蜗牛爬行一样。程序设计使用的语言和编译系统生成的机器代码质量会产生很大的影响,编程人员用程序实现算法的效率也会在很大程度上影响运行速度。如果你要在一台指定的机器上,在给定的时间和空间限制下运行一个程序,以上这些因素都会对结果产生影响。但是,这些因素与两种算法或数据结构的差异无关。为了公平起见,同一个问题的两种算法所对应的两个程序,应该在同样的条件下用同一个编译器编译,在同一台计算机上运行。并且,两次编程所花费的精力也应该尽可能地相等,以使得算法的实现"等效"。做到以上几点,上面提到的那些因素就不会对结果产生影响,因为它们对每一个算法都是公平的。

如果你真想知道一种算法的运行时间,只考虑主频、编程语言、编译器之类的因素是不够的。从理论上来说,要在标准环境下测算一种算法的时间代价。然而事实上,只是在某台计算机上运行算法的载体。唯一的选择是选用另一种尺度来代替运行时间。

判断算法性能的一个基本考虑是处理一定"规模"(size)的输入时该算法所需要执行的"基本操作"(basic operation)数。"基本操作"和"规模"这两个名词的含义都是模糊不清的,而且要视具体算法而定。"规模"一般是指输入量的数目。例如,在排序问题中,问题的规模就可以很典型地用被排序的元素个数来衡量。一个"基本操作"必须具有这样的性质:完成该操作所需时间与操作数的具体取值无关。在大多数高级语言中,两个整数相加及比较两个整数的大小都是基本操作,而 n 个整数累加就不是基本操作,因为其代价(cost)依赖于 n 的值(即大小)。

例 3.1 下面是查找一维 n 元整数数组中最大元素的算法。该算法依次遍历数组中的所有元素,并保存当前的最大元素,称为"最大元素顺序搜索"。下面就是使用C++语言编写的程序:

```
// Return position of largest value in "A" of size "n"
int largest(int A[], int n) {
  int currlarge = 0; // Holds largest element position
  for (int i=1; i<n; i++)   // For each array element
   if (A[currlarge] < A[i]) // if A[i] is larger
      currlarge = i;        //   remember its position
  return currlarge;         // Return largest position
}
```

其中,问题的规模为 **A.length**,这些整数存放在数组 **A** 中,基本操作是把一个整数值与现有最大整数相比较。可以认为,这样检查数组中的某个整数所需要的时间就是一定的,与该整数的大小或其在数组中的位置无关。

因为影响时间代价的最主要因素,一般来说是输入的规模,经常把执行算法所需要的时间 $\mathbf{T}$ 写成输入规模 n 的函数,记为 $\mathbf{T}(n)$。注意,总是假设 $\mathbf{T}(n)$ 为非负值。把 **largest** 函数中检查一个元素所需要的时间记为 c。现在不考虑 c 的实际值,也不考虑变量 i 增值(这是处理数组中每一个元素都要做的工作)的时间,以及函数初始化 **currlarge** 所需要的一小部分额外时间。只想得到执行该算法的一个合理的近似时间。因此,运行 **largest** 函数的总时间可以近似地认为是 cn,因为一共需要 n 步检查工作,每一步需要

时间 c。**largest** 函数(或者广义上来说,即最大元素顺序搜索法)的时间代价可以用下面的等式来表示:

$$\mathbf{T}(n) = cn$$

这个等式表明了最大元素顺序搜索法时间代价的增长率。

例 3.2 把一个整数数组的第一个元素值赋给另一个变量,只要复制这个元素的值就可以了。可以认为完成这一功能所需要的时间是固定的,与这个元素的具体取值无关。在一台特定的计算机中,无论这个数组有多大(只要内存和定义的数组大小允许),复制该数组第一个元素值的时间总是确定的,记为 c_1。因此该算法时间代价的等式就是

$$\mathbf{T}(n) = c_1$$

输入规模 n 对运行时间不会产生影响,这称为常数运行时间(constant running time)。

例 3.3 看看下面的C++程序段:

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

如何计算这个程序段的运行时间呢?显然,随着 n 的增大,其运行时间也会增大。本例中的基本操作是变量 *sum* 的累加,可以认为该操作所需要的时间是一定的,记为 c_2(在此可以忽略初始化 *sum* 和循环变量 i 与 j 累加的时间。事实上,这些时间开销都可以计入 c_2)。要执行的基本操作总数为 n^2,因此,运行时间函数为:$\mathbf{T}(n) = c_2 n^2$。

算法的增长率(growth rate)是指当输入的值增长时,算法代价的增长速率。图 3.1 给出了一个运行时间函数的曲线,每个多项式反映一个程序或者一种算法的时间代价,图中显示了不同算法的增长率。标记为 $10n$ 和 $20n$ 的两个函数图像为直线,表达式为 cn(c 为任意正的常数)的增长率称为线性增长率(linear growth rate)或者线性时间代价(linear time cost)。这说明当 n 增大时,算法的运行时间以相同的比例增加。n 增大一倍,运行时间也增大一倍。如果算法的运行时间函数中含有形如 n^2 的高次项,则称为二次增长率(quadratic growth rate)。在图 3.1 中,标有 $2n^2$ 的那条曲线就代表二次增长率。标有 2^n 的曲线属于指数增长率(exponential growth rate),这是因为 n 出现在指数位置而得名。标有 $n!$ 的曲线也同样是指数增长的。

从图中可以看到,运行时间代价分别为 $\mathbf{T}(n) = 10n$ 和 $\mathbf{T}(n) = 2n^2$ 的两种算法,当 n 增加时有着天壤之别。当 $n > 5$ 时,相应的 $\mathbf{T}(n) = 2n^2$ 的算法已经很慢了,尽管 $10n$ 的系数比 $2n^2$ 的系数要大。比较标有 $20n$ 和 $2n^2$ 的两条曲线,还会发现,改变一个函数的常数系数只改变两个曲线的交点,当 $n > 10$ 时,对应于 $\mathbf{T}(n) = 2n^2$ 的算法比对应于 $\mathbf{T}(n) = 20n$ 的算法要慢。该图还表明,运行时间代价为 $\mathbf{T}(n) = 5n \log n$ 的曲线增长速度比 $\mathbf{T}(n) = 10n$ 和 $\mathbf{T}(n) = 20n$ 都稍快,但是又比 $\mathbf{T}(n) = 2n^2$ 慢。当 a、b 为大于 1 的任意正常数时,n^a 的增长速度比 $\log^b n$ 和 $\log n^b$ 快。最后还应该指出,即使 n 值很小,运行时间代价为 $\mathbf{T}(n) = 2^n$ 和 $\mathbf{T}(n) = n!$ 的算法时间开销也很大。注意,当 a, $b \geqslant 1$ 时,a^n 的增长速度比 n^b 快。

在图 3.2 中可以更深入地了解各种算法相应的增长率。该图显示了典型算法在某些典型输入值下出现的大多数增长率。再一次看到了增长率对算法消耗的资源有重大影响。

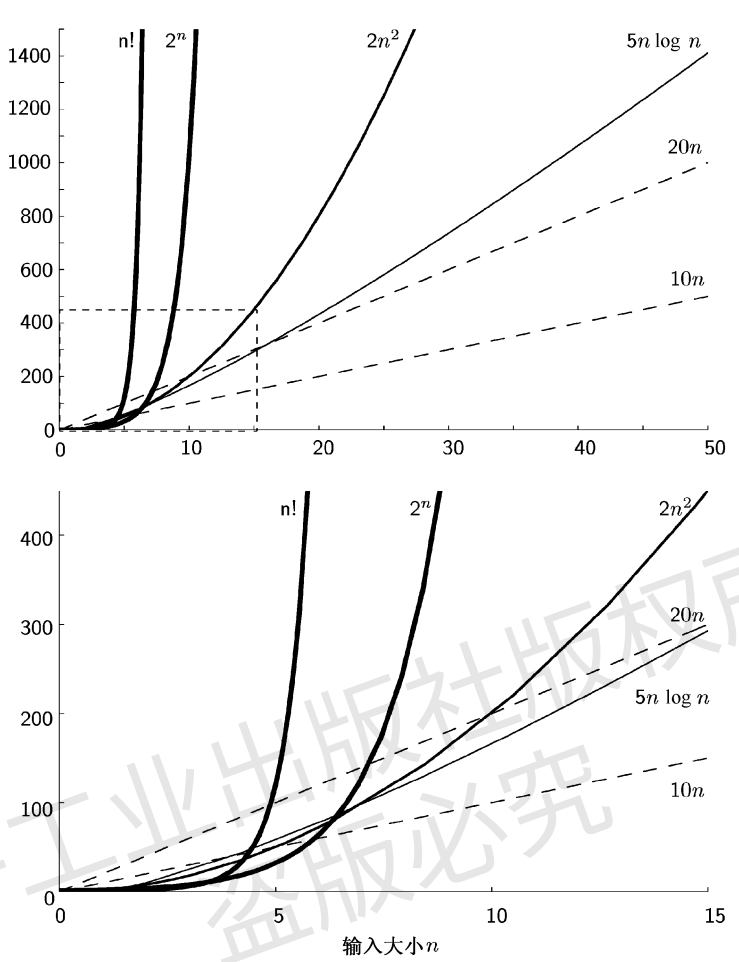

图 3.1 这是一幅图的两个视图，分别对应 6 个函数的增长率。下图是上图左下角的放大。水平轴代表输入规模，垂直轴表示时间、空间或其他开销

n	$\log\log n$	$\log n$	n	$n\log n$	n^2	n^3	2^n
16	2	4	2^4	$2\cdot 2^4 = 2^5$	2^8	2^{12}	2^{16}
256	3	8	2^8	$8\cdot 2^8 = 2^{11}$	2^{16}	2^{24}	2^{256}
1024	≈ 3.3	10	2^{10}	$10\cdot 2^{10} \approx 2^{13}$	2^{20}	2^{30}	2^{1024}
64K	4	16	2^{16}	$16\cdot 2^{16} = 2^{20}$	2^{32}	2^{48}	$2^{64\mathrm{K}}$
1M	≈ 4.3	20	2^{20}	$20\cdot 2^{20} \approx 2^{24}$	2^{40}	2^{60}	$2^{1\mathrm{M}}$
1G	≈ 4.9	30	2^{30}	$30\cdot 2^{30} \approx 2^{35}$	2^{60}	2^{90}	$2^{1\mathrm{G}}$

图 3.2 大多数计算机算法典型增长率的代价

3.2 最佳、最差和平均情况

计算 n 的阶乘的问题，只有一个输入，即给定的“大小”(也就是说，对于每个不同的 n，都只对应于一个问题)。现在考虑例 3.1 顺序查找最大元素的算法，它总是检查数组中的每一个元素。这个算法对于不同大小的 n 都适用。也就是说，对于一个给定的大小 n，可以有

很多种 n 个元素的数组。然而，不管算法查看的是哪个元素，算法的代价和查找数组中所有元素的代价都相同。

对于某些算法，即使问题规模相同，如果输入数据不同，其时间开销也不同。例如，现在要从一个 n 元一维数组中找出一个给定的 K（假设该数组中有且仅有一个元素值为 K）。顺序搜索法（sequential search）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。一旦找到了 K，算法也就完成了。这与例 3.1 的最大元素顺序搜索不同，后者必须检查每一个元素的值。

这样，顺序搜索法的时间开销可能在一个很大的范围内浮动。数组中的第一个元素可能恰恰就是 K，于是只要检查一个元素就行了。在这种情况下，运行时间很短，称为算法的最佳情况（best case），因为顺序搜索法不可能执行比检查一个元素更少的操作了。另一种情况，如果数组的最后一个元素是 K，运行时间就会相当长，因为这个算法要检查所有 n 个元素。这是算法的最差情况（worst case），因为该算法不可能检查 n 个以上的元素。如果用一个程序来实现顺序搜索法，并用该程序对许多不同的 n 元数组进行搜索，或者在同一个数组中搜索不同的 K 值，就会发现平均搜索到整个数组的一半就能找到 K。也就是说，这种算法平均要检查 $n/2$ 个元素，称之为算法的平均情况（average case）时间代价。

分析一种算法时，应该研究最佳、最差还是平均情况呢？一般来说，对最佳情况没有多大兴趣，因为它发生的概率太小，而且对于条件的考虑太过乐观了。换言之，最佳情况不能作为算法性能的代表。不过，也有一小部分情况，最佳情况分析是有用的——尤其是当最佳情况出现概率较大的时候。在第 7 章，还会看到一些排序算法的例子，可以用最佳情况运行时间来分析该算法，非常迅速而有效。

那么最差情况呢？分析最差情况有一个好处：它能让你知道算法至少能做得多快。这一点在实时系统中尤其重要，例如空运处理系统。在这个系统中，一个“绝大部分”情况下能管理 n 架飞机的算法，如果它不能在规定的时间内管理来自于同一方向的 n 架飞机，那么它是不能被接受的。

在另外一些情况下——特别是想要知道程序要对许多不同的输入运行多次时的总计时间开销——最差情况分析就不适合于用来衡量一种算法的性能了。通常会更希望知道平均情况的时间代价，也就是说，当输入规模为 n 时算法的“典型”表现。可惜，平均情况分析并不总是可行的。首先，它要求人们清楚程序的实际输入在所有可能的输入集合中是如何分布的。例如，上面提到过顺序搜索法在平均情况下要检查数组中一半的元素。这只在整数 K 所对应的元素在数组中每个位置出现的概率相等时才成立。如果这个假设不成立，那么算法的平均情况就不一定是检查一半的元素了。9.2 节将对这个问题做进一步的讨论。

数据分布的特点对于很多搜索算法都会有很大影响，例如 9.4 节提到的散列检索和 5.4 节给出的检索树。不正确的假设有时会给程序的时间或空间性能带来灾难性的影响。一些特殊的数据分布也会带来好处，这一点在 9.2 节的例子中得到了很好的体现。

总之，在实时系统中，比较关注最差情况算法分析。在其他情况下，通常考虑平均情况，只要知道计算平均情况所需要的输入数据的分布即可。否则，就只能求助于最差情况分析了。

3.3 换一台更快的计算机，还是换一种更快的算法

假设你要解决一个问题，而且已经知道一种算法，其时间代价为 cn^2（c 为常数）。但是，程序运行时间比规定的时间慢了 10 倍。如果现在换一台运行速度比原有运行速度快 10 倍的计算机，这种算法是否就可行了呢？如果问题规模不变，也许这台新计算机的确可以按时完成任务，尽管你的算法的增长率很高。但是，对于大多数拥有更先进计算机的人来说，事情并不那么简单——他们并不想解决相同的问题，而是要解决更大规模的问题！假定你希望原来的计算机对 10 000 个数进行排序，因为这是一个午餐休息时间所能完成的任务，那么现在对于这台新的计算机，你可能会要求它在相同的时间里对 100 000 个数进行排序。午餐时间不会缩短，因此你希望解决一个更大规模的问题。既然新机器的运行速度快了 10 倍，你很自然希望它解决问题的规模也增大 10 倍。

如果你的算法增长率是线性的（即运行时间函数 $\mathbf{T}(n)=cn$，c 为常数），那么新计算机处理 100 000 个数据的时间与原来的计算机处理 10 000 个数据的时间相同。如果算法的增长率高于 cn（如 c_1n^2），那么在相同的时间里就不能在一台速度提高 10 倍的计算机上完成一个规模扩大 10 倍的问题的计算。

那么，在给定的时间内，速度变快的机器能够处理问题的规模扩大了几倍呢？假设新的计算机的运行速度是原来的 10 倍，原有机器 1 小时能完成的问题规模为 n，那么新机器 1 小时最多可以解决的问题的规模有多大？图 3.3 给出了图 3.1 中列出的 5 个运行时间函数可以解决问题的规模。

f(n)	n	n'	关系	n'/n
$10n$	1 000	10, 000	$n'=10n$	10
$20n$	500	5 000	$n'=10n$	10
$5n\log n$	250	1 842	$\sqrt{10}n<n'<10n$	7.37
$2n^2$	70	223	$n'=\sqrt{10}n$	3.16
2^n	13	16	$n'=n+3$	−−

图 3.3 在规定时间内，速度是原来 10 倍的计算机所能处理问题规模的增长情况。第1栏列出了图3.1中给出的5个运行时间函数。不妨假设原来的计算机能在1小时内完成10 000个基本操作。第2栏显示了10 000个基本操作所能完成的n的最大值。第3栏是n'的值，即新机器解决该问题的最大值。第4栏给出了n与n'之间的函数关系。第5栏给出了n'与n的比值

从图中可以看出许多重要内容。前两个函数是线性的，只是常数系数不同。新机器处理二者时，问题规模的增长都是 10 倍。也就是说，系数大小虽然影响一定时间内能够解决问题的规模，却不能影响机器速度加快时问题规模的增长（与原问题规模之间的比值）。无论算法增长率是多少，下面这个结论总是成立的：常数系数不会改变机器速度加快时问题规模的增长倍数。

运行时间 $\mathbf{T}(n)=2n^2$ 的算法问题规模增长倍数要比线性增长的算法小。后者增长了 10 倍，前者只是它的平方根：$\sqrt{10}\approx 3.16$。可见，增长率高的算法从机器升级中获益较少，能解决的问题规模也较小。随着计算机的速度加快，不同算法解决的问题规模的差别就变得更大。

$\mathbf{T}(n)=5n\log n$ 的算法比二次增长率的算法提高较多，但是也不如线性增长。注意指数增长率的算法有一些特别。图3.1中显示，增长率为 2^n 的算法对应的曲线增长得很快。在图3.3中，新的计算机可以处理的问题的规模约为 $n+3$（确切地说是 $n+\log_2 10$）。其问题规模的增长只是加上了一个常数，而不是乘上一个因数。原来的 n 值为13，新的问题规模就是16。假如明年再换一台计算机，比现在还要快10倍，它也不过只能处理规模为19的问题。假如还有另一个增长率为 2^n 的算法，某台计算机能在1小时内处理规模为1 000的该问题，那么换一台速度加快10倍的机器，1小时也只能解决规模是1 003的问题！可见，指数增长率的算法与图3.3中的其他算法有根本的不同。关于这种差异的意义，将在第17章做进一步的探讨。

看来，你不应该急着买一台新机器，而应该考虑使用另一种运行时间为 $n\log n$ 的算法来代替现有的运行时间为 n^2 的算法。在图3.1中，水平轴线代表时间。如果在规定时间内，两条对应不同增长率的曲线已经相交，那么增长速度较慢的那种算法就会较快。当问题规模 $n=1\ 024$ 时，运行时间 $\mathbf{T}(n)=n^2$ 的算法需要 $1\ 024\times1\ 024=1\ 048\ 576$ 个单位时间，$\mathbf{T}(n)=n\log n$ 的算法需要 $1\ 024\times10=10\ 240$ 个单位时间，两者之比远远大于10倍。因为 $n>58$ 时，有 $n^2>10n\log n$，所以如果问题规模大于58，最好换一种算法，而不是换一台计算机。而且，即使购买了一台速度更快的计算机，就同一时间所能解决问题规模的增长情况而言，增长率较小的算法才能从中获益较多。

3.4　渐近分析

虽然图3.1中标有 $10n$ 的曲线系数较大，$2n^2$ 曲线还是在 $n=5$ 时就超过它了。如果把这个线性方程的系数再增加一倍，结果又将如何呢？如图所示，当 $n=10$ 时，$20n$ 对应的曲线也被 $2n^2$ 超过。这两个线性增长率的系数并没有产生很大影响，只不过改变了交点的横坐标值。推而广之，改变其中一个函数的常系数，只改变两个曲线“在何处”相交，而不能改变它们“是否”相交。

当你拥有一台更快的计算机或者一个更快的编译系统时，一定时间内一定增长率下可以完成问题的规模增长倍数是一个定值，与运行时间函数中的系数无关。类似地，两个增长率不同的算法对应的时间曲线总是会相交的，与运行时间函数的系数无关。因此，估算一种算法运行的时间或者其他开销的增长率时，经常忽略其系数。这样能够简化算法分析，并且把注意力集中在最重要的一点上：增长率。这就是渐近算法分析。准确地说，渐近分析是指当输入规模很大，或者说达到极限（在微积分意义上）时，对一种算法进行的研究。实践证明忽略这些系数很有用，因此渐近分析也被广泛应用于算法比较。

并不是任何时候都能忽略常数。当算法要解决的问题规模 n 很小时，系数就会起到举足轻重的作用。例如，现在要对5个数排序，那么用来给成千上万个数排序的算法可能就并不是很适合，尽管它的渐近分析表明该算法性能良好。也有少量的例子，两种被比较的算法，其中具有较小增长率的算法，由于其较大的系数而对大多数情况不适宜。渐近分析是对算法资源开销的一种不精确的估算，是一种“信封背面的计算”。它提供了对算法资源开销进行评估的简化模型。但是千万不要忘记渐近分析的局限性，尤其是在那些系数也起到重要作用的少数情况下。

3.4.1 上限

有几个术语是专门用于描述算法时间函数的。这些术语及其符号能够准确反映函数某一方面的特性。算法运行时间的上限(upper bound)就是其中之一,用来表示该算法可能有的最高增长率。

“增长率的上限为$f(n)$”这句话太长了,但是它在分析算法时又极其常用,于是采用一种特殊的表示法,称为大 O 表示法(big Oh notation),读做“大欧”表示法。如果某个算法的增长率上限(在最差情况下)是$f(n)$,那么就说这种算法“最差情况下在集合 $O(f(n))$中”,或者直接说“最差情况下在 $O(f(n))$中”。例如,如果在最差情况下 $\mathbf{T}(n)$增长速度与 n^2相同,则称算法最差情况下在 $O(n^2)$中。

下面给出了上限的一个精确定义。其中 $\mathbf{T}(n)$表示算法的实际运行时间,$f(n)$则是上限的一个函数表达式。

对于非负函数 $\mathbf{T}(n)$,如果存在两个正常数 c 和 n_0,对任意 $n > n_0$,有 $\mathbf{T}(n) \leqslant cf(n)$,则称 $\mathbf{T}(n)$在集合 $O(f(n))$中。

常数 n_0是使上限成立的 n 的最小值。一般情况下 n_0都很小,如取 1,但是并不一定要如此。必须能够找出这样的一个常数 c,而 c 确切是多少无关紧要。换言之,定义指出对于问题的所有(比如最差情况)输入,只要输入规模足够大(即 $n > n_0$),该算法总是能在 $cf(n)$步以内完成,c 是某个确定的常数。

例 3.4 考虑找出数组中某个元素的顺序搜索法。如果访问并检查数组中的一个元素需要时间 c_s(c_s为正数),如果待查元素在数组每个位置的出现概率均等,那么在平均情况下 $\mathbf{T}(n) = c_s n/2$。对于 $n > 1$,$c_s n/2 \leqslant c_s n$。所以根据定义,$\mathbf{T}(n)$在 $O(n)$中,$n_0 = 1$,$c = c_s$。

例 3.5 某一算法平均情况下 $\mathbf{T}(n) = c_1 n^2 + c_2 n$,$c_1$、$c_2$为正数。若 $n > 1$,$c_1 n^2 + c_2 n \leqslant c_1 n^2 + c_2 n^2 \leqslant (c_1 + c_2) n^2$。因此取 $c = c_1 + c_2$,$n_0 = 1$,有 $\mathbf{T}(n) \leqslant cn^2$。根据第二次定义,$\mathbf{T}(n)$在 $O(n^2)$中。

例 3.6 把数组中的第一个元素值赋给一个变量,这个算法的运行时间是一定的,与数组大小无关。因此,在最佳、最差和平均情况下恒有 $\mathbf{T}(n) = c$。可以认为在这种情况下 $\mathbf{T}(n)$在 $O(c)$中。不过,按照传统说法,运行时间上限为常数的算法在 $O(1)$中。

如果有人突然问你“什么是最好的?”,你最自然的反应会是“最好的什么?”。同样,当你被问及“这个问题的增长率是什么?”,你应该会反问“在什么情况下?最好情况?平均情况?或者是最差情况?”一些算法在不同情况下有相同的结果。一个例子就是在数字中寻找最大值。但是对于许多问题,不同情况下会有很大的区别,比如在一个未排序的数组中寻找某一个数。所以如果要做一个关于算法上限的论断,它应与输入规模 n 有关。几乎总是考虑最佳、最差、平均情况下的上限,因此不能说“这种算法增长率的上限为 n^2”,而应该说“这种算法平均情况下增长率的上限为 n^2”。

要知道,某个算法在 $O(f(n))$中只是说事情顶多能坏到某种地步,事实上也许并不是那么糟。如果知道顺序搜索法在 $O(n)$中,那么也可以说它在 $O(n^2)$中。但是顺序搜索法对于很大的 n 也是可行的,其他在 $O(n^2)$中的算法就不一定如此了。人们总是试图给算法的时间

代价找到一个最“紧”(即最小)的上限，因此，一般说顺序搜索法在 $O(n)$ 中。这也说明了为什么用“在 $O(f(n))$ 中”或用记号“$\in O(f(n))$”的表示方法，而不用“是 $O(f(n))$”或“$=O(f(n))$”。使用大 O 表示法没有严格的等号。$O(n)$ 在 $O(n^2)$ 中，但 $O(n^2)$ 不在 $O(n)$ 中。

3.4.2　下限

大 O 表示法描述上限。也就是说，当某一类数据的输入规模为 n 时（通常为最差情况，所有可能输入情况的平均，或最佳情况输入），一种算法消耗某种资源(通常是时间)的最大值。

相似的表示方法可以用来描述算法在某类数据输入时所需要的最少资源。与大 O 表示法类似，它也是算法增长率的一个衡量尺度。它同样可以表示任何资源，但是一般衡量最小时间代价。还有一点类似之处，对于输入规模 n，针对一些特殊的输入来估计资源开销：最差、最佳和平均情况下的资源开销。

算法(或以后将讲到的问题)的下限用符号 Ω 来表示，读做“大欧米伽”(Omega)或“欧米伽”。下面给出 Ω 的定义，它与大 O 表示法的定义极其相似。

*若存在两个正常数 c 和 n_0，对于 $n>n_0$，有 $\mathbf{T}(n)\geqslant cg(n)$，则称 $\mathbf{T}(n)$ 在集合 $\Omega(g(n))$ 中。*①

例 3.7　假定 $\mathbf{T}(n)=c_1n^2+c_2n(c_1,c_2>0)$，则有

$$c_1n^2+c_2n\geqslant c_1n^2$$

因此，取 $c=c_1$，$n_0=1$，有 $\mathbf{T}(n)\geqslant cn^2$，根据定义，$\mathbf{T}(n)$ 在 $\Omega(n^2)$ 中。

也可以说例 3.7 中 $\mathbf{T}(n)$ 也在 $\Omega(n)$ 中。但是，正如大 O 表示法一样，同样希望找到一个最“紧”的可能限制(对于 Ω 表示法是最大的)。因此，一般说这个 $\mathbf{T}(n)$ 在 $\Omega(n^2)$ 中。

回忆一下在数组中寻找值为 K 的元素的顺序搜索法。在平均情况和最差情况下，这个算法在 $\Omega(n)$ 中，因为在这两种情况下，都至少要检查 cn 个元素(在平均情况下 $c=1/2$，在最差情况下 $c=1$)。

---

① 也可以用另一种方法定义 Ω：

如果存在正常数 c，有无穷多个 n 使 $\mathbf{T}(n)\geqslant cg(n)$ 成立，则称 $\mathbf{T}(n)$ 在集合 $\Omega(g(n))$ 中。

这个定义要求在无限多的情况下，该算法的时间代价超过 $cg(n)$。注意该定义与大 O 表示法的定义不太相似。当 $g(n)$ 是下限时，定义并不要求对所有大于某一常数的 n，都有 $\mathbf{T}(n)\geqslant cg(n)$ 成立。它只要求这种状况发生得足够频繁，特别是有无穷多个 n 满足这个条件。这种定义的优点可以在下面这个例子中发现。

假定某个算法表达式如下：

$$\mathbf{T}(n)=\begin{cases} n, & \text{所有奇数 } n\geqslant 1 \\ n^2/100, & \text{所有偶数 } n\geqslant 0 \end{cases}$$

$n\geqslant 0$ 且 n 为偶数时，$n^2/100\geqslant(1/100)n^2$。取 $c=1/100$，则有无数多个 n(偶数的 n)使 $\mathbf{T}(n)\geqslant cn^2$ 成立。根据定义，$\mathbf{T}(n)$ 在 $\Omega(n^2)$ 中。

对于 $\mathbf{T}(n)$ 的运行时间函数而言，输入规模为 n 的时间代价至少为 cn。但是，有无穷多个 n，当输入规模为 n 时，运行时间为 cn^2，因此更倾向于说该算法在 $\Omega(n^2)$ 中。但是，用第一个定义会得到下限为 $\Omega(n)$，因为不能找到常数 c 和 n_0，使任何 $n>n_0$ 都有 $\mathbf{T}(n)\geqslant cn^2$。而用现在这个定义的确能够得到该算法的下限为 $\Omega(n^2)$，这更符合通常的标准。幸而在现实生活中，很少有程序或算法会有如此奇怪的特性，Ω 的第一个定义一般还是能得到正确结果的。

从以上讨论中可以看到，渐近分析的下限表示法不是一种自然存在的规律，只不过是用以描述算法性能的有用工具而已。

3.4.3 Θ 表示法

大 O 表示法和 Ω 表示法能够描述某一算法的上限(如果能找到某一类输入下开销最大的函数)和下限(如果能找到某一类输入下开销最小的函数)。当上、下限相等时,可能用 Θ 表示法,读做“西塔”(Theta)或“大西塔”。如果一种算法既在 $O(h(n))$ 中,又在 $\Omega(h(n))$ 中,则称其为 $\Theta(h(n))$。注意在 Θ 表示法中,不再说“在……中”,因为两个 Θ 相同的函数有交换性,也就是说,若 $f(n)$ 是 $\Theta(g(n))$,则 $g(n)$ 是 $\Theta(f(n))$。

因为在平均情况下,顺序搜索法既在 $O(n)$ 中,又在 $\Omega(n)$ 中,所以说平均情况下它是 $\Theta(n)$。

给出一个反映某算法时间代价的算术表达式,其上、下限通常都是相等的。这是因为从某种意义上说,已经对该算法有了一个精确的分析,用运行时间函数表示出来。对于很多算法(或者其实现形式,如程序),能够很容易地写出反映它们时间代价的函数。本书所列举的绝大部分算法都很浅显易懂,并且可以做出 Θ 分析。但是,在第 17 章,将看到整整一类算法无法用 Θ 表示法分析,有的甚至不能用大 O 和 Ω 表示法分析。习题 3.14 给出了一个简短的程序,但是目前还没有人能够求出它的确切上下限。

某些教材和程序员有时说一个算法是大 O 某个代价函数的。如果具有关于该算法的足够知识,确切知道其上限和下限正好相等,则 Θ 表示法一般比大 O 表示法要好。在本书中,对算法了解较深时,一般采用 Θ 表示法,但是限于分析能力,在有的算法中还会用到大 O 或者 Ω 表示法。在极少数情况下,当明确知道讨论的是问题或算法的上限还是下限时,使用相应的表示法,而不是使用 Θ 表示法。

3.4.4 化简法则

一旦知道了算法的运行时间函数,从中推导出大 O、Ω 和 Θ 表达式并不是一件很困难的事情。并不需要严格遵循定义来推导,可以用下面的法则来求得其最简形式:

1. 若 $f(n)$ 在 $O(g(n))$ 中,且 $g(n)$ 在 $O(h(n))$ 中,则 $f(n)$ 在 $O(h(n))$ 中;
2. 若 $f(n)$ 在 $O(kg(n))$ 中,对于任意常数 $k>0$ 成立,则 $f(n)$ 在 $O(g(n))$ 中;
3. 若 $f_1(n)$ 在 $O(g_1(n))$ 中,且 $f_2(n)$ 在 $O(g_2(n))$ 中,则 $f_1(n)+f_2(n)$ 在 $O(\max(g_1(n), g_2(n)))$ 中;
4. 若 $f_1(n)$ 在 $O(g_1(n))$ 中,且 $f_2(n)$ 在 $O(g_2(n))$ 中,则 $f_1(n)f_2(n)$ 在 $O(g_1(n)g_2(n))$ 中。

第一条法则是说,如果 $g(n)$ 是算法代价函数的一个上限,则 $g(n)$ 的任意上限也是该算法代价的上限。对 Ω 表示法有类似的性质:若 $g(n)$ 是算法代价函数的一个下限,$g(n)$ 的任意下限也是该算法代价的下限,Θ 表示法同理。

法则 2 的意义在于使人们能够忽略大 O 表示法中的常数因子。对于 Ω 和 Θ 表示法同样有这样的性质。

法则 3 说明,顺序给出一个程序的两个部分(两组语句或两段代码),只需要考虑其中开销较大的部分。类似地,在 Ω 与 Θ 表示法中,也只看开销大的部分就可以了。

法则 4 用于分析程序中的简单循环。如果要有限次地重复某种操作,且每次重复的开销相等,则总开销为每次的开销与重复次数之积。对于 Ω 和 Θ 表示法结论也成立。

综合考虑前三条性质,可以在计算任何算法开销的渐近增长率时,忽略所有常数和低次项。忽略常数的优点和不足在本节前面的内容中已有论述。进行算法分析时忽略低次项也合

乎情理。因为当 n 增大时，相对于高次项来说，低次项在总开销中所占比例微乎其微。因此，如果 $\mathbf{T}(n)=3n^4+5n^2$，可以说 $\mathbf{T}(n)$ 在 $O(n^4)$ 中，因为 n^2 项，在 n 比较大时，对于总体开销来说无足轻重。

在本书的其他内容中讨论算法或程序的开销时，会不断使用这些化简法则。

3.4.5　函数分类

给定函数 $f(n)$ 和 $g(n)$，其增长率都采用算术公式表达，需要判断哪个增长率更快。最好的方式就是判断这两个函数比值的极限，

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}$$

如果极限值趋向于 ∞，则 $f(n)$ 在 $\Omega(g(n))$ 中，因为 $f(n)$ 增长得更快。如果极限值趋向于0，则 $f(n)$ 在 $O(g(n))$ 中，因为 $g(n)$ 增长得更快。如果极限值趋向于某个非0常数，则 $f(n)=\Theta(g(n))$，因为二者增长率相同。

例3.8　如果 $f(n)=2n\log n$ 和 $g(n)=n^2$，那么 $f(n)$ 在 $O(g(n))$、$\Omega(g(n))$ 还是 $\Theta(g(n))$ 中？因为

$$\frac{n^2}{2n\log n}=\frac{n}{2\log n}$$

可以很容易看出

$$\lim_{n\to\infty}\frac{n^2}{2n\log n}=\infty$$

由于 n 增长得比 $2\log n$ 更快，因此 n^2 在 $\Omega(2n\log n)$ 中。

3.5　程序运行时间的计算

这一节给出了几个简单程序段的分析。

例3.9　首先来看一个给整型变量赋值的简单语句：

```
a = b;
```

该语句执行时间为常数级的，即 $\Theta(1)$。

例3.10　再来看一个简单的 **for** 循环：

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

第一条语句的时间代价为 $\Theta(1)$。**for** 循环重复了 n 次，第三个语句时间代价为一常量，根据3.4.4节中的化简法则4，后两行的 **for** 循环的总时间代价为 $\Theta(n)$。根据法则3，整个程序段的代价也是 $\Theta(n)$。

例3.11　下面是一个含有多个 **for** 循环的程序段，其中有些是嵌套的。

```
sum = 0;
for (i=1; i<=n; i++)      // First for loop
   for (j=1; j<=i; j++)   //   is a double loop
      sum++;
for (k=0; k<n; k++)       // Second for loop
   A[k] = k;
```

该程序段有三个相对独立的片断：一个赋值语句和两个 for 循环结构。同样，赋值语句时间代价为常量，记为 c_1；第二个 for 循环与例 3.10 相似，时间代价为 $c_2n=\Theta(n)$。

第一个 for 循环是一个双重循环，需要一点特殊的技巧。从内层循环入手：运行 sum++ 需要时间为一常量，记为 c_3，内层循环执行 i 次，根据法则 4，时间开销为 c_3i。外层循环共执行 n 次，但是每一次内层循环的时间开销都因 i 的变化而不同。可以看到，第一次执行外层循环时 $i=1$，第二次执行时 $i=2$。每执行一次外层循环，i 就以 1 为步长递增，直至最后一次 $i=n$。因此，总的时间开销是从 1 累加到 n 再乘以 c_3，根据式(2.1)，可以得出：

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

即 $\Theta(n^2)$，根据法则 3，总的运行时间为 $\Theta(c_1+c_2n+c_3n^2)$，可简化为 $\Theta(n^2)$。

例 3.12 比较下面两段程序的算法分析：

```
sum1 = 0;
for (i=1; i<=n; i++)      // First double loop
   for (j=1; j<=n; j++)   //   do n times
      sum1++;

sum2 = 0;
for (i=1; i<=n; i++)      // Second double loop
   for (j=1; j<=i; j++)   //   do i times
      sum2++;
```

在第一个双重循环中，内层 for 循环总是执行 n 次。因为外层循环执行 n 次，所以 sum1++ 语句显然恰好执行 n^2 次。而第二个循环与上题的例子相仿，时间代价为 $\sum_{j=1}^{n} j$，近似于 $\frac{1}{2}n^2$。因此，两个二重循环的时间代价都为 $\Theta(n^2)$，不过第二个程序段的运行时间约为第一个的一半。

例 3.13 并非所有的嵌套 for 循环的时间代价都是 $\Theta(n^2)$。以下面的两个嵌套循环为例：

```
sum1 = 0;
for (k=1; k<=n; k*=2)     // Do log n times
   for (j=1; j<=n; j++)   // Do n times
      sum1++;

sum2 = 0;
for (k=1; k<=n; k*=2)     // Do log n times
   for (j=1; j<=k; j++)   // Do k times
      sum2++;
```

当分析上面两段代码的时候，假定 n 是 2 的幂。第一段程序的外层 for 循环执行 $\log n+1$ 次，因为每循环一次 k 就乘以 2，直至 $k>n$。由于内层循环执行次数恒为 n，所以第一段程序的总时间代价可以表示为 $\sum_{i=0}^{\log n} n$。这里有一个变量替换：$k=2^i$。根据式(2.3)，其和为 $\Theta(n\log n)$。在第二段程序中，外层循环也执行 $\log n+1$ 次，而内层循环重复 k 次，随着外层循环的增长而成倍递增。其总和可以表示为 $\sum_{i=0}^{\log n} 2^i$，其中 n 假定恰为 2 的幂，且 $k=2^i$。因为外层循环执行 $\log n$ 次，而内层循环执行 i 次，i 每次都成倍递增，由式(2.8)可知，其和为 $\Theta(n)$。

那么其他控制语句又如何呢？while 循环的分析方法与 for 循环类似。if 语句的最差

情况时间代价是 **then** 和 **else** 语句中时间代价较大的那一个。对于平均情况代价也是如此，假如 n 的取值与执行哪一条指令的概率无关(通常情况下都是如此，但不一定如此)。**switch** 语句的最差情况代价是所有分支中开销最大的那一个。子程序调用时，只要加上执行子程序的时间即可。

有少数情况，执行 **if** 或 **switch** 语句中的某一个分支的概率是输入规模的函数。例如，输入规模为 n 时，**if** 语句中的 **then** 语句执行的概率为 $1/n$。举个简单的例子，某个 **if** 语句规定，当对 n 个数中最小的那一个进行操作时，才执行 **then** 语句。对这类问题进行分析时，不能简单地将其处理成开销较大分支的时间代价。这时，均摊分析方法(amortized analysis，见14.3节)的技巧就很有用了。

计算一个递归子程序的执行时间是一件令人头疼的事情。递归过程的运行时间一般都能通过一个递归关系式很好地体现。例如，2.5节提到的递归函数 **fact** 每递归调用自身一次，问题规模就减少1。调用返回的结果与输入参数相乘，这个操作的运行时间是一个常量。因此，如果希望用乘数来度量代价，那么函数的时间代价就大于该乘数乘上在少量输入上执行递归调用的次数。因为在初始情况下不做乘法运算，所以代价是0。因此，这个函数的运行时间可以表示成

$$\mathbf{T}(n) = \mathbf{T}(n-1) + 1 \qquad n > 1;\ T(1) = 0$$

由例2.8和例2.13可知，这个递归关系的闭合形式解为 $\Theta(n)$。

本节的最后一个实例是比较两个完成搜索功能的算法。在以前的学习中，已经知道当被搜索元素 K 在数组中任意一个位置出现的概率相等时，顺序搜索法的平均和最差情况代价都是 $\Theta(n)$。下面将其与二分法检索(binary search)的运行代价进行比较，假设数组元素按照从小到大的顺序存储。

二分法检索从检查数组中间位置的元素开始；把这个位置记为 *mid*，相应的元素值记为 k_{mid}。如果 $k_{mid} = K$，那么搜索工作就完成了。当然，这是不太可能的事情。不过，这个中间元素值还是能够给出一些有用的信息，有助于继续搜索。当 $k_{mid} > K$ 时，知道 K 不可能在 *mid* 后面的位置上出现，因此，在以后的搜索中不必再考虑后半部分的元素。相反，如果 $k_{mid} < K$，就可以忽略 *mid* 前面的部分。无论哪种情况，都可以缩小一半范围。二分法的下一步工作是检查 K 可能存在的那部分元素的中间位置。这个位置上的元素值又能缩小一半搜索范围。重复这个过程，直到找到指定的元素，或者确定值 K 不在数组中。图3.4给出了二分法检索的图示。图3.5给出了二分法检索的实现。

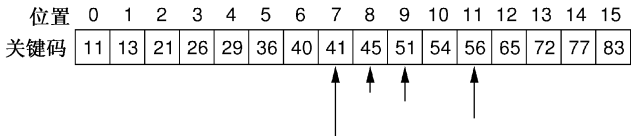

图3.4 16个元素的顺序数组二分法检索示意图。考察 $K=45$ 的搜索。首先检查7号位置的元素。因为 $41 < K$，K 不可能存放在小于7的位置。第二步，二分法检查11号位置的元素。$56 > K$，被搜索元素(若存在)必须在位置7～11之间。9号元素是下一个要检查的元素，还是太大了。最后一个要检查的是8号位置的元素，恰好是要搜索的元素。这样，**binary** 函数的最终返回值为8。如果 $K = 44$，搜索过程几乎完全相同，不过检查完8号元素之后，**binary** 将返回一个值 n，表示搜索失败

```
// Return the position of an element in sorted array "A" of
// size "n" with value "K".  If "K" is not in "A", return
// the value "n".
int binary(int A[], int n, int K) {
  int l = -1;
  int r = n;          // l and r are beyond array bounds
  while (l+1 != r) {  // Stop when l and r meet
    int i = (l+r)/2;  // Check middle of remaining subarray
    if (K < A[i]) r = i;      // In left half
    if (K == A[i]) return i; // Found it
    if (K > A[i]) l = i;      // In right half
  }
  return n; // Search value not in A
}
```

图 3.5 二分法检索的实现

要算出这个算法在最差情况下的代价，可以把它的运行时间模型化为递归，然后找出递归的闭合形式解。对 **binary** 的每一次递归调用都使数组减小近一半，因此可以将最差情况模型化为下面的式子，并简单地假定 n 是 2 的幂。

$$\mathbf{T}(n) = \mathbf{T}(n/2) + 1 \qquad n > 1; \quad \mathbf{T}(1) = 1$$

如果展开这个递归，就可以发现在到达初始情况之前只做 $\log n$ 次，并且每次展开代价增加 1。因此，这个递归的闭合形式解为 $\mathbf{T}(n) = \log n$。

函数 **binary** 的功能是查找 K 的(唯一)位置，并返回该位置。若 K 不在数组中，则返回一个特定信息。还可以对此算法稍加改动，使之能够返回 K 在数组中第一次出现的位置(若数组元素允许有重复)，或者当 K 不在数组中时返回小于 K 的最大元素的位置。

对比顺序检索和二分法检索，就可以看到当 n 增大时，顺序检索的平均和最差情况代价 $\Theta(n)$ 将远远大于二分法检索的代价 $\Theta(\log n)$。孤立地看，二分法比顺序检索的效率高得多。这种看法没有把用于二分法检索的常数因子大于顺序检索的因素考虑在内，而在二分法检索中转到下一个搜索位置的计算代价大于顺序检索中只在当前位置增 1 的代价。

但是，注意顺序检索的时间代价几乎相差不多，无论数组中的元素是否按照顺序保存。相反，二分法检索要求元素必须按从低到高的顺序保存。根据二分法检索使用的环境，这个排序的要求可能会对时间代价产生损害，因为要保持数组的有序性，在插入新元素时会增加时间代价。这里有一个权衡问题：使用二分法检索比较容易，但是维持一个有序的数组比较费时，怎样权衡其利弊呢？只有在解决具体问题时才能知道是否利大于弊。

3.6 问题的分析

通常，用“算法”分析的技巧来分析算法或者其对应的程序。其实，还可以用这种技巧来分析问题的代价。应该指出，一个问题代价的上限不应该超过已知最优解法的代价上限。但是一个问题代价的下限是什么意思呢？

假设对于一个问题，给定一个算法之后，输入不同的 n 能画出一个计算代价的图。定义 $\mathcal{A}$ 为解决问题的所有算法的集合(理论上来说，这样的算法有无数个)。对于集合 $\mathcal{A}$ 中的每个算法，都可以得到一个关于代价的图。那么，最差情况的下限就是所有图中最高点的最小值。

表明一个问题在 $\Omega(f(n))$ 中比表明一个算法(或程序)在 $\Omega(f(n))$ 中困难得多。因为一个问题在 $\Omega(f(n))$ 中意味着所有可能的解法都在 $\Omega(f(n))$ 中，即使是人们尚未考虑到的解法。

迄今为止，所给算法分析的例子都有明确的结果，而且大 O 与 Ω 可以统一。为了更好地理解大 O、Ω 和 Θ 三种表示法是如何描述一个问题或算法的特性的，最好来看一个你可能所知不详的问题。

现在来分析一个排序的问题。所有排序算法的最差情况代价中最小可能值是多少？一个排序算法至少必须检查输入的待排序序列中的每个元素，以便确定该序列是否已经有序了。于是，仅仅为了做输入和输出工作就至少要花 cn 时间。在很多类似的问题中，根据观察可以发现有 n 个数需要输入，于是很容易得到 $\Omega(n)$ 为一个下限。

在以前的学习中，你可能看到过最差情况代价在 $O(n^2)$ 中的排序算法。在程序设计入门课程中看到过的排序算法范例，最差情况的时间代价可能就是 $O(n^2)$。因此，$O(n^2)$ 是排序问题的一个上限。但是，$\Omega(n)$ 与 $O(n^2)$ 之间的情况又如何呢？是否存在更好的排序方法：如果不能找到最差情况代价小于 $O(n^2)$ 的算法，也不能找到一种分析方法说明排序问题最差情况代价的下限大于 $\Omega(n)$，那就不能确定是否存在一个更好的算法。

第 7 章给出了一种排序方法，时间代价在 $O(n \log n)$ 中。这使得上下限之间的差距大大缩小了。现在知道了一个下限 $\Omega(n)$ 和一个上限 $O(n \log n)$。是否还能找到更快的排序算法呢？人们为此付出了很大努力，但是徒劳无功。幸好（抑或不幸？）第 7 章中还给出了一种证明：任何排序算法的最差情况代价都在 $\Omega(n \log n)$ 中[①]。这个证明是算法分析领域的一个重要结果，它说明对于规模为 n 的排序问题，没有一种算法的运行时间比 $cn \log n$ 短。因此，可以得出结论：排序问题的最差情况代价为 $\Theta(n \log n)$，因为上限和下限重合了。

3.7　容易混淆的概念

渐近分析是计算机专业学生需要面对的最大智力挑战之一。大多数人把增长率和渐近分析相混淆，因此在概念上和术语上产生了误解。本节有助于了解易混淆概念的标准概念，以便避免混淆概念的情况发生。

区分上限和下限的一个问题是：对于大多数算法，识别出它们真正的增长率是很容易的。如果知道一个完整的代价函数，其上限和下限往往是相同的。只有在不完全清楚待处理任务时，区别上限和下限才有意义。如果对这种差别还不清楚，请重读一遍 3.6 节。使用 Θ 表示法来表示算法的上限和下限的增长率没有明显的差别（这也是一些简单算法的通常情况）。

混淆上限和下限的概念及最差情况和最佳情况的概念，这是常见的错误。最佳、最差或者平均情况给出了明确的实例，可以应用在问题中，以得到代价的衡量值。上限和下限则描述了这种代价的增长率。所以，定义一个算法或问题的增长率时，需要确定衡量指标（最佳、最差、平均情况），以及对代价增长率的描述（O，Ω，Θ）。

一个算法的上限与给定输入规模为 n 时的最差情况是不一样的。上限不是用来确定真正的代价的（对于给定的 n 值，即可确定具体的运行时间），而是用来确定代价的增长率。对于单个点，例如给定的一个 n 值，是没有增长率的概念的。增长率用于体现伴随输入规模变化的代价变化。同样，对于给定的规模 n，下限与给定输入规模 n 时的最佳情况也是不同的。

另一个常见的错误概念是：当输入规模尽可能小时出现算法的最佳情况，或者当输入规

① 幸运的是，可以证明这一结论；遗憾的是，排序问题是 $\Theta(n \log n)$，而非 $\Theta(n)$。

模尽可能大时出现算法的最差情况。事实上是：对每种输入规模都存在最佳和最差的情况。即对给定的一个规模(例如 i)的所有输入来说，规模为 i 的一个(或多个)输入是最佳情况，而另外一个(或多个)输入则是最差情况。通常(但不总是!)，对于一个任意规模，能给出最佳输入情况，也能给出最差输入情况。在理想情况下，当输入规模增大时，能够确定在最佳、最差和平均情况下的增长率。

例 3.14 对于顺序检索来说，最佳情况的增长率是多少？对于一个大小为 n 的数组，最佳情况是所找的那个数正好在数组的第一个位置。这与数组的大小无关。因此，无论 n 为多少，最佳情况都是待查数出现在 n 个位置中的第一个位置的时候，代价为 1。所以，没有必要说最佳情况出现在 $n=1$ 的时候。

例 3.15 对于一个在 n 个数值中寻找最大值的问题，假设要画一个当 n 增长时运算代价的图示。x 轴是 n，y 轴为代价。当 n 增长时，这是一个向右增长的对角线图(在继续阅读之前，你可以在草稿纸上画一下)。

现在，考虑在 20 个元素的数组中寻找最大值这个问题，画出一个代价图。x 轴的第一个位置对应最大值在数组第一个位置的概率。x 轴的第二个位置对应最大值在数组的第二个位置的概率。以此类推。当然，代价永远是 20。因此，画出来的图会是一条水平线，值为 20。你可以试着在草稿纸上画一下。

下面考虑在数组中做线性查找的情况。考虑一个代价图展现了所有大小为 20 的问题。第一个问题就是要找的数值在数组中的第一个位置，这样代价就为 1。第二个问题就是待找的数字在数组中的第二个位置，这样代价就为 2。如果将这些问题按照代价从小到大排序，画出来的图是一个从左下(0)到右上(20)的对角线。读者可以在草稿纸上画一下。最后，考虑 n 变大时的顺序查找问题。图会是什么样的？遗憾的是，这个问题没有一个简单的答案，因为这是在找最大值。

图的形状取决于是否考虑最佳情况的代价(这将会是值为 1 的水平线)，最差情况的代价(这将会是值为 i 的水平线，当 x 轴值为 i 时)和平均情况的代价(这将会是值为 $i/2$ 的水平线，当 x 轴值为 i 时)。这就是为什么说在最佳、最差、平均情况下函数 $f(n)$ 在 $O(g(n))$ 中。如果不知道问题输入是什么，那么就无法确定应该使用何种衡量代价的方法。

3.8 多参数问题

在有些情况下，要准确分析一个算法并给出其时间代价函数需要多个参数。为了阐释这一概念，来看看根据一幅图中各像素值出现的频率对这些值进行排序的一个算法。图通常用一个二维数组来表示，每个像素都是图的一个单元。像素的对应值是图中这个点的颜色或密度。假设每个像素的取值都可以是 0 到 $C-1$ 的任意整数。现在问题是如何计算出每种取值所对应的像素个数，并根据每种取值在图中出现的次数对其进行排序。假定该图是一个包含 P 个像素的矩形。下面是对应的算法：

```
for (i=0; i<C; i++)    // Initialize count
   count[i] = 0;
for (i=0; i<P; i++)    // Look at all of the pixels
   count[value(i)]++; // Increment a pixel value count
sort(count, C);        // Sort pixel value counts
```