第1章

绪论

作为万物之灵的人，与动物的根本区别在于理性，而计算则是理性的一种重要而具体的表现形式。计算机是人类从事计算的工具，是抽象计算模型的具体物化。基于图灵模型的现代计算机，既是人类现代文明的标志与基础，更是人脑思维的拓展与延伸。

尽管计算机的性能日益提高，但这种能力在解决实际应用问题时能否真正得以发挥，决定性的关键因素仍在于人类自身。具体地，通过深入思考与分析获得对问题本质的透彻理解，按照长期积淀而成的框架与模式设计出合乎问题内在规律的算法，选用、改进或定制足以支撑算法高效实现的数据结构，并在真实的应用环境中充分测试、调校和改进，构成了应用计算机高效求解实际问题的典型流程与不二法门。任何一位有志于驾驭计算机的学生，都应该从这些方面入手，不断学习，反复练习，勤于总结。

本章将介绍与计算相关的基本概念，包括算法构成的基本要素、算法效率的衡量尺度、计算复杂度的分析方法与界定技巧、算法设计的基本框架与典型模式，这些也构成了全书所讨论的各类数据结构及相关算法的基础与出发点。

§1.1 计算机与算法

1946年问世的ENIAC开启了现代电子数字计算机的时代，计算机科学（computer science）也在随后应运而生。计算机科学的核心在于研究计算方法与过程的规律，而不仅仅是作为计算工具的计算机本身，因此E. Dijkstra及其追随者更倾向于将这门科学称作计算科学（computing science）。

实际上，人类使用不同工具从事计算的历史可以追溯到更为久远的时代，计算以及计算工具始终与我们如影相随地穿越漫长的时光岁月，不断推动人类及人类社会的进化发展。从最初颜色各异的贝壳、长短不一的刻痕、周载轮回的日影、粗细有别的绳结，以至后来的直尺、圆规和算盘，都曾经甚至依然是人类有力的计算工具。

1.1.1 古埃及人的绳索

古埃及人以其复杂而浩大的建筑工程而著称于世，在长期规划与实施此类工程的过程中，他们逐渐归纳并掌握了基本的几何度量和测绘方法。考古研究发现，公元前2000年的古埃及人已经知道如何解决如下实际工程问题：通过直线l上给定的点P，作该直线的垂线。

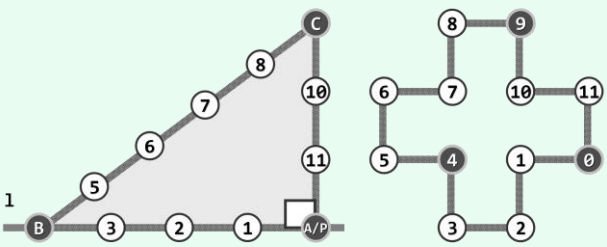

图1.1 古埃及人使用的绳索计算机及其算法

他们所采用的方法，原理及过程如图1.1所示，翻译成现代的算法语言可描述如下。

```
perpendicular(l, P)
输入：直线l及其上一点P
输出：经过P且垂直于l的直线
1．取12段等长绳索，依次首尾联结成环 //联结处称作“结”，按顺时针方向编号为0..11
2．奴隶A看管0号结，将其固定于点P处
3．奴隶B牵动4号结，将绳索沿直线l方向尽可能地拉直
4．奴隶C牵动9号结，将绳索尽可能地拉直
5．经过0号和9号结，绘制一条直线
```

算法1.1 过直线上给定点作直角

以上由古埃及人发明、由奴隶与绳索组成的这套计算工具，乍看起来与现代的电子计算机相去甚远。但就本质而言，二者之间的相似之处远多于差异，它们同样都是用于支持和实现计算过程的物理机制，亦即广义的计算机。因此就这一意义而言，将其称作“绳索计算机”毫不过分。

1.1.2 欧几里得的尺规

欧几里得几何是现代公理系统的鼻祖。从计算的角度来看，针对不同的几何问题，欧氏几何都分别给出了一套几何作图流程，也就是具体的算法。比如，经典的线段三等分过程可描述为如算法1.2所示。该算法的一个典型的执行实例如图1.2所示。

```
tripartition(AB)
输入：线段AB
输出：将AB三等分的两个点C和D
1．从A发出一条与AB不重合的射线ρ
2．任取ρ上三点C'、D'和B'，使|AC'| = |C'D'| = |D'B'|
3．联接B'B
4．过D'做B'B的平行线，交AB于D
5．过C'做B'B的平行线，交AB于C
```

算法1.2 三等分给定线段

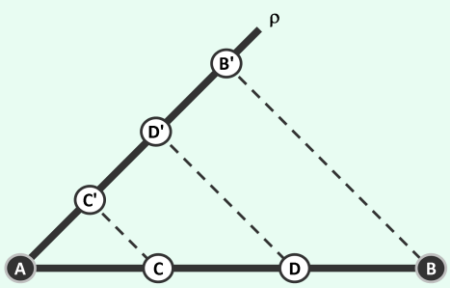

图1.2 古希腊人的尺规计算机

在以上算法中，输入为所给的直线段AB，输出为将其三等分的C和D点。我们知道，欧氏几何还给出了大量过程与功能更为复杂的几何作图算法，为将这些算法变成可行的实际操作序列，欧氏几何使用了两种相互配合的基本工具：不带刻度的直尺，以及半径跨度不受限制的圆规。同样地，从计算的角度来看，由直尺和圆规构成的这一物理机制也不妨可以称作“尺规计算机”。在尺规计算机中，可行的基本操作不外乎以下五类：

1 过两个点作一直线
2 确定两条直线的交点
3 以任一点为圆心，以任意半径作一个圆
4 确定任一直线和任一圆的交点（若二者的确相交）
5 确定两个圆的交点（若二者的确相交）

每一欧氏作图算法均可分解为一系列上述操作的组合，故称之为基本操作恰如其分。

1.1.3 起泡排序

D. Knuth[3]曾指出，四分之一以上的CPU时间都用于执行同一类型的计算：按照某种约定的次序，将给定的一组元素顺序排列，比如将n个整数按通常的大小次序排成一个非降序列。这类操作统称排序（sorting）。

就广义而言，我们今天借助计算机所完成的计算任务中，有更高的比例都可归入此类。例如，从浩如烟海的万维网中找出与特定关键词最相关的前100个页面，就是此类计算的一种典型形式。排序问题在算法设计与分析中扮演着重要的角色，以下不妨首先就此做一讨论。为简化起见，这里暂且只讨论对整数的排序。

■ 局部有序与整体有序

在由一组整数组成的序列A[0, n - 1]中，满足A[i - 1] ≤ A[i]的相邻元素称作顺序的；否则是逆序的。不难看出，有序序列中每一对相邻元素都是顺序的，亦即，对任意1 ≤ i < n都有A[i - 1] ≤ A[i]；反之，所有相邻元素均顺序的序列，也必然整体有序。

■ 扫描交换

由有序序列的上述特征，我们可以通过不断改善局部的有序性实现整体的有序：从前向后依次检查每一对相邻元素，一旦发现逆序即交换二者的位置。对于长度为n的序列，共需做n - 1次比较和不超过n - 1次交换，这一过程称作一趟扫描交换。

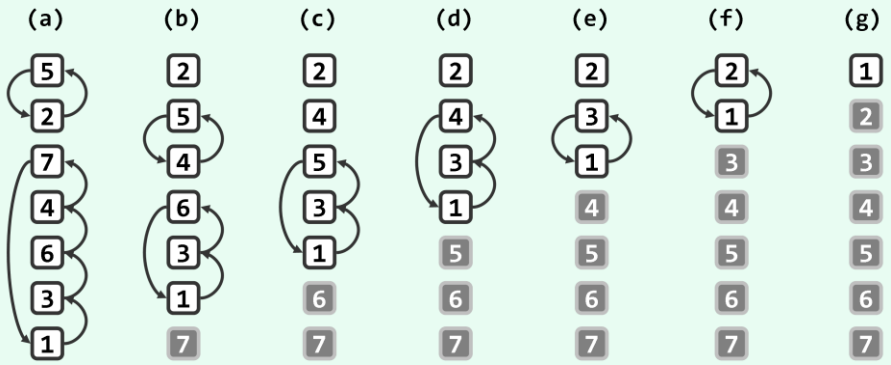

图1.3 通过6趟扫描交换对七个整数排序（其中已就位的元素以深色示意）

以图1.3(a)中由7个整数组成的序列A[0, 6] = { 5, 2, 7, 4, 6, 3, 1 }为例。

在第一趟扫描交换过程中，{ 5, 2 }交换位置，{ 7, 4, 6, 3, 1 }循环交换位置，扫描交换后的结果如图(b)所示。

■ 起泡排序

可见，经过这样的一趟扫描，序列未必达到整体有序。果真如此，则可对该序列再做一趟扫描交换，比如，图(b)再经一趟扫描交换的结果如图(c)。事实上，很有可能如图(c~f)所示，需要反复进行多次扫描交换，直到如图(g)所示，在序列中不再含有任何逆序的相邻元素。多数的这类交换操作，都会使得越小（大）的元素朝上（下）方移动（习题[1-3]），直至它们抵达各自应处的位置。

排序过程中，所有元素朝各自最终位置亦步亦趋的移动过程，犹如气泡在水中的上下沉浮，起泡排序（bubblesort）算法也因此得名。

■ 实现

上述起泡排序的思路，可准确描述和实现为如代码1.1所示的函数bubblesort1A()。

```
1  void bubblesort1A ( int A[], int n ) { //起泡排序算法（版本1A）：0 <= n
2     bool sorted = false; //整体排序标志，首先假定尚未排序
3     while ( !sorted ) { //在尚未确认已全局排序之前，逐趟进行扫描交换
4        sorted = true; //假定已经排序
5        for ( int i = 1; i < n; i++ ) { //自左向右逐对检查当前范围A[0, n)内的各相邻元素
6           if ( A[i - 1] > A[i] ) { //一旦A[i - 1]与A[i]逆序，则
7              swap ( A[i - 1], A[i] ); //交换之，并
8              sorted = false; //因整体排序不能保证，需要清除排序标志
9           }
10       }
11       n--; //至此末元素必然就位，故可以缩短待排序序列的有效长度
12    }
13 } //借助布尔型标志位sorted，可及时提前退出，而不致总是蛮力地做n - 1趟扫描交换
```

代码1.1 整数数组的起泡排序

1.1.4 算法

以上三例都可称作算法。那么，究竟什么是算法呢？所谓算法，是指基于特定的计算模型，旨在解决某一信息处理问题而设计的一个指令序列。比如，针对“过直线上一点作垂直线”这一问题，基于由绳索和奴隶构成的计算模型，由古埃及人设计的算法1.1；针对“三等分线段”这一问题，基于由直尺和圆规构成的计算模型，由欧几里得设计的算法1.2；以及针对“将若干元素按大小排序”这一问题，基于图灵机模型而设计的bubblesort1A()算法，等等。

一般地，本书所说的算法还应必须具备以下要素。

■ 输入与输出

待计算问题的任一实例，都需要以某种方式交给对应的算法，对所求解问题特定实例的这种描述统称为输入（input）。对于上述三个例子而言，输入分别是“某条直线及其上一点”、“某条线段”以及“由n个整数组成的某一序列”。其中，第三个实例的输入具体地由A[]与n共同描述和定义，前者为存放待排序整数的数组，后者为整数的总数。

经计算和处理之后得到的信息，即针对输入问题实例的答案，称作输出（output）。比如，对于上述三个例子而言，输出分别是“垂直线”、“三等分点”以及“有序序列”。在物理上，输出有可能存放于单独的存储空间中，也可能直接存放于原输入所占的存储空间中。比如，第三个实例即属于后一情形，经排序的整数将按非降次序存放在数组A[]中。

■ 基本操作、确定性与可行性

所谓确定性和可行性是指，算法应可描述为由若干语义明确的基本操作组成的指令序列，且每一基本操作在对应的计算模型中均可兑现。以上述算法1.1为例，整个求解过程可以明白无误地描述为一系列借助绳索可以兑现的基本操作，比如“取等长绳索”、“联结绳索”、“将绳结固定于指定点”以及“拉直绳索”等。再如算法1.2中，“从一点任意发出一条射线”、“在直线上任取三个等距点”、“联接指定两点”等，也都属于借助尺规可以兑现的基本操作。

细心的读者可能会注意到，算法1.2所涉及的操作并不都是基本的，比如，最后两句都要求“过直线外一点作其平行线”，这本身就是另一几何作图问题。幸运的是，借助基本操作的适当组合，这一子问题也可圆满解决，对应的算法则不妨称作是算法1.2的“子算法”。

从现代程序设计语言的角度，可以更加便捷而准确地理解算法的确定性与可行性。具体地，一个算法满足确定性与可行性，当且仅当它可以通过程序设计语言精确地描述，比如，起泡排序算法可以具体地描述和实现为代码1.1中的函数bubblesort1A()，其中“读取某一元素的内容”、“修改某一元素的内容”、“比较两个元素的大小”、“逻辑表达式求值”以及“根据逻辑判断确定分支转向”等等，都属于现代电子计算机所支持的基本操作。

■ 有穷性与正确性

不难理解，任意算法都应在执行有限次基本操作之后终止并给出输出，此即所谓算法的有穷性（finiteness）。进一步地，算法不仅应该迟早会终止，而且所给的输出还应该能够符合由问题本身在事先确定的条件，此即所谓算法的正确性（correctness）。

对以上前两个算法实例而言，在针对任一输入实例的计算过程中，每条基本操作语句仅执行一次，故其有穷性不证自明。另外，根据勾股定理以及平行等比原理，其正确性也一目了然。然而对于更为复杂的算法，这两条性质的证明往往颇需费些周折（习题[1-27]和[1-28]），有些问题甚至尚无定论（习题[1-29]）。即便是简单的起泡排序，bubblesort1A()算法的有穷性和正确性也不是由代码1.1自身的结构直接保证的。以下就以此为例做一分析。

■ 起泡排序

图1.3给出了bubblesort1A()的一次具体执行过程和排序结果，然而严格地说，这远不足以证明起泡排序就是一个名副其实的算法。比如，对于任意一组整数，经过若干趟的起泡交换之后该算法是否总能完成排序？事实上，即便是其有穷性也值得怀疑。就代码结构而言，只有在前一趟扫描交换中未做任何元素交换的情况下，外层循环才会因条件“!sorted”不再满足而退出。但是，这一情况对任何输入实例都总能出现吗？反过来，是否存在某一（某些）输入序列，无论做多少趟起泡交换也无济于事？这种担心并非毫无道理。细心的读者或许已注意到，在起泡交换的过程中，尽管多数时候元素会朝着各自的最终位置不断靠近，但有的时候某些元素也的确会暂时朝着远离自己应处位置的方向移动（习题[1-3]）。

证明算法有穷性和正确性的一个重要技巧，就是从适当的角度审视整个计算过程，并找出其所具有的某种不变性和单调性。其中的单调性通常是指，问题的有效规模会随着算法的推进不断递减。不变性则不仅应在算法初始状态下自然满足，而且应与最终的正确性相呼应——当问题的有效规模缩减到0时，不变性应随即等价于正确性。

那么，具体到bubblesort1A()算法，其单调性和不变性应如何定义和体现呢？

反观图1.3不难看出，每经过一趟扫描交换，尽管并不能保证序列立即达到整体有序，但从“待求解问题的规模”这一角度来看，整体的有序性必然有所改善。以全局最大的元素（图1.3中的整数7）为例，在第一趟扫描交换的过程中，一旦触及该元素，它必将与后续的所有元素依次交换。于是如图1.3(b)所示，经过第一趟扫描之后，该最大元素必然就位；而且在此后的各趟扫描交换中，该元素将绝不会参与任何交换。这就意味着，经过一趟扫描交换之后，我们只需要关注前面更小的那n - 1个元素。实际上，这一结论对后续的各趟扫描交换也都成立——考查图1.3(c~g)中的元素6~2，不难验证这一点。

于是，起泡排序算法的不变性和单调性可分别概括为：经过k趟扫描交换之后，最大的前k个元素必然就位；经过k趟扫描交换之后，待求解问题的有效规模将缩减至n - k。

反观如代码1.1所示的bubblesort1A()算法，外层while循环会不断缩减待排序序列的有效长度n。现在我们已经可以理解，该算法之所以能够如此处理，正是基于以上不变性和单调性。

特别地，初始状态下k = 0，这两条性质都自然满足。另一方面，由以上单调性可知，无论如何，至多经n - 1趟扫描交换后，问题的有效规模必将缩减至1。此时，仅含单个元素的序列，有序性不言而喻；而由该算法的不变性，其余n - 1个元素在此前的n - 1步迭代中业已相继就位。因此，算法不仅必然终止，而且输出序列必然整体有序，其有穷性与正确性由此得证。

■ 退化与鲁棒性

同一问题往往不限于一种算法，而同一算法也常常会有多种实现方式，因此除了以上必须具备的基本属性，在应用环境中还需从实用的角度对不同算法及其不同版本做更为细致考量和取舍。这些细致的要求尽管应纳入软件工程的范畴，但也不失为成熟算法的重要标志。

比如其中之一就是，除一般性情况外，实用的算法还应能够处理各种极端的输入实例。仍以排序问题为例，极端情况下待排序序列的长度可能不是正数（参数n = 0甚至n < 0），或者反过来长度达到或者超过系统支持的最大值（n = INT_MAX），或者A[]中的元素不见得互异甚至全体相等，以上种种都属于所谓的退化（degeneracy）情况。算法所谓的鲁棒性（robustness），就是要求能够尽可能充分地应对此类情况。请读者自行验证，对于以上退化情况，代码1.1中bubblesort1A()算法依然可以正确返回而不致出现异常。

■ 重用性

从实用角度评判不同算法及其不同实现方式时，可采用的另一标准是：算法的总体框架能否便捷地推广至其它场合。仍以起泡排序为例。实际上，起泡算法的正确性与所处理序列中元素的类型关系不大，无论是对于float、char或其它类型，只要元素之间可以比较大小，算法的整体框架就依然可以沿用。算法模式可推广并适用于不同类型基本元素的这种特性，即是重用性的一种典型形式。很遗憾，代码1.1所实现的bubblesort1A()算法尚不满足这一要求；而稍后的第2章和第3章，将使包括起泡排序在内的各种排序算法具有这一特性。

1.1.5 算法效率

■ 可计算性

相信本书的读者已经学习并掌握了至少一种高级程序设计语言，如C、C++或Java等。学习程序设计语言的目的，在于学会如何编写合法（即合乎特定程序语言的语法）的程序，从而保证编写的程序或者能够经过编译和链接生成执行代码，或者能够由解释器解释执行。然而从通过计算有效解决实际问题的角度来看，这只是第一个层次，仅仅做到语法正确还远远不够。很遗憾，算法所应具备的更多基本性质，合法的程序并非总是自然具备。

以前面提到的有穷性为例，完全合乎语法的程序却往往未必能够满足。相信每一位编写过程序的读者都有过这样的体验：很多合法的程序可以顺利编译链接，但在实际运行的过程中却因无穷循环或递归溢出导致异常。更糟糕的是，就大量的应用问题而言，根本就不可能设计出必然终止的算法。从这个意义讲，它们都属于不可解的问题。当然，关于此类问题的界定和研究，应归入可计算性（computability）理论的范畴，本书将不予过多涉及。

■ **难解性**

实际上我们不仅需要确定，算法对任何输入都能够在有穷次操作之后终止，而且更加关注该过程所需的时间。很遗憾，很多算法即便满足有穷性，但在终止之前所花费的时间成本却太高。比如，理论研究的成果显示，大量问题的最低求解时间成本，都远远超出目前实际系统所能提供的计算能力。同样地，此类难解性（intractability）问题，在本书中也不予过多讨论。

■ **计算效率**

在“编写合法程序”这一基础之上，本书将更多地关注于非“不可解和难解”的一般性问题，并讨论如何高效率地解决这一层面的计算问题。为此，首先需要确立一种尺度，用以从时间和空间等方面度量算法的计算成本，进而依此尺度对不同算法进行比较和评判。当然，更重要的是研究和归纳算法设计与实现过程中的一般性规律与技巧，以编写出效率更高、能够处理更大规模数据的程序。这两点既是本书的基本主题，也是贯穿始终的主体脉络。

■ **数据结构**

由上可知，无论是算法的初始输入、中间结果还是最终输出，在计算机中都可以数据的形式表示。对于数据的存储、组织、转移及变换等操作，不同计算模型和平台环境所支持的具体形式不尽相同，其执行效率将直接影响和决定算法的整体效率。数据结构这一学科正是以“数据”这一信息的表现形式为研究对象，旨在建立支持高效算法的数据信息处理策略、技巧与方法。要做到根据实际应用需求自如地设计、实现和选用适当的数据结构，必须首先对算法设计的技巧以及相应数据结构的特性了然于心，这些也是本书的重点与难点。

§1.2 复杂度度量

算法的计算成本涵盖诸多方面，为确定计算成本的度量标准，我们不妨先从计算速度这一主要因素入手。具体地，如何度量一个算法所需的计算时间呢？

1.2.1 时间复杂度

上述问题并不容易直接回答，原因在于，运行时间是由多种因素综合作用而决定的。首先，即使是同一算法，对于不同的输入所需的运行时间并不相同。以排序问题为例，输入序列的规模、其中各元素的数值以及次序均不确定，这些因素都将影响到排序算法最终的运行时间。为针对运行时间建立起一种可行、可信的评估标准，我们不得不首先考虑其中最为关键的因素。其中，问题实例的规模往往是决定计算成本的主要因素。一般地，问题规模越接近，相应的计算成本也越接近；而随着问题规模的扩大，计算成本通常也呈上升趋势。

如此，本节开头所提的问题即可转化为：随着输入规模的扩大，算法的执行时间将如何增长？执行时间的这一变化趋势可表示为输入规模的一个函数，称作该算法的时间复杂度（time complexity）。具体地，特定算法处理规模为n的问题所需的时间可记作T(n)。

细心的读者可能注意到，根据规模并不能唯一确定具体的输入，规模相同的输入通常都有多个，而算法对其进行处理所需时间也不尽相同。仍以排序问题为例，由n个元素组成的输入序列有n!种，有时所有元素都需交换，有时却无需任何交换（习题[1-3]）。故严格说来，以上定义的T(n)并不明确。为此需要再做一次简化，即从保守估计的角度出发，在规模为n的所有输入中选择执行时间最长者作为T(n)，并以T(n)度量该算法的时间复杂度。

1.2.2 渐进复杂度

至此，对于同一问题的两个算法A和B，通过比较其时间复杂度$T_A(n)$和$T_B(n)$，即可评价二者对于同一输入规模n的计算效率高低。然而，藉此还不足以就其性能优劣做出总体性的评判，比如对于某些问题，一些算法更适用于小规模输入，而另一些则相反（习题[1-5]）。

幸运的是，在评价算法运行效率时，我们往往可以忽略其处理小规模问题时的能力差异，转而关注其在处理更大规模问题时的表现。其中的原因不难理解，小规模问题所需的处理时间本来就相对更少，故此时不同算法的实际效率差异并不明显；而在处理更大规模的问题时，效率的些许差异都将对实际执行效果产生巨大的影响。这种着眼长远、更为注重时间复杂度的总体变化趋势和增长速度的策略与方法，即所谓的渐进分析（asymptotic analysis）。

那么，针对足够大的输入规模n，算法执行时间T(n)的渐进增长速度，应如何度量和评价呢？

■ 大$\mathcal{O}$记号

同样地出于保守的估计，我们首先关注T(n)的渐进上界。为此可引入所谓“大$\mathcal{O}$记号”（big-$\mathcal{O}$ notation）。具体地，若存在正的常数c和函数f(n)，使得对任何n >> 2都有

$$T(n) \le c \cdot f(n)$$

则可认为在n足够大之后，f(n)给出了T(n)增长速度的一个渐进上界。此时，记之为：

$$T(n) = \mathcal{O}(f(n))$$

由这一定义，可导出大$\mathcal{O}$记号的以下性质：

(1) 对于任一常数c > 0，有$\mathcal{O}(f(n)) = \mathcal{O}(c \cdot f(n))$

(2) 对于任意常数a > b > 0，有$\mathcal{O}(n^a + n^b) = \mathcal{O}(n^a)$

前一性质意味着，在大$\mathcal{O}$记号的意义下，函数各项正的常系数可以忽略并等同于1。后一性质则意味着，多项式中的低次项均可忽略，只需保留最高次项。可以看出，大$\mathcal{O}$记号的这些性质的确体现了对函数总体渐进增长趋势的关注和刻画。

■ 环境差异

在实际环境中直接测得的执行时间T(n)，虽不失为衡量算法性能的一种指标，但作为评判不同算法性能优劣的标准，其可信度值得推敲。事实上，即便是同一算法、同一输入，在不同的硬件平台上、不同的操作系统中甚至不同的时间，所需要的计算时间都不尽相同。因此，有必要按照超脱于具体硬件平台和软件环境的某一客观标准，来度量算法的时间复杂度，并进而评价不同算法的效率差异。

■ 基本操作

一种自然且可行的解决办法是，将时间复杂度理解为算法中各条指令的执行时间之和。在图灵机（Turing Machine, TM）和随机存储机（Random Access Machine, RAM）等计算模型[4]中，指令语句均可分解为若干次基本操作，比如算术运算、比较、分支、子程序调用与返回等；而在大多数实际的计算环境中，每一次这类基本操作都可在常数时间内完成。

如此，不妨将T(n)定义为算法所执行基本操作的总次数。也就是说，T(n)决定于组成算法的所有语句各自的执行次数，以及其中所含基本操作的数目。以代码1.1中起泡排序bubblesort1A()算法为例，若将该算法处理长度为n的序列所需的时间记作T(n)，则按照上述分析，只需统计出该算法所执行基本操作的总次数，即可确定T(n)的上界。

■ 起泡排序

bubblesort1A()算法由内、外两层循环组成。内循环从前向后，依次比较各对相邻元素，如有必要则将其交换。故在每一轮内循环中，需要扫描和比较n - 1对元素，至多需要交换n - 1对元素。元素的比较和交换，都属于基本操作，故每一轮内循环至多需要执行2(n - 1)次基本操作。另外，根据1.1.4节对该算法正确性的分析结论，外循环至多执行n - 1轮。因此，总共需要执行的基本操作不会超过$2(n - 1)^2$次。若以此来度量该算法的时间复杂度，则有

$$T(n) = \mathcal{O}(2(n-1)^2)$$

根据大$\mathcal{O}$记号的性质，可进一步简化和整理为：

$$T(n) = \mathcal{O}(2n^2 - 4n + 2) = \mathcal{O}(2n^2) = \mathcal{O}(n^2)$$

■ 最坏、最好与平均情况

由上可见，以大$\mathcal{O}$记号形式表示的时间复杂度，实质上是对算法执行时间的一种保守估计——对于规模为n的任意输入，算法的运行时间都不会超过$\mathcal{O}(f(n))$。比如，“起泡排序算法复杂度$T(n) = \mathcal{O}(n^2)$”意味着，该算法处理任何序列所需的时间绝不会超过$\mathcal{O}(n^2)$。的确需要这么长计算时间的输入实例，称作最坏实例或最坏情况（worst case）。

需强调的是，这种保守估计并不排斥更好情况甚至最好情况（best case）的存在和出现。比如，对于某些输入序列，起泡排序算法的内循环的执行轮数可能少于n-1，甚至只需执行一轮（习题[1-3]）。当然，有时也需要考查所谓的平均情况（average case），也就是按照某种约定的概率分布，将规模为n的所有输入对应的计算时间加权平均。

比较而言，“最坏情况复杂度”是人们最为关注且使用最多的，在一些特殊的场合甚至成为唯一的指标。比如控制核电站运转、管理神经外科手术室现场的系统而言，从最好或平均角度评判算法的响应速度都不具有任何意义，在最坏情况下的响应速度才是唯一的指标。

■ 大Ω记号

为了对算法的复杂度最好情况做出估计，需要借助另一个记号。如果存在正的常数c和函数g(n)，使得对于任何n >> 2都有

$$T(n) \geq c \cdot g(n)$$

就可以认为，在n足够大之后，g(n)给出了T(n)的一个渐进下界。此时，我们记之为：

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

这里的Ω称作“大Ω记号”（big-omega notation）。与大$\mathcal{O}$记号恰好相反，大Ω记号是对算法执行效率的乐观估计——对于规模为n的任意输入，算法的运行时间都不低于Ω(g(n))。比如，即便在最好情况下，起泡排序也至少需要$T(n) = \Omega(n)$的计算时间（习题[1-4]）。

■ 大Θ记号

借助大$\mathcal{O}$记号、大Ω记号，可以对算法的时间复杂度作出定量的界定，亦即，从渐进的趋势看，T(n)介于Ω(g(n))与$\mathcal{O}(f(n))$之间。若恰巧出现g(n) = f(n)的情况，则可以使用另一记号来表示。

如果存在正的常数$c_1 < c_2$和函数h(n)，使得对于任何n >> 2都有

$$c_1 \cdot h(n) \leq T(n) \leq c_2 \cdot h(n)$$

就可以认为在n足够大之后，h(n)给出了T(n)的一个确界。此时，我们记之为：

$$T(n) = \Theta(h(n))$$

这里的Θ称作“大Θ记号”（big-theta notation），它是对算法复杂度的准确估计——对于规模为n的任何输入，算法的运行时间T(n)都与Θ(h(n))同阶。

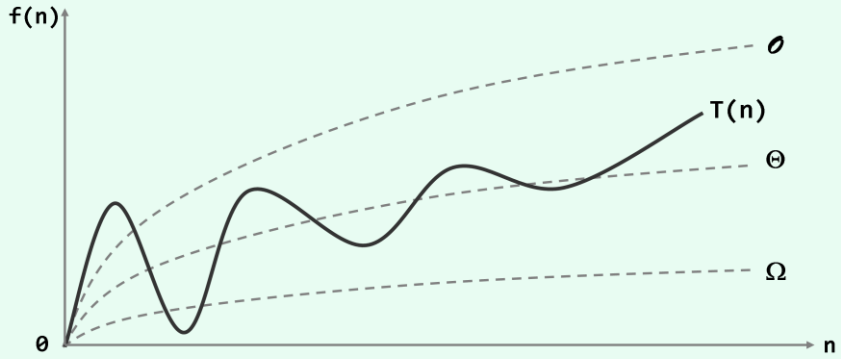

图1.4 大O记号、大Ω记号和大Θ记号

以上主要的这三种渐进复杂度记号之间的联系与区别，可直观地由图1.4示意。

1.2.3 空间复杂度

除了执行时间的长短，算法所需存储空间的多少也是衡量其性能的一个重要方面，此即所谓的空间复杂度（space complexity）。实际上，以上针对时间复杂度所引入的几种渐进记号，也适用于对空间复杂度的度量，其原理及方法基本相同，不再赘述。

需要注意的是，为了更为客观地评价算法性能的优劣，除非特别申明，空间复杂度通常并不计入原始输入本身所占用的空间——对于同一问题，这一指标对任何算法都是相同的。反之，其它（如转储、中转、索引、映射、缓冲等）各个方面所消耗的空间，则都应计入。

另外，很多时候我们都是更多地甚至仅仅关注于算法的时间复杂度，而不必对空间复杂度做专门的考查。这种简便评测方式的依据，来自于以下事实：就渐进复杂度的意义而言，在任一算法的任何一次运行过程中所消耗的存储空间，都不会多于其间所执行基本操作的累计次数。

实际上根据定义，每次基本操作所涉及的存储空间，都不会超过常数规模；纵然每次基本操作所占用或访问的存储空间都是新开辟的，整个算法所需的空间总量，也不过与基本操作的次数同阶。从这个意义上说，时间复杂度本身就是空间复杂度的一个天然的上界。

当然，对空间复杂度的分析也有其自身的意义，尤其在对空间效率非常在乎的应用场合中，或当问题的输入规模极为庞大时，由时间复杂度所确立的平凡上界已经难以令人满意。这类情况下，人们将更为精细地考查不同算法的空间效率，并尽力在此方面不断优化。本书的后续章节，将结合一些实际问题介绍相关的方法与技巧。

§1.3 复杂度分析

在明确了算法复杂度的度量标准之后，如何分析具体算法的复杂度呢？1.2.2节所引入的三种记号中，大O记号是最基本的，也是最常用到的。从渐进分析的角度，大O记号将各算法的复杂度由低到高划分为若干层次级别。以下依次介绍若干典型的复杂度级别，并介绍主要的分析方法与技巧。读者可参照以下介绍的方法，做进一步的练习（习题[1-30]和[1-32]）。

1.3.1 常数$O(1)$

■ 问题与算法

考查如下常规元素的选取问题，该问题一种解法如算法1.3所示。

```
ordinaryElement(S[], n) //从n ≥ 3个互异整数中，除最大、最小者以外，任取一个“常规元素”
   任取的三个元素x, y, z ∈ S; //这三个元素亦必互异
   通过比较，对它们做排序; //设经排序后，依次重命名为：a < b < c
   输出b;
```

算法1.3 取非极端元素

该算法的正确性不言而喻，但它需要运行多少时间？与输入的规模n有何联系？

■ 复杂度

既然S是有限集，故其中的最大、最小元素各有且仅有一个。因此，无论S的规模有多大，在任意三个元素中至少都有一个是非极端元素。不妨取前三个元素x = S[0]、y = S[1]和z = S[2]，这一步只需执行三次（从特定单元读取元素的）基本操作，耗费$O(3)$时间。接下来，为确定这三个元素的大小次序，最多需要做三次比较（习题[2-37]），也需$O(3)$时间。最后，输出居中的非极端元素只需$O(1)$时间。因此综合起来，算法1.3的运行时间为：

$$T(n) = O(3) + O(3) + O(1) = O(7) = O(1)$$

运行时间可表示和度量为$T(n) = O(1)$的这一类算法，统称作“常数时间复杂度算法”（constant-time algorithm）。此类算法已是最为理想的，因为不可能奢望“不劳而获”。

一般地，仅含一次或常数次基本操作的算法（如算法1.1和算法1.2）均属此类。此类算法通常不含循环、分支、子程序调用等，但也不能仅凭语法结构的表面形式一概而论（习题[1-7]）。

采用1.2.3节的分析方法不难看出，除了输入数组等参数之外，该算法仅需常数规模的辅助空间。此类仅需$O(1)$辅助空间的算法，亦称作就地算法（in-place algorithm）。

1.3.2 对数$O(\log n)$

■ 问题与算法

考查如下问题：对于任意非负整数，统计其二进制展开中数位1的总数。

该问题的一个算法可实现如代码1.2所示。该算法使用一个计数器ones记录数位1的数目，其初始值为0。随后进入一个循环：通过二进制位的与（and）运算，检查n的二进制展开的最低位，若该位为1则累计至ones。由于每次循环都将n的二进制展开右移一位，故整体效果等同于逐个检验所有数位是否为1，该算法的正确性也不难由此得证。

以$n = 441_{(10)} = 110111001_{(2)}$为例，采用以上算法，变量n与计数器ones在计算过程中的演变过程如表1.1所示。

表1.1 countOnes(441)的执行过程

十 进 制	二 进 制	数位1计数
441	110111001	0
220	11011100	1
110	1101110	1
55	110111	
27	11011	2
13	1101	3
6	110	4
3	11	4
1	1	5
0	0	6

```
1 int countOnes ( unsigned int n ) { //统计整数二进制展开中数位1的总数：O(logn)
2    int ones = 0; //计数器复位
3    while ( 0 < n ) { //在n缩减至0之前，反复地
4       ones += ( 1 & n ); //检查最低位，若为1则计数
5       n >>= 1; //右移一位
6    }
7    return ones; //返回计数
8 } //等效于glibc的内置函数int __builtin_popcount (unsigned int n)
```

代码1.2 整数二进制展开中数位1总数的统计

■ **复杂度**

根据右移运算的性质，每右移一位，n都至少缩减一半。也就是说，至多经过$1 + \lfloor \log_2 n \rfloor$次循环，n必然缩减至0，从而算法终止。实际上从另一角度来看，$1 + \lfloor \log_2 n \rfloor$恰为n二进制展开的总位数，每次循环都将其右移一位，总的循环次数自然也应是$1 + \lfloor \log_2 n \rfloor$。后一解释，也可以从表1.1中n的二进制展开一列清晰地看出。

无论是该循环体之前、之内还是之后，均只涉及常数次（逻辑判断、位与运算、加法、右移等）基本操作。因此，countOnes()算法的执行时间主要由循环的次数决定，亦即：

$$\mathcal{O}(1 + \lfloor \log_2 n \rfloor) = \mathcal{O}(\lfloor \log_2 n \rfloor) = \mathcal{O}(\log_2 n)$$

由大$\mathcal{O}$记号定义，在用函数$\log_r n$界定渐进复杂度时，常底数r的具体取值无所谓（习题[1-8]），故通常不予专门标出而笼统地记作$\log n$。比如，尽管此处底数为常数2，却可直接记作$\mathcal{O}(\log n)$。此类算法称作具有“对数时间复杂度”（logarithmic-time algorithm）。

实际上，代码1.2中的countOnes()算法仍有巨大的改进余地（习题[1-12]）。

■ **对数多项式复杂度**

更一般地，凡运行时间可以表示和度量为$T(n) = \mathcal{O}(\log^c n)$形式的这一类算法（其中常数$c > 0$），均统称作“对数多项式时间复杂度的算法”（polylogarithmic-time algorithm）。上述$\mathcal{O}(\log n)$即$c = 1$的特例。此类算法的效率虽不如常数复杂度算法理想，但从多项式的角度看仍能无限接近于后者（习题[1-9]），故也是极为高效的一类算法。

1.3.3 线性$\mathcal{O}(n)$

■ **问题与算法**

考查如下问题：计算给定n个整数的总和。该问题可由代码1.3中的算法sumI()解决。

```
1 int sumI ( int A[], int n ) { //数组求和算法（迭代版）
2    int sum = 0; //初始化累计器，O(1)
3    for ( int i = 0; i < n; i++ ) //对全部共O(n)个元素，逐一
4       sum += A[i]; //累计，O(1)
5    return sum; //返回累计值，O(1)
6 } //O(1) + O(n)*O(1) + O(1) = O(n+2) = O(n)
```

代码1.3 数组元素求和算法sumI()

■ 复杂度

sumI()算法的正确性一目了然，它需要运行多少时间呢？

首先，对s的初始化需要$\mathcal{O}(1)$时间。算法的主体部分是一个循环，每一轮循环中只需进行一次累加运算，这属于基本操作，可在$\mathcal{O}(1)$时间内完成。每经过一轮循环，都将一个元素累加至s，故总共需要做n轮循环，于是该算法的运行时间应为：

$$\mathcal{O}(1) + \mathcal{O}(1) \times n = \mathcal{O}(n + 1) = \mathcal{O}(n)$$

凡运行时间可以表示和度量为$T(n) = \mathcal{O}(n)$形式的这一类算法，均统称作“线性时间复杂度算法”（linear-time algorithm）。比如，算法1.2只需略加修改，即可解决“n等分给定线段”问题，这个通用版本相对于输入n就是一个线性时间复杂度的算法。

也就是说，对于输入的每一单元，此类算法平均消耗常数时间。就大多数问题而言，在对输入的每一单元均至少访问一次之前，不可能得出解答。以数组求和为例，在尚未得知每一元素的具体数值之前，绝不可能确定其总和。故就此意义而言，此类算法的效率亦足以令人满意。

1.3.4 多项式$\mathcal{O}$(polynomial(n))

若运行时间可以表示和度量为$T(n) = \mathcal{O}(f(n))$的形式，而且f(x)为多项式，则对应的算法称作“多项式时间复杂度算法”（polynomial-time algorithm）。比如根据1.2.2节的分析，1.1.3节所实现起泡排序bubblesort1A()算法的时间复杂度应为$T(n) = \mathcal{O}(n^2)$，故该算法即属于此类。当然，以上所介绍的线性时间复杂度算法，也属于多项式时间复杂度算法的特例，其中线性多项式$f(n) = n$的次数为1。

在算法复杂度理论中，多项式时间复杂度被视作一个具有特殊意义的复杂度级别。多项式级的运行时间成本，在实际应用中一般被认为是可接受的或可忍受的。某问题若存在一个复杂度在此范围以内的算法，则称该问题是可有效求解的或易解的（tractable）。

请注意，这里仅要求多项式的次数为一个正的常数，而并未对其最大取值范围设置任何具体上限，故实际上该复杂度级别涵盖了很大的一类算法。比如，从理论上讲，复杂度分别为$\mathcal{O}(n^2)$和$\mathcal{O}(n^{2012})$算法都同属此类，尽管二者实际的计算效率有天壤之别。之所以如此，是因为相对于以下的指数级复杂度，二者之间不超过多项式规模的差异只是小巫见大巫。

1.3.5 指数$\mathcal{O}(2^n)$

■ 问题与算法

考查如下问题：在禁止超过1位的移位运算的前提下，对任意非负整数n，计算幂2^n。

```
1 __int64 power2BF_I ( int n ) { //幂函数2^n算法（蛮力迭代版），n >= 0
2    __int64 pow = 1; //O(1)：累积器初始化为2^0
3    while ( 0 < n -- ) //O(n)：迭代n轮，每轮都
4       pow <<= 1; //O(1)：将累积器翻倍
5    return pow; //O(1)：返回累积器
6 } //O(n) = O(2^r)，r为输入指数n的比特位数
```

代码1.4 幂函数算法（蛮力迭代版）

■ 复杂度

如代码1.4所示的算法power2BF_I()由n轮迭代组成，各需做一次累乘和一次递减，均属于基本操作，故整个算法共需$O(n)$时间。若以输入指数n的二进制位数$r = 1 + \lfloor \log_2 n \rfloor$作为输入规模，则运行时间为$O(2^r)$。稍后在1.4.3节我们将看到，该算法仍有巨大的改进余地。

一般地，凡运行时间可以表示和度量为$T(n) = O(a^n)$形式的算法（$a > 1$），均属于“指数时间复杂度算法”（exponential-time algorithm）。

■ 从多项式到指数

从常数、对数、线性、平方到多项式时间复杂度，算法效率的差异还在可接受的范围。然而，在多项式与指数时间复杂度之间，却有着一道巨大的鸿沟。当问题规模较大后，指数复杂度算法的实际效率将急剧下降，计算时间之长很快就会达到令人难以忍受的地步。因此通常认为，指数复杂度算法无法真正应用于实际问题中，它们不是有效算法，甚至不能称作算法。相应地，不存在多项式复杂度算法的问题，也称作难解的（intractable）问题。

需注意的是，在问题规模不大时，指数复杂度反而可能在较长一段区间内均低于多项式复杂度。比如，在$1 \le n \le 116,690$以内，指数复杂度1.0001^n反而低于多项式复杂度$n^{1.0001}$；但前者迟早必然超越后者，且随着n的进一步增大，二者的差距无法保持在多项式倍的范围。因此，从渐进复杂度的角度看，多项式与指数是无法等量齐观的两个截然不同的量级。

实际上很遗憾，绝大多数计算问题并不存在多项式时间的算法（习题[1-16]、[1-23]和[1-27]），也就是说，试图求解此类问题的任一算法，都至少需要运行指数量级的时间。特别地，很多问题甚至需要无穷的时间，由于有穷性不能满足或者尚未得到证明（习题[1-29]），也可以说不存在解决这些问题的算法。不过，这类问题均不属于本书的讨论范围。

1.3.6 复杂度层次

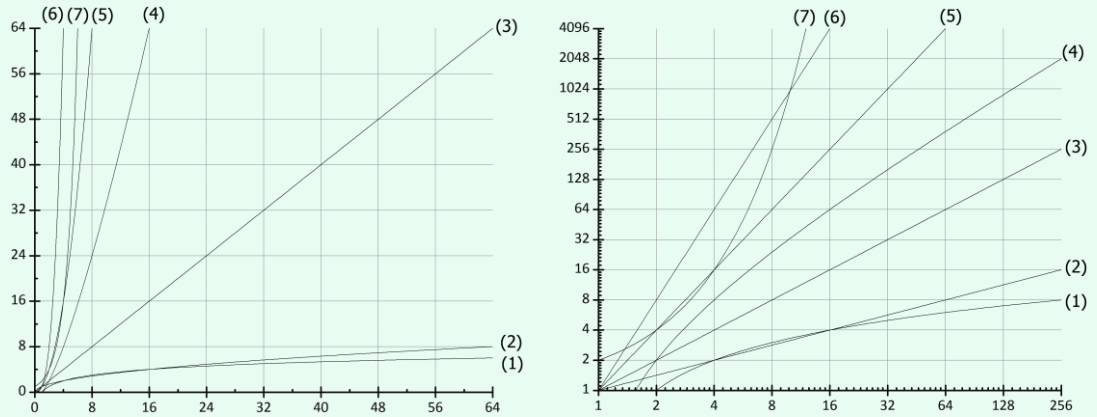

图1.5 复杂度的典型层次：(1)~(7)依次为$O(\log n)$、$O(\sqrt{n})$、$O(n)$、$O(n\log n)$、$O(n^2)$、$O(n^3)$和$O(2^n)$

利用大O记号，不仅可以定量地把握算法复杂度的主要部分，而且可以定性地由低至高将复杂度划分为若干层次。典型的复杂度层次包括$O(1)$、$O(\log^* n)$、$O(\log\log n)$、$O(\log n)$、$O(\mathrm{sqrt}(n))$、$O(n)$、$O(n\log^* n)$、$O(n\log\log n)$、$O(n\log n)$、$O(n^2)$、$O(n^3)$、$O(n^c)$、$O(2^n)$等，图1.5绘出了其中七个层次复杂度函数对应的渐进增长趋势。

请注意，在图1.5的左图中，层次(7)的2^n显得比层次(6)的n^3更低，但这只是在问题规模n较小时的暂时现象。从覆盖更大范围的右图可以看出，当问题规模不小于10之后，层次(7)的复杂度将远远高于层次(6)。另外，右图还采用了双对数坐标，将层次(6)、(5)、(3)和(2)表示为直线，从而更为清晰地显示出各层次之间的高低关系。

1.3.7 输入规模

对算法复杂度的界定，都是相对于问题的输入规模而言的。然而，细心的读者可能已经注意到，不同的人在不同场合下关于“输入规模”的理解、定义和度量可能不尽相同，因此也可能导致复杂度分析的结论有所差异。比如，1.3.2节中关于“countOnes()算法的复杂度为$\mathcal{O}(\log n)$”的结论，是相对于输入整数本身的数值n而言；而若以n二进制展开的宽度$r = 1 + \lfloor \log_2 n \rfloor$作为输入规模，则应为线性复杂度$\mathcal{O}(r)$。再如，1.3.5节中关于“power2BF_I()算法的复杂度为$\mathcal{O}(2^r)$”的结论，是相对于输入指数n的二进制数位r而言；而若以n本身的数值作为输入规模，却应为线性复杂度$\mathcal{O}(n)$。

严格地说，所谓待计算问题的输入规模，应严格定义为“用以描述输入所需的空间规模”。因此就上述两个例子而言，将输入参数n二进制展开的宽度r作为输入规模更为合理。也就是说，将这两个算法的复杂度界定为$\mathcal{O}(r)$和$\mathcal{O}(2^r)$更妥。对应地，以输入参数n本身的数值作为基准而得出的$\mathcal{O}(\log n)$和$\mathcal{O}(n)$复杂度，则应分别称作伪对数的（pseudo-logarithmic）和伪线性的（pseudo-linear）复杂度。

§1.4 *递归

分支转向是算法的灵魂；函数和过程及其之间的相互调用，是在经过抽象和封装之后，实现分支转向的一种重要机制；而递归则是函数和过程调用的一种特殊形式，即允许函数和过程进行自我调用。因其高度的抽象性和简洁性，递归已成为多数高级程序语言普遍支持的一项重要特性。比如在C++语言中，递归调用（recursive call）就是某一方法调用自身。这种自我调用通常是直接的，即在函数体中包含一条或多条调用自身的语句。递归也可能以间接的形式出现，即某个方法首先调用其它方法，再辗转通过其它方法的相互调用，最终调用起始的方法自身。

递归的价值在于，许多应用问题都可简洁而准确地描述为递归形式。以操作系统为例，多数文件系统的目录结构都是递归定义的。具体地，每个文件系统都有一个最顶层的目录，其中可以包含若干文件和下一层的子目录；而在每一子目录中，也同样可能包含若干文件和再下一层的子目录；如此递推，直至不含任何下层的子目录。通过如此的递归定义，文件系统中的目录就可以根据实际应用的需要嵌套任意多层（只要系统的存储资源足以支持）。

递归也是一种基本而典型的算法设计模式。这一模式可以对实际问题中反复出现的结构和形式做高度概括，并从本质层面加以描述与刻画，进而导出高效的算法。从程序结构的角度看，递归模式能够统筹纷繁多变的具体情况，避免复杂的分支以及嵌套的循环，从而更为简明地描述和实现算法，减少代码量，提高算法的可读性，保证算法的整体效率。

以下将从递归的基本模式入手，循序渐进地介绍如何选择和应用（线性递归、二分递归和多分支递归等）不同的递归形式，以实现（遍历、分治等）算法策略，以及如何利用递归跟踪和递推方程等方法分析递归算法的复杂度。

1.4.1 线性递归

■ 数组求和

仍以1.3.3节的数组求和问题为例。易见，若n = 0则总和必为0，这也是最终的平凡情况；否则一般地，总和可理解为前n - 1个整数（A[0, n - 1)）之和，再加上末元素（A[n - 1]）。按这一思路，可基于线性递归模式，设计出另一sum()算法如代码1.5所示。

```
1 int sum ( int A[], int n ) { //数组求和算法（线性递归版）
2    if ( 1 > n ) //平凡情况，递归基
3       return 0; //直接（非递归式）计算
4    else //一般情况
5       return sum ( A, n - 1 ) + A[n - 1]; //递归：前n - 1项之和，再累计第n - 1项
6 } //O(1)*递归深度 = O(1)*(n + 1) = O(n)
```

代码1.5 数组求和算法（线性递归版）

由此实例，可以看出保证递归算法有穷性的基本技巧：首先判断并处理n = 0之类的平凡情况，以免因无限递归而导致系统溢出。这类平凡情况统称“递归基”（base case of recursion）。平凡情况可能有多种，但至少要有一种（比如此处），且迟早必然会出现。

■ 线性递归

算法sum()可能朝着更深一层进行自我调用，且每一递归实例对自身的调用至多一次。于是，每一层次上至多只有一个实例，且它们构成一个线性的次序关系。此类递归模式因而称作“线性递归”（linear recursion），它也是递归的最基本形式。

这种形式中，应用问题总可分解为两个独立的子问题：其一对应于单独的某个元素，故可直接求解（比如A[n - 1]）；另一个对应于剩余部分，且其结构与原问题相同（比如A[0, n - 1)）。另外，子问题的解经简单的合并（比如整数相加）之后，即可得到原问题的解。

■ 减而治之

线性递归的模式，往往对应于所谓减而治之（decrease-and-conquer）的算法策略：递归每深入一层，待求解问题的规模都缩减一个常数，直至最终蜕化为平凡的小（简单）问题。

按照减而治之策略，此处随着递归的深入，调用参数将单调地线性递减。因此无论最初输入的n有多大，递归调用的总次数都是有限的，故算法的执行迟早会终止，即满足有穷性。当抵达递归基时，算法将执行非递归的计算（这里是返回0）。

1.4.2 递归分析

递归算法时间和空间复杂度的分析与常规算法很不一样，有其自身的规律和特定的技巧，以下介绍递归跟踪与递推方程这两种主要的方法。

■ 递归跟踪

作为一种直观且可视的方法，递归跟踪（recursion trace）可用以分析递归算法的总体运行时间与空间。具体地，就是按照以下原则，将递归算法的执行过程整理为图的形式：

① 算法的每一递归实例都表示为一个方框，其中注明了该实例调用的参数
② 若实例M调用实例N，则在M与N对应的方框之间添加一条有向联线

按上述约定，代码1.5中sum()算法的递归跟踪如图1.6所示。其中，sum()算法的每一递归实例分别对应于一个方框，并标有相应的调用参数。每发生一次递归调用，就从当前实例向下引出一条有向边，指向下层对应于新实例的方框。

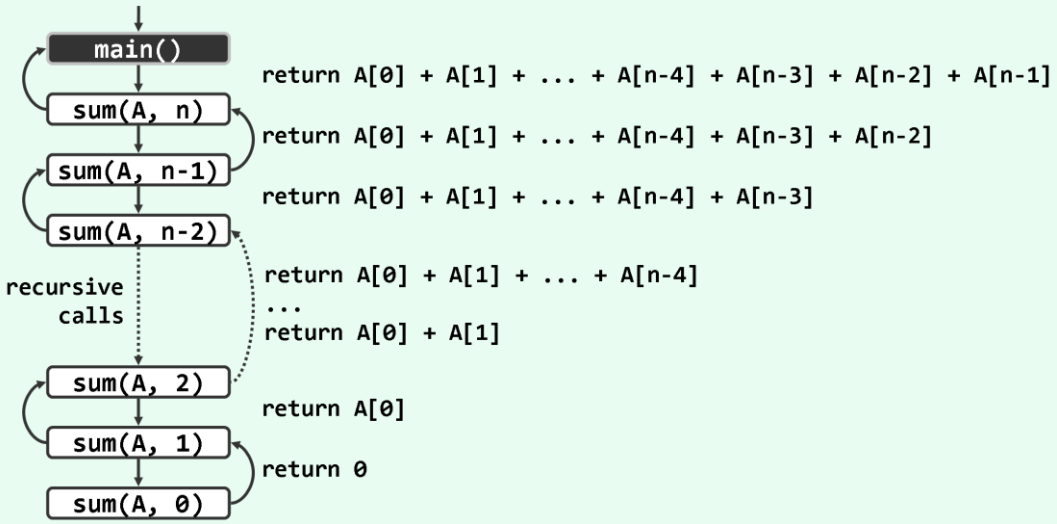

图1.6 对sum(A, 5)的递归跟踪分析

该图清晰地给出了算法执行的整个过程：首先对参数n进行调用，再转向对参数n - 1的调用，再转向对参数n - 2的调用，...，直至最终的参数0。在抵达递归基后不再递归，而是将平凡的解（长度为0数组的总和0）返回给对参数1的调用；累加上A[0]之后，再返回给对参数2的调用；累加上A[1]之后，继续返回给对参数3的调用；...；如此依次返回，直到最终返回给对参数n的调用，此时，只需累加A[n - 1]即得到整个数组的总和。

从图1.6可清楚地看出，整个算法所需的计算时间，应该等于所有递归实例的创建、执行和销毁所需的时间总和。其中，递归实例的创建、销毁均由操作系统负责完成，其对应的时间成本通常可以近似为常数，不会超过递归实例中实质计算步骤所需的时间成本，故往往均予忽略。为便于估算，启动各实例的每一条递归调用语句所需的时间，也可以计入被创建的递归实例的账上，如此我们只需统计各递归实例中非递归调用部分所需的时间。

具体地，就以上的sum()算法而言，每一递归实例中非递归部分所涉及的计算无非三类（判断n是否为0、累加sum(n - 1)与A[n - 1]、返回当前总和），且至多各执行一次。鉴于它们均属于基本操作，每个递归实例实际所需的计算时间都应为常数$\mathcal{O}(3)$。由图1.6还可以看出，对于长度为n的输入数组，递归深度应为n + 1，故整个sum()算法的运行时间为：

$$(n + 1) \times \mathcal{O}(3) = \mathcal{O}(n)$$

那么，sum()算法的空间复杂度又是多少呢？由图1.6不难看出，在创建了最后一个递归实例（即到达递归基）时，占用的空间量达到最大——准确地说，等于所有递归实例各自所占空间量的总和。这里每一递归实例所需存放的数据，无非是调用参数（数组A的起始地址和长度n）以及用于累加总和的临时变量。这些数据各自只需常数规模的空间，其总量也应为常数。故此可知，sum()算法的空间复杂度线性正比于其递归的深度，亦即$\mathcal{O}(n)$。

■ 递推方程

递归算法的另一常用分析方法，即递推方程（recurrence equation）法。与递归跟踪分析相反，该方法无需绘出具体的调用过程，而是通过对递归模式的数学归纳，导出复杂度定界函数的递推方程（组）及其边界条件，从而将复杂度的分析，转化为递归方程（组）的求解。

在总体思路上，该方法与微分方程法颇为相似：很多复杂函数的显式表示通常不易直接获得，但是它们的微分形式却往往遵循某些相对简洁的规律，通过求解描述这些规律的一组微分方程，即可最终导出原函数的显式表示。微分方程的解通常并不唯一，除非给定足够多的边界条件。类似地，为使复杂度定界函数的递推方程能够给出确定的解，也需要给定某些边界条件。以下我们将看到，这类边界条件往往可以通过对递归基的分析而获得。

仍以代码1.5 中线性递归版sum()算法为例，将该算法处理长度为n的数组所需的时间成本记作T(n)。我们将该算法的思路重新表述如下：为解决问题sum(A, n)，需递归地解决问题sum(A, n - 1)，然后累加上A[n - 1]。按照这一新的理解，求解sum(A, n)所需的时间，应该等于求解sum(A, n - 1)所需的时间，另加一次整数加法运算所需的时间。

根据以上分析，可以得到关于T(n)的如下一般性的递推关系：

$$T(n) = T(n - 1) + \mathcal{O}(1) = T(n - 1) + c_1, \quad \text{其中} c_1 \text{为常数}$$

另一方面，当递归过程抵达递归基时，求解平凡问题sum(A, 0)只需（用于直接返回0的）常数时间。如此，即可获得如下边界条件：

$$T(0) = \mathcal{O}(1) = c_2, \quad \text{其中} c_2 \text{为常数}$$

联立以上两个方程，最终可以解得：

$$T(n) = c_1 n + c_2 = \mathcal{O}(n)$$

这一结论与递归跟踪分析殊途同归。

另外，运用以上方法，同样也可以界定sum()算法的空间复杂度（习题[1-18]）。

1.4.3 递归模式

■ 多递归基

为保证有穷性，递归算法都必须设置递归基，且确保总能执行到。为此，针对每一类可能出现的平凡情况，都需设置对应的递归基，故同一算法的递归基可能（显式或隐式地）不止一个。

以下考查数组倒置问题，也就是将数组中各元素的次序前后翻转。比如，若输入数组为：

A[] = { 3, 1, 4, 1, 5, 9, 2, 6 }

则倒置后为：

A[] = { 6, 2, 9, 5, 1, 4, 1, 3 }

这里先介绍该问题的一个递归版算法，1.4.4节还将介绍另一等效的迭代版算法。无论何种实现，均由如下reverse()函数作为统一的启动入口。

```
1 void reverse ( int*, int, int ); //重载的倒置算法原型
2 void reverse ( int* A, int n ) //数组倒置（算法的初始入口，调用的可能是reverse()的递归版或迭代版）
3 { reverse ( A, 0, n - 1 ); } //由重载的入口启动递归或迭代算法
```

代码1.6 数组倒置算法的统一入口

借助线性递归不难解决这一问题，为此只需注意到并利用如下事实：为得到整个数组的倒置，可以先对换其首、末元素，然后递归地倒置除这两个元素以外的部分。按照这一思路，可实现如代码1.7所示的算法。通过递归跟踪可以证明（习题[1-31]），其时间复杂度为$O(n)$。

```
1 void reverse ( int* A, int lo, int hi ) { //数组倒置（多递归基递归版）
2    if ( lo < hi ) {
3       swap ( A[lo], A[hi] ); //交换A[lo]和A[hi]
4       reverse ( A, lo + 1, hi - 1 ); //递归倒置A(lo, hi)
5    } //else隐含了两种递归基
6 } //O(hi - lo + 1)
```

代码1.7 数组倒置的递归算法

■ 实现递归

在设计递归算法时，往往需要从多个角度反复尝试，方能确定对问题的输入及其规模的最佳划分方式。有时，还可能需要从不同的角度重新定义和描述原问题，使得经分解所得的子问题与原问题具有相同的语义形式。

例如，在代码1.7线性递归版reverse()算法中，通过引入参数lo和hi，使得对全数组以及其后各子数组的递归调用都统一为相同的语法形式。另外，还利用C++的函数重载（overload）机制定义了名称相同、参数表有别的另一函数reverse(A, n)，作为统一的初始入口。

■ 多向递归

递归算法中，不仅递归基可能有多个，递归调用也可能有多种可供选择的分支。以下的简单实例中，每一递归实例虽有多个可能的递归方向，但只能从中选择其一，故各层次上的递归实例依然构成一个线性次序关系，这种情况依然属于线性递归。至于一个递归实例可能执行多次递归调用的情况，稍后将于1.4.5节再做介绍。

再次讨论1.3.5节中，计算幂函数power(2, n) = 2^n的问题。按照线性递归的构思，该函数可以重新定义和表述如下：

$$\text{power2}(n) = \begin{cases} 1 & (\text{若}n = 0) \\ 2 \cdot \text{power2}(n-1) & (\text{否则}) \end{cases}$$

由此不难直接导出一个线性递归的算法，其复杂度与代码1.4中蛮力的power2BF_I()算法完全一样，总共需要做$O(n)$次递归调用（习题[1-13]）。但实际上，若能从其它角度分析该函数并给出新的递归定义，完全可以更为快速地完成幂函数的计算。以下就是一例：

$$\text{power2}(n) = \begin{cases} 1 & (\text{若}n = 0) \\ \text{power2}(\lfloor n/2 \rfloor)^2 \times 2 & (\text{若}n > 0\text{且为奇数}) \\ \text{power2}(\lfloor n/2 \rfloor)^2 & (\text{若}n > 0\text{且为偶数}) \end{cases}$$

按照这一新的表述和理解，可按二进制展开n之后的各比特位，通过反复的平方运算和加倍运算得到power2(n)。比如：

$$2^1 = 2^{001_{(2)}} = (2^2^2)^0 \times (2^2)^0 \times 2^1 = (((1 \times 2^0)^2 \times 2^0)^2 \times 2^1)$$
$$2^2 = 2^{010_{(2)}} = (2^2^2)^0 \times (2^2)^1 \times 2^0 = (((1 \times 2^0)^2 \times 2^1)^2 \times 2^0)$$
$$2^3 = 2^{011_{(2)}} = (2^2^2)^0 \times (2^2)^1 \times 2^1 = (((1 \times 2^0)^2 \times 2^1)^2 \times 2^1)$$

$$2\text{^}4 = 2\text{^}100_{(2)} = (2\text{^}2\text{^}2)^1 \times (2\text{^}2)^0 \times 2^0 = (((1 \times 2^1)\text{^}2 \times 2^0)\text{^}2 \times 2^0)$$
$$2\text{^}5 = 2\text{^}101_{(2)} = (2\text{^}2\text{^}2)^1 \times (2\text{^}2)^0 \times 2^1 = (((1 \times 2^1)\text{^}2 \times 2^0)\text{^}2 \times 2^1)$$
$$2\text{^}6 = 2\text{^}110_{(2)} = (2\text{^}2\text{^}2)^1 \times (2\text{^}2)^1 \times 2^0 = (((1 \times 2^1)\text{^}2 \times 2^1)\text{^}2 \times 2^0)$$
$$2\text{^}7 = 2\text{^}111_{(2)} = (2\text{^}2\text{^}2)^1 \times (2\text{^}2)^1 \times 2^1 = (((1 \times 2^1)\text{^}2 \times 2^1)\text{^}2 \times 2^1)$$
$$\ldots$$

一般地，若n的二进制展开式为$b_1b_2b_3\ldots b_k$，则有

$$2\text{^}n = (\ldots(((1 \times 2^{b1})\text{^}2 \times 2^{b2})\text{^}2 \times 2^{b3})\text{^}2 \ldots \times 2^{bk})$$

若n_{k-1}和n_k的二进制展开式分别为$b_1b_2\ldots b_{k-1}$和$b_1b_2\ldots b_{k-1}b_k$，则有

$$2\text{^}n_k = (2\text{^}n_{k-1})\text{^}2 \times 2^{bk}$$

由此可以归纳得出如下递推式：

$$\text{power2}(n_k) = \begin{cases} \text{power2}(n_{k-1})\text{^}2 \times 2 & (\text{若}b_k = 1) \\ \text{power2}(n_{k-1})\text{^}2 & (\text{若}b_k = 0) \end{cases}$$

基于这一递推式，即可如代码1.8所示，实现幂函数的多向递归版本power2()：

```
1 inline __int64 sqr ( __int64 a ) { return a * a; }
2 __int64 power2 ( int n ) { //幂函数2^n算法（优化递归版），n >= 0
3    if ( 0 == n ) return 1; //递归基；否则，视n的奇偶分别递归
4    return ( n & 1 ) ? sqr ( power2 ( n >> 1 ) ) << 1 : sqr ( power2 ( n >> 1 ) );
5 } //O(logn) = O(r)，r为输入指数n的比特位数
```

代码1.8 优化的幂函数算法（线性递归版）

针对输入参数n为奇数或偶数的两种可能，这里分别设有不同的递归方向。尽管如此，每个递归实例都只能沿其中的一个方向深入到下层递归，整个算法的递归跟踪分析图的拓扑结构仍然与图1.6类似，故依然属于线性递归。可以证明（习题[1-31]），该算法的时间复杂度为：

$$\mathcal{O}(\text{logn}) \times \mathcal{O}(1) = \mathcal{O}(r)$$

与此前代码1.4中蛮力版本的$\mathcal{O}(n) = \mathcal{O}(2^n)$相比，计算效率得到了极大提高。

1.4.4 递归消除

由上可见，按照递归的思想可使我们得以从宏观上理解和把握应用问题的实质，深入挖掘和洞悉算法过程的主要矛盾和一般性模式，并最终设计和编写出简洁优美且精确紧凑的算法。然而，递归模式并非十全十美，其众多优点的背后也隐含着某些代价。

■ **空间成本**

首先，从递归跟踪分析的角度不难看出，递归算法所消耗的空间量主要取决于递归深度（习题[1-17]），故较之同一算法的迭代版，递归版往往需耗费更多空间，并进而影响实际的运行速度。另外，就操作系统而言，为实现递归调用需要花费大量额外的时间以创建、维护和销毁各递归实例，这些也会令计算的负担雪上加霜。有鉴于此，在对运行速度要求极高、存储空间需精打细算的场合，往往应将递归算法改写成等价的非递归版本。

一般的转换思路，无非是利用栈结构（第4章）模拟操作系统的工作过程。这类的通用方法已超出本书的范围，以下仅针对一种简单而常见的情况，略作介绍。

■ 尾递归及其消除

在线性递归算法中，若递归调用在递归实例中恰好以最后一步操作的形式出现，则称作尾递归（tail recursion）。比如代码1.7中reverse(A, lo, hi)算法的最后一步操作，是对去除了首、末元素之后总长缩减两个单元的子数组进行递归倒置，即属于典型的尾递归。实际上，属于尾递归形式的算法，均可以简捷地转换为等效的迭代版本。

仍以代码1.7中reverse(A, lo, hi)算法为例。如代码1.9所示，首先在起始位置插入一个跳转标志next，然后将尾递归语句调用替换为一条指向next标志的跳转语句。

```
1 void reverse ( int* A, int lo, int hi ) { //数组倒置（直接改造而得的迭代版）
2 next: //算法起始位置添加跳转标志
3    if ( lo < hi ) {
4       swap ( A[lo], A[hi] ); //交换A[lo]和A[hi]
5       lo++; hi--; //收缩待倒置区间
6       goto next; //跳转至算法体的起始位置，迭代地倒置A(lo, hi)
7    } //else隐含了迭代的终止
8 } //O(hi - lo + 1)
```

代码1.9 由递归版改造而得的数组倒置算法（迭代版）

新的迭代版与原递归版功能等效，但其中使用的goto语句有悖于结构化程序设计的原则。这一语句虽仍不得不被C++等高级语言保留，但最好还是尽力回避。为此可如代码1.10所示，将next标志与if判断综合考查，并代之以一条逻辑条件等价的while语句。

```
1 void reverse ( int* A, int lo, int hi ) { //数组倒置（规范整理之后的迭代版）
2    while ( lo < hi ) //用while替换跳转标志和if，完全等效
3       swap ( A[lo++], A[hi--] ); //交换A[lo]和A[hi]，收缩待倒置区间
4 } //O(hi - lo + 1)
```

代码1.10 进一步调整代码1.9的结构，消除goto语句

请注意，尾递归的判断应依据对算法实际执行过程的分析，而不仅仅是算法外在的语法形式。比如，递归语句出现在代码体的最后一行，并不见得就是尾递归；严格地说，只有当该算法（除平凡递归基外）任一实例都终止于这一递归调用时，才属于尾递归。以代码1.5中线性递归版sum()算法为例，尽管从表面看似乎最后一行是递归调用，但实际上却并非尾递归——实质的最后一次操作是加法运算。有趣的是，此类算法的非递归化转换方法仍与尾递归如出一辙，相信读者不难将其改写为类似于代码1.3中sumI()算法的迭代版本。

1.4.5 二分递归

■ 分而治之

面对输入规模庞大的应用问题，每每感慨于头绪纷杂而无从下手的你，不妨从先哲孙子的名言中获取灵感——“凡治众如治寡，分数是也”。是的，解决此类问题的有效方法之一，就是将其分解为若干规模更小的子问题，再通过递归机制分别求解。这种分解持续进行，直到子问题规模缩减至平凡情况。这也就是所谓的分而治之（divide-and-conquer）策略。

与减而治之策略一样，这里也要求对原问题重新表述，以保证子问题与原问题在接口形式上的一致。既然每一递归实例都可能做多次递归，故称作“多路递归”（multi-way recursion）。通常都是将原问题一分为二，故称作“二分递归”（binary recursion）。需强调的是，无论是分解为两个还是更大常数个子问题，对算法总体的渐进复杂度并无实质影响。

■ 数组求和

以下就采用分而治之的策略，按照二分递归的模式再次解决数组求和问题。新算法的思路是：以居中的元素为界将数组一分为二；递归地对子数组分别求和；最后，子数组之和相加即为原数组的总和。具体过程可描述如代码1.11，算法入口的调用形式为sum(A, 0, n)。

```
1 int sum ( int A[], int lo, int hi ) { //数组求和算法（二分递归版，入口为sum(A, 0, n - 1)）
2    if ( lo == hi ) //如遇递归基（区间长度已降至1），则
3       return A[lo]; //直接返回该元素
4    else { //否则（一般情况下lo < hi），则
5       int mi = ( lo + hi ) >> 1; //以居中单元为界，将原区间一分为二
6       return sum ( A, lo, mi ) + sum ( A, mi + 1, hi ); //递归对各子数组求和，然后合计
7    }
8 } //O(hi - lo + 1)，线性正比于区间的长度
```

代码1.11 通过二分递归计算数组元素之和

该算法的正确性无需解释。为分析其复杂度，不妨只考查n = 2^m形式的长度。

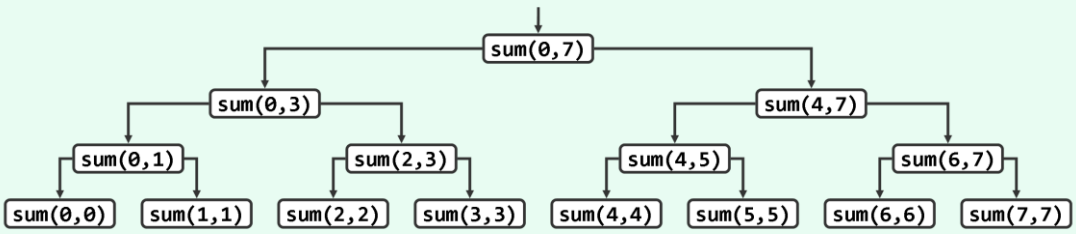

图1.7 对sum(A, 0, 7)的递归跟踪分析

图1.7针对n = 8的情况给出了sum(A, 0, 7)执行过程的递归跟踪。其中各方框都标注有对应的lo和hi值，即子数组区间的起、止单元。可见，按照调用的关系及次序，该方法的所有实例构成一个层次结构（即第5章将介绍的二叉树）。沿着这个层次结构每下降一层，每个递归实例sum(lo, hi)都分裂为一对更小的实例sum(lo, mi)和sum(mi + 1, hi)——准确地说，每经过一次递归调用，子问题对应的数组区间长度hi - lo + 1都将减半。

算法启动后经连续$m = \log_2 n$次递归调用，数组区间的长度从最初的n首次缩减至1，并到达第一个递归基。实际上，刚到达任一递归基时，已执行的递归调用总是比递归返回多$m = \log_2 n$次。更一般地，到达区间长度为2^k的任一递归实例之前，已执行的递归调用总是比递归返回多m-k次。因此，递归深度（即任一时刻的活跃递归实例的总数）不会超过m + 1。鉴于每个递归实例仅需常数空间，故除数组本身所占的空间，该算法只需要$O(m + 1) = O(\log n)$的附加空间。我们还记得，代码1.5 中线性递归版sum()算法共需$O(n)$的附加空间，就这一点而言，新的二分递归版sum()算法有很大改进。

与线性递归版sum()算法一样，此处每一递归实例中的非递归计算都只需要常数时间。递归

实例共计2n - 1个，故新算法的运行时间为$\mathcal{O}(2n - 1) = \mathcal{O}(n)$，与线性递归版相同。

此处每个递归实例可向下深入递归两次，故属于多路递归中的二分递归。二分递归与此前介绍的线性递归有很大区别。比如，在线性递归中整个计算过程仅出现一次递归基，而在二分递归过程中递归基的出现相当频繁，总体而言有超过半数的递归实例都是递归基。

■ 效率

当然，并非所有问题都适宜于采用分治策略。实际上除了递归，此类算法的计算消耗主要来自两个方面。首先是子问题划分，即把原问题分解为形式相同、规模更小的多个子问题，比如代码1.11中sum()算法将待求和数组分为前、后两段。其次是子解答合并，即由递归所得子问题的解，得到原问题的整体解，比如由子数组之和累加得到整个数组之和。

为使分治策略真正有效，不仅必须保证以上两方面的计算都能高效地实现，还必须保证子问题之间相互独立——各子问题可独立求解，而无需借助其它子问题的原始数据或中间结果。否则，或者子问题之间必须传递数据，或者子问题之间需要相互调用，无论如何都会导致时间和空间复杂度的无谓增加。以下就以Fibonacci数列的计算为例说明这一点。

■ Fibonacci数：二分递归

考查Fibonacci数列第n项fib(n)的计算问题，该数列递归形式的定义如下：

$$fib(n) = \begin{cases} n & (若n \le 1) \\ fib(n-1) + fib(n-2) & (若n \ge 2) \end{cases}$$

据此定义，可直接导出如代码1.12所示的二分递归版fib()算法：

```
1 __int64 fib ( int n ) { //计算Fibonacci数列的第n项（二分递归版）：O(2^n)
2    return ( 2 > n ) ?
3           ( __int64 ) n //若到达递归基，直接取值
4           : fib ( n - 1 ) + fib ( n - 2 ); //否则，递归计算前两项，其和即为正解
5 }
```

代码1.12 通过二分递归计算Fibonacci数

基于Fibonacci数列原始定义的这一实现，不仅正确性一目了然，而且简洁自然。然而不幸的是，在这种场合采用二分递归策略的效率极其低下。实际上，该算法需要运行$\mathcal{O}(2^n)$时间才能计算出第n个Fibonacci数。这一指数复杂度的算法，在实际环境中毫无价值。

为确切地界定该算法的复杂度，不妨将计算fib(n)所需的时间记作T(n)。按该算法的思路，为计算出fib(n)，先花费T(n - 1)时间计算fib(n - 1)，再花费T(n - 2)时间计算fib(n - 2)，最后花费一个单位的时间将它们累加起来。由此，可得T(n)的递推式如下：

$$T(n) = \begin{cases} 1 & (若n \le 1) \\ T(n - 1) + T(n - 2) + 1 & (否则) \end{cases}$$

若令S(n) = [T(n) + 1]/2，则有：

$$S(n) = \begin{cases} 1 & (若n \le 1) \\ S(n - 1) + S(n - 2) & (否则) \end{cases}$$

我们发现，S(n)的递推形式与fib(n)完全一致，只是起始项不同：

$$S(0) = (T(0) + 1) / 2 = 1 = fib(1)$$

$$S(1) = (T(1) + 1) / 2 = 1 = fib(2)$$

亦即，S(n)整体上相对于fib(n)提前了一个单元。由此可知：

$$S(n) = fib(n + 1) = (\Phi^{n+1} - \hat{\Phi}^{n+1})/\sqrt{5},\ \Phi = (1 + \sqrt{5})/2,\ \hat{\Phi} = (1 - \sqrt{5})/2$$

$$T(n) = 2 \cdot S(n) - 1 = 2 \cdot fib(n + 1) - 1 = \mathcal{O}(\Phi^{n+1}) = \mathcal{O}(2^n)$$

这一版本fib()算法的时间复杂度高达指数量级，究其原因在于，计算过程中所出现的递归实例的重复度极高——只需画出递归跟踪分析图的前几层，即不难验证这一点。若需更为精确的界定，可以借助递推方程（习题[1-19]），将得到相同的结论。

■ 优化策略

为消除递归算法中重复的递归实例，一种自然而然的思路和技巧，可以概括为：

借助一定量的辅助空间，在各子问题求解之后，及时记录下其对应的解答

比如，可以从原问题出发自顶而下，每当遇到一个子问题，都首先查验它是否已经计算过，以期通过直接调阅记录获得解答，从而避免重新计算。也可以从递归基出发，自底而上递推地得出各子问题的解，直至最终原问题的解。前者即所谓的制表（tabulation）或记忆（memoization）策略，后者即所谓的动态规划（dynamic programming）策略。

■ Fibonacci数：线性递归

为应用上述制表的策略，首先需从改造Fibonacci数的递归定义入手。

反观代码1.12，原fib()算法之所以采用二分递归模式，完全是因为受到该问题原始定义的表面特征——fib(n)由fib(n - 1)和fib(n - 2)共同决定——的误导。然而不难看出，子问题fib(n - 1)和fib(n - 2)实际上并非彼此独立。比如，只要转而采用定义如下的递归函数，计算一对相邻的Fibonacci数：

(fib(k-1), fib(k))

即可如代码1.13所示，得到效率更高（习题[1-21]）的线性递归版fib()算法。

```
1 __int64 fib ( int n, __int64& prev ) { //计算Fibonacci数列第n项（线性递归版）：入口形式fib(n, prev)
2    if ( 0 == n ) //若到达递归基，则
3       { prev = 1; return 0; } //直接取值：fib(-1) = 1, fib(0) = 0
4    else { //否则
5       __int64 prevPrev; prev = fib ( n - 1, prevPrev ); //递归计算前两项
6       return prevPrev + prev; //其和即为正解
7    }
8 } //用辅助变量记录前一项，返回数列的当前项，O(n)
```

代码1.13 通过线性递归计算Fibonacci数

请注意，原二分递归版本中对应于fib(n - 2)的另一次递归，在这里被省略掉了。其对应的解答，可借助形式参数的机制，通过变量prevPrev“调阅”此前的记录直接获得。

该算法呈线性递归模式，递归的深度线性正比于输入n，前后共计仅出现$\mathcal{O}(n)$个递归实例，累计耗时不超过$\mathcal{O}(n)$。遗憾的是，该算法共需使用$\mathcal{O}(n)$规模的附加空间。如何进一步改进呢？

■ Fibonacci数：迭代

反观以上线性递归版fib()算法可见，其中所记录的每一个子问题的解答，只会用到一次。在该算法抵达递归基之后的逐层返回过程中，每向上返回一层，以下各层的解答均不必继续保留。

若将以上逐层返回的过程，等效地视作从递归基出发，按规模自小而大求解各子问题的过程，即可采用动态规划的策略，将以上算法进一步改写为如代码1.14所示的迭代版。

```
1 __int64 fibI ( int n ) { //计算Fibonacci数列的第n项（迭代版）：O(n)
2    __int64 f = 0, g = 1; //初始化：fib(0) = 0, fib(1) = 1
3    while ( 0 < n-- ) { g += f; f = g - f; } //依据原始定义，通过n次加法和减法计算fib(n)
4    return f; //返回
5 }
```

代码1.14 基于动态规划策略计算Fibonacci数

这里仅使用了两个中间变量f和g，记录当前的一对相邻Fibonacci数。整个算法仅需线性步的迭代，时间复杂度为$\mathcal{O}(n)$。更重要的是，该版本仅需常数规模的附加空间，空间效率也有了极大提高。

§1.5 抽象数据类型

各种数据结构都可看作是由若干数据项组成的集合，同时对数据项定义一组标准的操作。现代数据结构普遍遵从“信息隐藏”的理念，通过统一接口和内部封装，分层次从整体上加以设计、实现与使用。

所谓封装，就是将数据项与相关的操作结合为一个整体，并将其从外部的可见性划分为若干级别，从而将数据结构的外部特性与其内部实现相分离，提供一致且标准的对外接口，隐藏内部的实现细节。于是，数据集合及其对应的操作可超脱于具体的程序设计语言、具体的实现方式，即构成所谓的抽象数据类型（abstract data type, ADT）。抽象数据类型的理论催生了现代面向对象的程序设计语言，而支持封装也是此类语言的基本特征。

本书将尽可能遵循抽象数据类型的规范来设计、实现并分析各种数据结构。具体地，将从各数据结构的对外功能接口（interface）出发，以C++语言为例逐层讲解其内部具体实现（implementation）的原理、方法与技巧，并就不同实现方式的效率及适用范围进行分析与比较。为体现数据结构的通用性，也将普遍采用模板类的描述模式。