第2章 程 序 性 能

以下是本章中所介绍的有关程序性能分析与测量的概念：

• 确定一个程序对内存及时间的需求。

• 使用操作数和执行步数来测量一个程序的时间需求。

• 采用渐进符号描述复杂性，如**O**、Ω、**Θ**、***o***。

• 利用计时函数测量一个程序的实际运行时间。

除了上述概念以外，本章还给出了许多应用代码，在后续章节中将可以看到，这些代码有很多用处。这些应用包括：

• 在一个数组中搜索具有指定特征的元素。本章中所使用的方法是顺序搜索和折半搜索。

• 对数组元素进行排序。本章给出了计数排序、选择排序、冒泡排序及插入排序的实现代码。

• 采用Horner 法则计算一个多项式。

• 执行矩阵运算，如矩阵加、矩阵转置和矩阵乘。

2.1 引言

所谓程序性能（program performance），是指运行一个程序所需要的内存大小和时间。可以采用两种方法来确定一个程序的性能，一个是分析的方法，一个是实验的方法。在进行性能分析（performance analysis）时，采用分析的方法，而在进行性能测量（performance measurement）时，借助于实验的方法。

程序的空间复杂性（space complexity）是指运行完一个程序所需要的内存大小。对一个程序的空间复杂性感兴趣的主要原因如下：

• 如果程序将要运行在一个多用户计算机系统中，可能需要指明分配给该程序的内存大小。

• 对任何一个计算机系统，想提前知道是否有足够可用的内存来运行该程序。

• 一个问题可能有若干个内存需求各不相同的解决方案。比如，对于你的计算机来说，某个C++编译器仅需要1MB的空间，而另一个C++编译器可能需要4MB的空间。如果你的计算机中内存少于4MB，你只能选择1MB的编译器。如果较小编译器的性能比得上较大的编译器，即使用户的计算机中有额外的内存，也宁愿使用较小的编译器。

• 可以利用空间复杂性来估算一个程序所能解决的问题的最大规模。例如，有一个电路模拟程序，用它模拟一个有 c个元件、w个连线的电路需要280K+10*（c+w）字节的内存。如果可利用的内存总量为640K字节，那么最大可以模拟$c+w \leq 36$K的电路。

程序的时间复杂性（time complexity）是指运行完该程序所需要的时间。对一个程序的时间复杂性感兴趣的主要原因如下：

• 有些计算机需要用户提供程序运行时间的上限，一旦达到这个上限，程序将被强制结束。一种简易的办法是简单地指定时间上限为几千年。然而这种办法可能会造成严重的财政问题，因为如果由于数据问题导致你的程序进入一个死循环，你可能需要为你所使用的机时付出巨额资金。因此我们希望能提供一个稍大于所期望运行时间的时间上限。

• 正在开发的程序可能需要提供一个满意的实时响应。例如，所有交互式程序都必须提供实时响应。一个需要1分钟才能把光标上移一页或下移一页的文本编辑器不可能被众多的用户接受；一个电子表格程序需要花费几分钟才能对一个表单中的单元进行重新计值，那么只有非常耐心的用户才会乐意使用它；如果一个数据库管理系统在对一个关系进行排序时，用户可以有时间去喝两杯咖啡，那么它也很难被用户接受。为交互式应用所设计的程序必须提供满意的实时响应。根据程序或程序模块的时间复杂性，可以决定其响应时间是否可以接受，如果不能接受，要么重新设计正在使用的算法，要么为用户提供一台更快的计算机。

• 如果有多种可选的方案来解决一个问题，那么具体决定采用哪一个主要基于这些方案之间的性能差异。对于各种解决方案的时间及空间复杂性将采用加权的方式进行评价。

练习

1. 给出两种以上的原因说明为什么程序分析员对程序的空间复杂性感兴趣？
2. 给出两种以上的原因说明为什么程序分析员对程序的时间复杂性感兴趣？

2.2 空间复杂性

2.2.1 空间复杂性的组成

程序所需要的空间主要由以下部分构成：

• 指令空间（instruction space） 指令空间是指用来存储经过编译之后的程序指令所需的空间。

• 数据空间（data space） 数据空间是指用来存储所有常量和所有变量值所需的空间。数据空间由两个部分构成：

1) 存储常量（见程序1-1至1-9中的数0、1和4）和简单变量（见程序1-1至1-6中的a、b和c）所需要的空间。

2) 存储复合变量（见程序1-8和1-9中的数组a）所需要的空间。这一类空间包括数据结构所需的空间及动态分配的空间。

• 环境栈空间（environment stack space） 环境栈用来保存函数调用返回时恢复运行所需要的信息。例如，如果函数fun1调用了函数fun2，那么至少必须保存fun2结束时fun1将要继续执行的指令的地址。

1. 指令空间

程序所需要的指令空间的数量取决于如下因素：

• 把程序编译成机器代码的编译器。

• 编译时实际采用的编译器选项。

• 目标计算机。

在决定最终代码需要多少空间的时候，编译器是一个最重要的因素。图 2-1给出了用来计算表达式a+b+b*c+(a+b-c)/(a+b)+4的三段可能的代码，它们都执行完全相同的算术操作（如，每个操作符都有相同的操作数），但每段代码所需要的空间都不一样。所使用的编译器将确定产生哪一种代码。

LOAD	a	LOAD	a	LOAD	a
ADD	b	ADD	b	ADD	b
STORE	t1	STORE	t1	STORE	t1
LOAD	b	SUB	c	SUB	c
MULT	c	DIV	t1	DIV	t1
STORE	t2	STORE	t2	STORE	t2
LOAD	t1	LOAD	b	LOAD	b
ADD	t2	MUL	c	MUL	c
STORE	t3	STORE	t3	ADD	t2
LOAD	a	LOAD	t1	ADD	t1
ADD	b	ADD	t3	ADD	4
SUB	c	ADD	t2		
STORE	t4	ADD	4		
LOAD	a				
ADD	b				
STORE	t5				
LOAD	t4				
DIV	t5				
STORE	t6				
LOAD	t3				
ADD	t6				
ADD	4				
a)		b)		c)	

图2-1 三段等价的代码

即使采用相同的编译器，所产生程序代码的大小也可能不一样。例如，一个编译器可能为用户提供优化选项，如代码优化以及执行时间优化等。比如，在图 2-1中，在非优化模式下，编译器可以产生图2-1b 的代码。在优化模式下，编译器可以利用知识 a+b+b*c=b*c+(a+b)来产生图2-1c 中更短、更高效的代码。使用优化模式通常会增加程序编译所需要的时间。

从图2-1的例子中可以看到，一个程序还可能需要其他额外的空间，即诸如临时变量 t1, t2, ..., t6 所占用的空间。

另外一种可以显著减少程序空间的编译器选项就是覆盖选项，在覆盖模式下，空间仅分配给当前正在执行的程序模块。在调用一个新的模块时，需要从磁盘或其他设备中读取，新模块的代码将覆盖原模块的代码。所以程序空间就等价于最大的模块所需要的空间（而不是所有模块之和）。

目标计算机的配置也会影响代码的规模。如果计算机具有浮点处理硬件，那么每个浮点操作可以转换成一条机器指令。如果没有安装浮点处理硬件，必须生成仿真的浮点计算代码。

2. 数据空间

对于简单变量和常量来说，所需要的空间取决于所使用的计算机和编译器以及变量与常量的数目。之所以如此是因为我们通常关心所需内存的字节数。由于每字节所占用的位数依赖于具体的机器环境，因此每个变量所需要的空间也会有所不同。此外，存储 2^{100} 也将比存储2^3需要更多的位数。

图2-2中列出了Borland C++中每种简单变量所占用的空间。对于一个结构变量，可以把它的每个成员所占用的空间累加起来即可得到该变量所需要的内存。类似地，可以得到一个数组变量所需要的空间，方法是用数组的大小乘以单个数组元素所需要的空间。

类 型	占用字节数	范 围
char	1	-128~127
unsigned char	1	0~255
short	2	-32 768~32 767
int	2	-32 768~32 767
unsigned int	2	0~65 535
long	4	-2^{31}~2^{31}-1
unsigned long	4	0~2^{32}-1
float	4	±3.4E±38
double	8	±1.7E±308
long double	10	3. 4E-4932~1.1E+4932
pointer	2	(near, _cs, _ds, _es, _ss指针)
pointer	4	(far, huge指针)

注意：在32位Borland C++程序中， int类型的长度为4

图2-2 Borland C++中每种简单变量所占用的空间（摘自 Borland C++ Programmer's Guide，Borland 公司，加州Scotts Valley, 1996）

考察如下的数组定义：

```
double a[100];
int  maze[rows][cols];
```

数组a 需要的空间为100个double类型元素所占用的空间，若每个元素占用 8个字节，则分配给该数组的空间总量为800字节。数组maze有rows*cols个int类型的元素，它所占用的总空间为2*rows*cols字节。

3. 环境栈

在刚开始从事性能分析时，分析员通常会忽略环境栈所需要的空间，因为他们不理解函数是如何被调用的以及在函数调用结束时会发生什么。每当一个函数被调用时，下面的数据将被保存在环境栈中：

- 返回地址。
- 函数被调用时所有局部变量的值以及传值形式参数的值（仅对于递归函数而言）。
- 所有引用参数及常量引用参数的定义。

每当递归函数Rsum(见程序1-9)被调用时，不管该调用是来自外部或第 4行，a的当前赋值、n的值以及程序运行结束时的返回地址都被存储在环境栈之中。

值得注意的是，有些编译器在保留局部变量的值、传值形式参数的值以及引用参数和常量引用参数的定义时，对于递归函数和非递归函数一视同仁，而有些编译器则仅为递归函数保存上述内容。所以实际使用的编译器将影响环境栈所需要的空间。

4. 小结

程序所需要的空间取决于多种因素，有些因素在构思或编写程序时是未知的（如将要使用

的计算机及编译器），不过即使这些因素已经完全确定，我们也无法精确地分析一个程序所需要的空间。

然而，我们可以确定程序中某些部分的空间需求，这些部分依赖于所解决实例的特征。一般来说，这些特征包含决定问题规模的那些因素（如，输入和输出的数量或相关数的大小）。例如，对于一个对n 个元素进行排序的程序，可以确定该程序所需要的空间为 n 的函数；对于一个累加两个$n\times n$ 矩阵的程序，可以使用n 作为其实例特征；而对于累加两个$m\times n$ 矩阵的程序，可以使用m 和n 作为实例特征。

指令空间的大小对于所解决的特定问题不够敏感。常量及简单变量所需要的数据空间也独立于所解决的问题，除非相关数的大小对于所选定的数据类型来说实在太大，这时，要么改变数据类型，要么使用多精度算法重写该程序，然后再对新程序进行分析。

复合变量及动态分配所需要的空间同样独立于问题的规模。而环境栈通常独立于实例的特征，除非正在使用递归函数。在使用递归函数时，实例特征通常（但不总是）会影响环境栈所需要的空间数量。

递归函数所需要的栈空间通常称之为递归栈空间（recursion stack space）。对于每个递归函数而言，该空间主要依赖于局部变量及形式参数所需要的空间。除此以外，该空间还依赖于递归的深度（即嵌套递归调用的最大层次）。在程序1-9中嵌套递归调用一直进行到 n=0，这时，嵌套调用的层次关系如图2-3所示。该程序的最大递归深度为n+1。

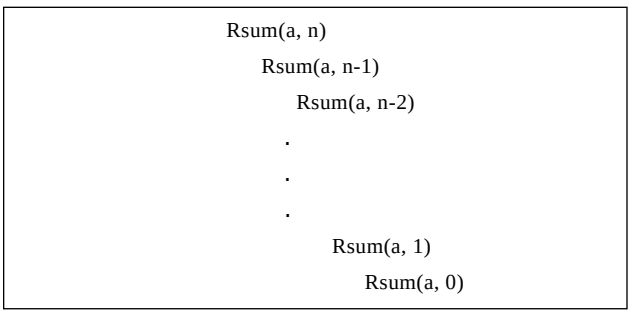

图2-3 程序1-9的嵌套调用层次

至此，可以把一个程序所需要的空间分成两部分：

• 固定部分，它独立于实例的特征。一般来说，这一部分包含指令空间（即代码空间）、简单变量及定长复合变量所占用空间、常量所占用空间等等。

• 可变部分，它由以下部分构成：复合变量所需的空间（这些变量的大小依赖于所解决的具体问题），动态分配的空间（这种空间一般都依赖于实例的特征），以及递归栈所需的空间（该空间也依赖于实例的特征）。

任意程序P 所需要的空间$S(P)$ 可以表示为：

$$S(P) = c + S_p \text{（实例特征）}$$

其中c 是一个常量，表示固定部分所需要的空间，S_p 表示可变部分所需要的空间。一个精确的分析还应当包括在编译期间所产生的临时变量所需要的空间（如图 2-1所示），这种空间是与编译器直接相关的，除依赖于递归函数外，它还依赖于实例的特征。本书将忽略这种空间。

在分析程序的空间复杂性时，我们将把注意力集中在估算 S_p（实例特征）上。对于任意给

定的问题，首先需要确定实例的特征以便于估算空间需求。实例特征的选择是一个很具体的问题，我们将求助于介绍各种可能性的实际例子。一般来说，我们的选择受到相关数的数量以及程序输入和输出的规模的限制。有时还会使用更复杂的估算数据，这些数据来自于数据项之间的相互关系。

2.2.2 举例

例2-1 考察程序1-4。在估算S_p之前，必须选择分析所使用的实例特征。两种可能性是：(1)数据类型T；(2) a, b 和c 的大小。假定使用T作为实例特征。由于a, b和c是引用参数，所以在函数中不需要为它们的值分配空间，但是必须保存指向这些参数的指针。如果每个指针需要 2个字节，那么共需要6个字节的指针空间。因此函数所需要的总空间是一个常量，S_{Abc}(实例特征)=0。如果函数Abc 的参数是传值参数，那么每个参数需要分配大小为sizeof (T) 的空间。在本例中，a, b 和c 所需要的空间为 3*sizeof (T)。所需要的其他空间都独立于 T。因此S_{Abc} (实例特征) =3*sizeof (T)。如果使用a, b 和c 的大小作为实例特征，则不管使用引用参数还是使用传值参数，都有S_{Abc} (实例特征)=0。注意在传值参数的情形下，分配给每个 a,b和c的空间均为sizeof(T)，而不考虑存储在这些变量中的实际值是多大。例如，如果T是double类型，那么每个字节将被分配8个字节的空间。

例2-2 **[顺序搜索]** 程序2-1从左至右检查数组a[0:n-1]中的元素，以查找与x相等的那些元素。如果找到一个元素与x 相等，则函数返回x 第一次出现所在的位置。如果在数组中没有找到这样的元素，函数返回-1。

程序2-1 顺序搜索

```
template<class T>
int SequentialSearch(T a[], const T& x, int n)
{// 在未排序的数组a[0:n-1]中搜索 x
 // 如果找到，则返回所在位置，否则返回-1
   int i;
   for (i = 0; i < n && a[i] != x; i++);
   if (i == n) return -1;
   return i;
}
```

我们希望采用实例特征n 来估算该函数的空间复杂性。假定T 为int 类型，则数组a 中的每个元素需要2个字节，实参x需要2个字节，传值形式参数n 需要2个字节，局部变量i 需要2个字节，每个整型常量0和-1也分别需要2个字节。因此，所需要的总的数据空间为 12字节。因为该空间独立于n，所以$S_{顺序搜索}$（n）=0。

注意数组a 必须足够大以容纳所查找的n 个元素。不过，该数组所需要的空间已在定义实际参数（对应于a）的函数中分配，所以，不需要把该数组所需要的空间加到函数SequentialSearch所需要的空间上去。

例2-3 考察函数Sum (见程序1-8)。假定我们有兴趣把该函数所需要的空间看成欲累加元素的总数的函数。在该函数中， a, n, i 和tsum 需要分配空间，所以程序所需要的空间与n 无关，因此有S_{Sum} (n) = 0。

例2-4 考察函数Rsum (见程序1-9)。如上例一样，假定实例特征为 n。递归栈空间包括形式参数a和n以及返回地址的空间。对于a，需要保留一个指针，而对于n 则需要保留一个int 类型的值。如果假定指针为near 指针，则该指针需要2个字节的存储空间。如果同时假定返回地址也占用2个字节，那么根据int 类型需要2个字节的常识，可以确定每一次调用 Rsum 需要6个字节的栈空间。由于递归的深度为 n+1，所以需要 6(n+1)字节的递归栈空间，因而 $S_{Rsum}(n)=6(n+1)$。

程序1-8所需要的空间比程序1-9所需要空间要小。

例2-5 **[阶乘运算]** 通过分析程序1-7中计算阶乘的函数可知，该程序的空间复杂性是n的函数而不是输入（只有一个）或输出（也只有一个）个数的函数。递归深度为 max {n, 1}。每次调用函数Factorial 时，递归栈需要保留返回地址（2个字节）和n 的值（2个字节）。此外没有其他依赖于n 的空间，所以$S_{Factorial}(n) = 4*\max\{n, 1\}$。

例2-6 **[排列方式]** 程序1-10输出一组元素的所有排列方式。对于初始调用 Perm (list, 0, n - 1)，递归的深度为n。由于每次调用需要10个字节的递归栈空间（每个返回地址、 list、k、m以及i各需要2个字节），所以需要10n 字节的递归栈空间，$S_{Perm}(n) = 10n$。

练习

3. 试采用两种C++编译器编译同一个C++程序，所得代码的长度相同吗？
4. 给出可能影响程序空间复杂性的其他因素。
5. 使用图2-2所提供的数据来计算如下数组所需要的字节数：

1) int matrix[10][100]
2) double x[100][5][20]
3) long double y[3]
4) float z[10][10][10][5]
5) short z[2][3][4]
6) long double b[3][3][3][3]

6. 程序2-2给出了一个在数组a[0:n-1]中查找元素x的递归函数。如果找到x,则函数返回x在a中的位置，否则返回 - 1。试计算$S_p(n)$。

程序2-2 执行顺序搜索的递归函数

```
template <class T>
int SequentialSearch(T a[], const T& x, int n)
{//在未排序的数组a[0:n - 1]中查找x
 //如果找到则返回所在位置，否则返回 - 1
 if (n <1) return  -1;
 if (a[n-1] == x) return n - 1;
 return SequentialSearch(a,x,n - 1);
}
```

7. 编写一个非递归函数计算$n!$（例1-1），并比较该函数的空间复杂性与程序1-7中递归函数的空间复杂性。

2.3 时间复杂性

2.3.1 时间复杂性的组成

影响一个程序空间复杂性的因素也都能影响程序的时间复杂性。一个程序在一台每秒钟能执行10^9条指令的机器上运行要比在每秒仅能执行10^6条指令的机器上快得多。图2-1c 中的代码要比图2-1a 中的代码运行时间更少。较小的问题所需要的运行时间通常要比较大的问题需要的时间少。

一个程序P所占用的时间$T(P)$=编译时间+运行时间。编译时间与实例的特征无关。另外，可以假定一个编译过的程序可以运行若干次而不需要重新编译。因此我们将主要关注程序的运行时间。运行时间通常用“t_p(实例特征)”来表示。

由于在构思一个程序时，影响t_p的许多因素还是未知的，所以我们有理由仅仅对t_p进行估算。如果我们了解所用编译器的特征，就可以确定代码P进行加、减、乘、除、比较、读、写等所需要的时间，从而可以得到一个计算t_p的公式。令n代表实例的特征，可以得到如下形式的表达式：

$$t_p(n) = c_a ADD(n) + c_s SUB(n) + c_m MUL(n) + c_d DIV(n) + \cdots$$

其中c_a, c_s, c_m和c_d分别表示一个加、减、乘和除操作所需要的时间，函数*ADD*、*SUB*、*MUL*和*DIV*分别表示代码P中所使用的加、减、乘和除操作的次数。

存在这样一个事实：一个算术操作所需要的时间取决于操作数的类型（ int, float, double 等），这个事实增加了获得一个精确的计算公式的烦琐程度。所以必须按照数据类型对操作进行分类。

有两个更可行的方法可用来估算运行时间： 1）找出一个或多个关键操作，确定这些关键操作所需要的执行时间；2）确定程序总的执行步数。

2.3.2 操作计数

估算一个程序或函数的时间复杂性的一种方式就是首先选择一种或多种操作 (如加、乘和比较等)，然后确定这种(些)操作分别执行了多少次。这种方法是否成功取决于识别关键操作的能力，这些关键操作对时间复杂性的影响最大。下面给出的几个例子都采用了这种方法。

例2-7 [最大元素] 程序1-31返回数组a[0:n-1]中最大元素的位置。我们可以根据数组元素之间所进行的比较数目来估算其时间复杂性。 for 循环中的每一次循环都需要执行一次这样的比较，所以总的比较次数为n-1。函数Max还执行了其他的比较（for 循环中的每一次循环之前都要比较一下i 和n），这些比较没有包含在上述估算之中。其他的操作，比如初始化 pos以及循环控制变量i 的每次增值也没有包含在估算之中。如果把这些操作都纳入计数，则操作计数将增加一个常量。

例2-8 [多项式求值] 考察多项式$P(x)=\sum_{i=0}^{n} c_i x^n$。如果$c_n \neq 0$，则$P$是一个$n$维多项式。程序2-3可用来计算对于给定的值$x$，$P$（$x$）的实际取值。假定根据for循环内部所执行的加和乘的次数来估算时间复杂性。可以使用维数n 作为实例特征。进入for循环的总次数为n，每次循环执行1次加法和2次乘法（这种操作计数不包含循环控制变量i每次递增所执行的加法）。加法的次数为n，乘法的次数为2n。

程序2-3 对多项式进行求值的函数

```
template <class T>
T PolyEval(T coeff[], int n, const T& x)
{//计算n次多项式的值，coeff[0:n]为多项式的系数
  T y=1, value= coeff[0];
  for ( int i = 1; i <= n; i++)
      {//累加下一项
      y *= x;
      value += y * coeff[i];
      }
  return value;
}
```

Horner 法则采用如下的分解式计算一个多项式：

$$P(x) = (\cdots(c_n * x + c_{n-1}) * x + c_{n-2}) * x + c_{n-3}) * x + \cdots) * x + c_0$$

相应的C++函数见程序2-4。采用与程序2-3相同的方法，可以估算出该程序的时间复杂性为n 次加法和n 次乘法。由于函数PolyEval 所执行的加法数与Horner 相同，而乘法数是Horner 的两倍，因此，函数Horner 应该更快。

程序2-4 利用Horner 法则对多项式进行求值

```
template <class T>
T Horner(T coeff[], int n, const T& x)
{//计算n次多项式的值，coeff[0:n]为多项式的系数
  T value= coeff[n];
  for( int i = 1; i <= n; i++)
     value = value * x + coeff[n - i];
  return value;
}
```

例2-9 [计算名次] 元素在队列中的名次（rank）可定义为队列中所有比它小的元素数目加上在它左边出现的与它相同的元素数目。例如，给定一个数组 a=[4, 3, 9, 3, 7]作为队列，则各元素的名次为r =[2, 0, 4, 1, 3]。函数Rank（见程序2-5）可用来计算数组a[0: n-1]中各元素的名次。可以根据a的元素之间所进行的比较操作来估算程序的时间复杂性。这些比较操作是由 if语句来完成的。对于i 的每个取值，执行比较的次数为 i,所以总的比较次数为 1+2+3+⋯+n-1 = (n-1) n/2。

程序2-5 计算名次

```
template <class T>
void Rank(T a[], int n, int r[])
{//计算a[0:n-1]中n个元素的排名
  for ( int i = 1; i < n; i++)
    r[i] = 0;          //初始化
  //逐对比较所有的元素
  for ( int i = 1; i < n; i++)
     for ( int j = 1; j < i; j++)
```

```
        if (a [j] <= a[ i])  r[ i ]++;
        else r[j ]++;
    }
```

注意在估算时间复杂性时没有考虑for 循环的额外开销、初始化数组*r* 的开销以及每次比较a 中两个元素时对r进行增值的开销。

例2-10 [按名次排序] 一旦使用程序2-5计算出数组中每个元素的名次，就可以利用元素名次按照递增的次序对数组中的元素进行重新排列，使得 a[0]≤a[1]≤…≤a[n-1]。如果能够使用一个附加的数组u, 那么可以采用程序2-6中给出的函数Rearrange来重新排列元素的次序。

在函数Rearrange执行期间移动元素的次数为2n。（练习11要求怎样才能将移动次数减至n）。完成整个排序需要执行（n-1）n/2次比较操作和2n次移动操作。这种排序方法被称为计数排序(rank sort)。另外一种重排元素的函数见程序2-11，该函数没有使用附加数组u。

程序2-6 利用附加数组重排数组元素

```
template <class T>
void Rearrange (T a[], int n, int r[])
{//按序重排数组a中的元素，使用附加数组u
  T *u = new T[n+1];
  //在u中移动到正确的位置
  for ( int i = 1; i < n; i++)
    u[r[i]] = a[i];
  //移回到a中
  for ( int i = 1; i < n; i++)
     a[i]=u[i];
  delete [ ]u;
}
```

例2-11 [选择排序] 例2-10给出了一种按递增次序重排数组a中元素的方法。另外一种可选的方法是：首先找出最大的元素，把它移动到 a[n-1]，然后在余下的n-1个元素中寻找最大的元素并把它移动到a[n-2]，如此进行下去，这种排序方法为选择排序（selection sort），程序2-7中给出了实现这一过程的C++函数SelectionSort，其中函数Max在程序1-31中已经给出。可以按照元素的比较次数来估算函数的时间复杂性。从例 2-7中已经知道每次调用 Max(a,size)需要执行size-1次比较，所以总的比较次数为 1+2+3+…+n-1=(n-1)n/2。元素的移动次数为3(n-1)。选择排序所需要的比较次数与按名次排序（见程序 2-10）所需要的比较次数相同，但所需要的元素移动次数多出50%。本节的后面将介绍另外一种选择排序的方法。

程序2-7 选择排序

```
template <class T>
void SelectionSort (T a[], int n)
{//对数组a[0:n-1]中的n个元素进行排序
  for ( int size = n; size > 1; size--) {
     int  j= Max(a, size);
     Swap(a[j],a[size-1]);
     }
}
```

例2-12 [冒泡排序] 冒泡排序（bubble sort）是另一种简单的排序方法。这种排序采用一种“冒泡策略”把最大元素移到右部。在冒泡过程中，对相邻的元素进行比较，如果左边的元素大于右边的元素，则交换这两个元素。假定我们有四个元素 [5,3,7,1]。首先对5和3进行比较并交换，得到[3,5,7,1]，然后对5和7进行比较，两者无须交换，接下来比较 7和1并交换，得到[3,5,1,7]。在一次冒泡过程结束后，可以确信最大的元素肯定在最右边的位置上。函数Bubble(见程序2-8)执行一次冒泡过程，其中元素比较的次数为n-1。

程序2-8 一次冒泡

```
template <class T>
void Bubble (T a[], int n)
{//把数组a[0:n-1]中最大的元素通过冒泡移到右边
  for ( int i = 0; i < n-1; i++)
    if( a[i] > a[i +1]) Swap(a[i], a[i + 1]);
}
```

由于函数Bubble可以把最大的元素移到最右边，因此可以用它来替换 SelectionSort中的Max，从而得到一个新的排序函数（见程序 2-9）。在新函数中，元素比较的次数为 (n-1)n/2，与函数SelectionSort相同。

程序2-9 冒泡排序

```
template <class T>
void BubbleSort (T a[], int n)
{//对数组a[0:n-1]中的n个元素进行冒泡排序
  for ( int i = n; i>1; i--)
    Bubble(a,i);
}
```

最好、最坏和平均操作数

到目前为止，在给出的例子中所使用的实例特征都很简单（如输入数和 /或输出数），操作计数都是这些特征的良性函数（nice function）。如果把其他的一些操作也计算在内，其中有些例子就可能变得很复杂了。例如，由 Bubble(见程序2-8)所执行的交换次数不仅依赖于问题 n，而且依赖于数组a 中的具体值。交换次数可在0到$n-1$之间变化。由于操作计数不总是由所选择的实例特征唯一确定，那么我们可能会问，最好的、最坏的和平均的操作数分别是多少。下面就来讨论这个问题。

令P 是一个程序。假定把操作计数$O_p(n_1, n_2, ..., n_k)$ 视为实例特征$n_1, n_2, ..., n_k$ 的函数。对于任意程序实例I，令$operation_p(I)$ 为该实例的操作计数，令$S(n_1, n_2, ..., n_k)$ 为集合$\{I|I$ 具有特征$n_1, n_2, ..., n_k\}$。则P最好的操作计数为：

$$O_P^{BC}(n_1, n_2 ..., n_k) = min\{operation_p(I) \mid I \in S(n_1, n_2, ..., n_k)\}$$

P最坏的操作计数为：

$$O_P^{WC}(n_1, n_2 ..., n_k) = max\{operation_p(I) \mid I \in S(n_1, n_2, ..., n_k)\}$$

P平均或期望的操作计数为：

$$o_P^{AVG}(n_1, n_2, \cdots, n_k) = \frac{1}{\mid S(n_1, n_2, \cdots, n_k) \mid} \sum_{I \in S(n_1, \cdots, n_k)} operation_P(I)$$

公式O_P^{AVG}假定所有的I∈S都是完全相似的实例，否则该公式必须修改为：

$$o_P^{AVG}(n_1,n_2,\cdots,n_k)=\sum_{I\in S(n_1,n_2,\cdots,n_k)}p(I)operation_P(I)$$

其中$p(I)$是实例I可以被成功解决的概率（=次数/$|S(n_1, n_2, ..., n_k)|$）。

确定平均操作数通常是十分困难的，因此，在下面的几个例子中，仅分析最好和最坏的操作数。

例2-13 [顺序搜索] 在执行程序2-1顺序搜索的代码期间，我们想知道x 与a 中元素之间的比较次数，一个很自然的实例特征就是 n。不幸的是，比较的次数不是由 n 唯一确定的。例如，如果n=100且x=a[0]，那么仅需要执行一次操作；如果x 不等于a 中的任何一个元素，则需要执行100次比较。

当x 是a 中的一员时称查找是成功的，其他情况都称为不成功查找。每当进行一次不成功查找，就需要执行n 次比较。对于成功查找来说，最好的比较次数是 1，最坏的比较次数为n。为了计算平均查找次数，假定所有的数组元素都是不同的，并且每个元素被查找的概率是相同的。成功查找的平均比较次数如下：

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

例2-14 [向有序数组中插入元素] 程序2-10向一个有序数组a[0:n-1]中插入一个元素。a中的元素在执行插入之前和插入之后都是按递增顺序排列的。例如，如果向数组 a[0:5] = [1, 2, 6, 8, 9, 11]中插入4，所得到的结果为a[0:6] = [1, 2, 4, 6, 8, 9, 11]。当我们为新元素找到欲插入的位置时，必须把该位置右边的所有元素分别向右移动一个位置。对于本例，需要移动 11, 9, 8和6，并把4插入到新空出来的位置a[2]中。

程序2-10 向一个有序数组中插入元素

```
template<class T>
void Insert(T a[], int& n, const T& x)
{// 向有序数组 a[0:n-1]中插入元素x
 // 假定a 的大小超过 n
   int i;
   for (i = n-1; i >= 0 && x < a[i]; i--)
      a[i+1] = a[i];
   a[i+1] = x;
   n++; // 添加了一个元素
}
```

现在我们想知道执行程序2-10时，x 与a 中元素之间的比较次数。一个很自然的实例特征就是初始数组a 的大小n。最好或最少的比较次数为1，这种情况发生在x被插入到数组尾部的时候。最大的比较次数为n，这发生在x 被插入到a 的首部之时。为了估算平均比较次数，假定x 有相等的机会被插入到任一个可能的位置上（共有n+1个可能的插入位置）。如果x 最终被插入到a 的i+1处，i≥0，则执行的比较次数为n-i。如果x 被插入到a[0]，则比较次数为n。所以平均比较次数为：

$$\frac{1}{n+1}\left(\sum_{i=0}^{n-1}(n-i)+n\right)=\frac{1}{n+1}\left(\sum_{j=1}^{n}j+n\right)=\frac{1}{n+1}(n(n+1)/2+n)=n/2+n/(n+1)$$

例2-15 [再看按名次排序] 假定已经使用函数Rank(见例2-9中的程序2-5)计算出一个数组中每个元素的名次，可以在原地把该数组中的元素按序重排，方法是从 a[0] 开始，每次检查一个元素。如果当前正在检查的元素为a[i] 且r[i]=i，那么可以跳过该元素，继续检查下一个元素；如果r[i]≠i，可以把a[i]与a[r[i]] 进行交换，交换的结果是把原 a[i] 中的元素放到正确的排序位置（r[i]）上去。这种交换操作在位置 i 处重复下去，直到应该排在位置 i 处的元素被交换到位置 i 处。之后，继续检查下一个位置。程序2-11给出了原地重排数组元素的函数Rearrange。

程序2-11 原地重排数组元素

```
template<class T>
void Rearrange(T a[], int n, int r[])
{// 原地重排数组元素
  for (int i = 0; i < n; i++)
    // 获取应该排在a[i]处的元素
    while (r[i] != i) {
      int t = r[i];
      Swap(a[i], a[t]);
      Swap(r[i], r[t]);
      }
}
```

程序2-11执行的最少交换次数为0（初始数组已经是按序排列），最大的交换次数为2（n-1）。注意每次交换操作至少把一个元素移到正确位置（如a[i]），所以在n-1次交换之后，所有的n个元素已全部按序排列。因此，在最好的情况下，交换次数为 0，最坏情况下为2(n-1)(包括名次交换)。当使用本函数来代替程序2-6中的函数时，最坏情况下所需要的执行时间将增加，因为需要移动更多的元素（每次交换需要移动三次），不过程序所需要的内存减少了。

例2-16 [再看选择排序] 程序2-7中选择排序函数的一个缺点是：即使元素已经按序排列，程序仍然继续运行。例如，即使在第二次循环过后数组元素可能已经按序排列， for循环仍需要执行n-1次。为了终止不必要的循环，在查找最大元素期间，可以顺便检查数组是否已按序排列。程序2-12给出了一个按照这种思想实现的选择排序函数。在该函数中，把查找最大元素的循环直接与函数SelectionSort合并在一起，而不是把它作为一个独立的函数。

程序2-12 及时终止的选择排序

```
template<class T>
void SelectionSort(T a[], int n)
{// 及时终止的选择排序
  bool sorted = false;
  for (int size = n; !sorted && (size > 1); size--) {
    int pos = 0;
    sorted = true;
    // 找最大元素
    for (int i = 1; i < size; i++)
      if (a[pos] <= a[i]) pos = i;
      else sorted = false; // 未按序排列
    Swap(a[pos], a[size - 1]);
```

```
    }
}
```

对于程序2-12中的函数，其最好的情况出现在数组 a最初已是有序数组的情形，此时，外部for循环仅执行一次，a 中元素之间的比较次数为n-1。在最坏的情况下，外部for循环一直循环到size=1，执行的比较次数为（n-1）n/2。最好和最坏情况下的交换次数与程序2-7完全相同。注意在最坏的情况下，程序2-12可能要略微慢一些，因为它必须进行变量维护的额外工作。

例2-17 [再看冒泡排序] 与选择排序的情形一样，可以重新设计一个及时终止的冒泡排序函数。如果在一次冒泡过程中没有发生元素互换，则说明数组已经按序排列，没有必要再继续进行冒泡过程。程序2-13给出了一个及时终止的冒泡排序函数。在最坏情况下所执行的比较次数与原来的函数（见程序2-9）一样。在最好情况下比较次数为n-1。

程序2-13 及时终止的冒泡排序

```
template<class T>
bool Bubble(T a[], int n)
{//把 a[0:n-1] 中最大元素冒泡至右端
  bool swapped = false; // 尚未发生交换
  for (int i = 0; i < n - 1; i++)
    if (a[i] > a[i+1]) {
      Swap(a[i], a[i + 1]);
      swapped = true; // 发生了交换
      }
  return swapped;
}
template<class T>
void BubbleSort(T a[], int n)
{// 及时终止的冒泡排序
  for (int i = n; i > 1 && Bubble(a, i); i--);
}
```

例2-18 [插入排序] 程序2-10可以作为一个排序函数的基础。因为只有一个元素的数组是一个有序数组，所以可以从仅包含欲排序的n 个元素的第一个元素的数组开始。通过把第二个元素插入到这个单元数组中，可以得到一个大小为 2的有序数组。插入第三个元素可以得到一个大小为3的有序数组。按照这种方法继续进行下去，最终将得到一个大小为 n的有序数组，这种排序方式为插入排序（insertion sort），函数InsertionSort (见程序2-14) 正是按照这种思想实现的。为此，重写了函数Insert (见程序2-10)，因为它执行了一些不必要的操作。实际上，还可以把Insert 的代码直接嵌入到函数InsertionSort 之中，从而得到另外一个插入排序函数（见程序2-15）。等价地，也可以把函数Insert 作为一个内联（inline）函数。注意，如果用代码Insert (a, i, a[i]) 取代InsertionSort 函数中的for 循环体，则程序将无法运行，因为Insert的形式参数是一个引用参数。

程序2-14 插入排序

```
template<class T>
void Insert(T a[], int n, const T& x)
```

```
{// 向有序数组 a[0:n-1]中插入元素x
   int i;
   for (i = n-1; i >= 0 && x < a[i]; i--)
      a[i+1] = a[i];
   a[i+1] = x;
}
template<class T>
void InsertionSort(T a[], int n)
{// 对 a[0:n-1]进行排序
   for (int i = 1; i < n; i++) {
      T t = a[i];
      Insert(a, i, t);
      }
}
```

程序2-15 另外一种插入排序

```
template<class T>
void InsertionSort(T a[], int n)
   for (int i = 1; i < n; i++) {
      //将a[i]插入a[0:i-1]
      T t = a[i];
      int j;
      for (j = i-1; j >= 0 && t < a[j]; j--)
         a[j+1] = a[j];
      a[j+1] = t;
      }
}
```

程序2-14和程序2-15所执行的比较次数完全相同。在最好的情况下比较次数为 n-1,而在最坏的情况下比较次数为（n-1）n/2。

2.3.3 执行步数

从上一节关于操作计数的例子中可以看出，利用操作计数方法来估算程序的时间复杂性忽略了所选择操作之外其他操作的开销。在统计执行步数（ step-count）的方法中，要统计程序/函数中所有部分的时间开销。与操作计数一样，执行步数也是实例特征的函数。尽管任一个特定的程序可能会有若干个特征（如输入个数，输出个数，输入和输出的大小），但可以把执行步数看成是其中一部分特征的函数。通常选择一些感兴趣的特征，例如，如果要了解程序的运行时间（即时间复杂性）是如何随着输入个数的增加而增加的，在这种情况下，可以把执行步数仅看成是输入个数的函数。因此，在确定一个程序的执行步数之前，必须确切地知道将要采用的实例特征。这些特征不仅定义了执行步数表达式中的变量，而且定义了以多少次计算作为一步。

选择了相关的实例特征以后，可以定义一个操作步（ step)。操作步是独立于所选特征的任意计算单位，10次加法可以视为一步，100次乘法也可以视为一步，但n 次加法不能视为一步，其中n 为实例特征。$m/2$次加法或$p+q$ 次减法也都不能看成一步，其中m, p 和q 都是实例特征。

定义 [**程序步**] 程序步（program step）可以定义为一个语法或语义意义上的程序片段，该片段的执行时间独立于实例特征。

由一个程序步所表示的计算量可能与其他形式表示的计算量不同。例如，下面这条完整语句：

```
return a + b + b*c + ( a + b - c ) / ( a + b ) + 4;
```

可以被视为一个程序步，只要它的执行时间独立于所选用的实例特征。也可以把如下语句视为一个程序步：

```
x = y;
```

可以通过创建一个全局变量count (其初值为0)来确定一个程序或函数为完成其预定任务所需要的执行步数。可以把count 引入到程序语句之中，每当原始程序或函数中的一条语句被执行时，就为count 累加上该语句所需要的执行步数。当程序或函数运行结束时所得到的 count的值即为所需要的执行步数。

例2-19 把计算count 的语句引入到程序1-8之中，可以得到程序2-16。在程序2-16运行结束时所得到的count 的值即为程序1-8的执行步数。

程序2-16 统计程序1-8的执行步数

```
template<class T>
T Sum(T a[], int n)
{// 计算 a[0:n - 1]中元素之和
  T tsum = 0;
  count++; // 对应于tsum = 0
  for (int i = 0; i < n; i++) {
    count++; // 对应于for语句
    tsum += a[i];
    count++; // 对应于赋值语句
    }
  count++; // 对应于最后一个for语句
  count++; //对应于return语句
  return tsum;
}
```

程序2-17作为程序2-16的一个简化版本，仅仅用来确定 count值的改变。对于count的每一个初始值，程序2-16和程序2-17最终所得到的count值都一样。在程序2-17的for循环中， count的值被增加了2n。如果count的初值为0，则在程序结束时它的值将变成 2n+3。因此Sum (见程序1-8) 的每次调用需要执行2n+3步。

程序2-17 程序2-16的简化版本

```
template<class T>
T Sum(T a[], int n)
{//计算 a[0:n - 1]中元素之和
  for (int i = 0; i < n; i++)
    count += 2;
  count += 3;
```

```
  return 0;
}
```

例2-20 把计算count的语句引入到程序1-9之中，可以得到程序2-18。

令t_{Rsum}(n) 为程序2-18结束时count 所增加的值，可以看出t_{Rsum}(0)=2。当n>0时，count 所增加的值为2加上调用函数Rsum（从then 语句中）所增加的值。从t_{Rsum}(n) 的定义可知，额外的增值为t_{Rsum}(n-1)。所以，如果count 的初值为0，在程序结束时它的值将变成 2+t_{Rsum}(n-1)，其中n>0。

程序2-18 **统计程序**1-9**的执行步数**

```
template<class T>
T Rsum(T a[], int n)
{//计算 a[0:n - 1]中元素之和
  count++; // 对应于if 条件
  if (n > 0) {count++; // 对应于return 和 Rsum 调用
              return Rsum(a, n - 1) + a[n - 1];}
  count++; //对应于return
  return 0;
}
```

在分析一个递归程序的执行步数时，通常可以得到一个计算执行步数的递归等式（如t_{Rsum}(n) = 2 + t_{Rsum} (n - 1)， n > 0且t_{Rsum} (2) = 0）。这种递归公式被称为递归等式（ recurrence equation）,或简称为递归。可以采用重复迭代的方法来计算递归等式，如：

$$t_{Rsum}(n)=2+t_{Rsum}(n\text{-}1)=2+2+t_{Rsum}(n\text{-}2)=4+t_{Rsum}(n\text{-}2)\ldots=2n+t_{Rsum}(0)=2(n+1)$$

其中n≥0，因此函数Rsum（见程序1-9）的执行步数为2(n+1)。

比较程序1-8和程序1-9的执行步数，可以看到程序1-9的执行步数小于程序1-8的执行步数。不过不能因此断定程序1-8就比程序1-9慢，因为程序步不代表精确的时间单位。 Rsum中的一步可能要比Sum中的一步花更多的时间，所以Rsum有可能要比Sum慢（预计可能会这样）。

执行步数可用来帮助我们了解程序的执行时间是如何随着实例特征的变化而变化的。从Sum的执行步数中可以看到，如果n 被加倍，程序运行时间也将加倍（近似地）；如果n增加10倍，运行时间也会增加10倍。所以，可以预计，运行时间随着 n 线性增长。我们称Sum是一个线性程序（其时间复杂性与实例特征n 呈线性关系）。

例2-21 **[矩阵加]** 考察程序2-19, 它把存储在二维数组a[0:rows-1][0:cols-1]和b[0:rows-1][0:cols-1]中的两个矩阵相加。

程序2-19 **矩阵加法**

```
template<class T>
void Add( T **a, T **b, T **c, int rows, int cols)
{// 矩阵 a 和 b 相加得到矩阵 c.
  for (int i = 0; i < rows; i++)
    for (int j = 0;  j < cols; j++)
      c[i][j] = a[i][j] + b[i][j];
}
```

在程序2-19中引入count计数语句可得到程序2-20。程序2-21是程序2-20的一个简化版本，它计算同样的count值。检查程序2-21可以发现，如果count的初值为0，则在程序2-21结束时所得到的值为2rows*cols+2rows+1。

从上述分析中可以看出，如果 rows>cols，最好交换程序2-19中的两条for语句，这样执行步数将变成2rows*cols+2cols+1。注意，在本例中所使用的实例特征为rows和cols。

如果不想使用count计值语句，可以建立一张表，在该表中列出每条语句所需要的执行步数。建立这张表时，可以首先确定每条语句每一次执行所需要的步数以及该语句总的执行次数（即频率），然后利用这两个量就可以得到每条语句总的执行步数，把所有语句的执行步数加在一起即可得到整个程序的执行步数。

程序2-20 统计程序2-19的执行步数

```
template<class T>
void Add( T **a, T **b, T **c, int rows, int cols)
{//矩阵 a 和 b 相加得到矩阵 c.
  for (int i = 0; i < rows; i++) {
    count++; //对应于上一条for语句
    for (int j = 0;  j < cols; j++) {
      count++; // 对应于上一条for语句
      c[i][j] = a[i][j] + b[i][j];
      count++; // 对应于赋值语句
      }
    count++; // 对应j 的最后一次for循环
    }
  count++; //对应i 的最后一次for循环
}
```

程序2-21 程序2-20的简化版本

```
template<class T>
void Add( T **a, T **b, T **c, int rows, int cols)
{//矩阵 a 和 b 相加得到矩阵 c.
  for (int i = 0; i < rows; i++) {
    for (int j = 0;  j < cols; j++) {
      c[i][j] = a[i][j] + b[i][j];
      count += 2;
      }
    count += 2;
    }
  count++;
}
```

一条语句的程序步数与该语句每次执行所需要的步数（s/e）之间有重要的区别，区别在于程序步数不能反映该语句的复杂程度。例如语句：

```
x = Sum(a,m);
```

的程序步数为1，而该语句的执行所导致的count值的实际变化为1加上调用Sum所导致的count值的变化（比如2m+3）之和，因此该语句每次执行所需要的步数为 1+2m+3=2m+4。语句每次

执行所需要的步数通常记为s/e，一条语句的s/e就等于执行该语句所产生的count值的变化量。

图2-4列出了函数Sum(见程序1-8)中每条语句的执行频率及每次执行所需要的步数(s/e)。该程序所需要的总的执行步数为2n+3。注意for语句的频率为n+1而不是n，因为*i*必须递增到n，for语句才能结束。

语 句	s/e	频 率	总 步 数
T Sum(T a[], int n)	0	0	0
{	0	0	0
T tsum = 0;	1	1	1
for(int i=0;i<n;i++)	1	n+1	n+1
tsum += a[i];	1	n	n
return tsum;	1	1	1
}	0	0	0
总 计			2n+3

图2-4 程序1-8的执行步数

图2-5采用s/e或列表方法列出了函数Rsum(见程序1-9)的执行步数，而图2-6则列出了函数Add（程序2-19）的执行步数。

语 句	s/e	频 率	总 步 数
T Rsum(T a[], int n)	0	0	0
{	0	0	0
if(n> 0)	1	n+1	n+1
return Rsum(a,n-1) + a[n-1];	1	n	n
return 0;	1	1	1
}	0	0	0
总 计			2n+2

图2-5 程序1-9的执行步数

语 句	s/e	频 率	总 步 数
void Add(T **a, ...)	0	0	0
{	0	0	0
for (int i = 0; i < rows; i++)	1	rows+1	rows+1
for (int j = 0; j < cols; j++)	1	rows*(cols+1)	rows*cols+rows
c[i][j] = a[i][j] + b[i][j];	1	rows*cols	rows*cols
}	0	0	0
总 计			2rows*cols+2rows+1

图2-6 程序2-19的执行步数

程序2-22用来转置一个rows×rows的矩阵a[0:rows-1][0:rows-1]。b是a的转置，当且仅当对于所有的i和j，有b[i][j]=a[j][i]。

程序2-22 矩阵转置

```
template<class T>
void Transpose(T **a, int rows)
{// 对矩阵 a[0:rows-1][0:rows-1]进行转置
  for (int i = 0; i < rows; i++)
    for (int j = i+1;  j < rows; j++)
      Swap(a[i][j], a[j][i]);
}
```

图2-7给出了一个执行步数表。让我们来推导第二条 for循环语句的频率。对于i的每个值，该语句执行rows-i 次，所以其频率为：

$$\sum_{i=0}^{rows-1}(rows-i)=\sum_{1}^{rows}i=rows(rows+1)/2$$

Swap的频率为：

$$\sum_{i=0}^{rows-1}(rows-i-1)=\sum_{i=0}^{rows-1}i=rows(rows-1)/2$$

语 句	s/e	频 率	总 步 数
void Transpose(T **a, int rows)	0	0	0
{	0	0	0
for (int i = 0; i < rows; i++)	1	rows+1	rows+1
for (int j = i+1; j < rows; j++)	1	rows*(rows+1)/2	rows*(rows+1)/2
Swap(a[i][j], a[j][i]);	1	rows*(rows-1)/2	rows*(rows-1)/2
}	0	0	0
总 计			$rows^2$ +rows+1

图2-7 程序2-22的执行步数

在某些情况下，一条语句每次执行所需要的执行步数可能随时发生变化。例如，对于函数Inef (见程序2-23) 的赋值语句。函数Inef用非常低效的方式来计算数组元素的和：

$$\sum_{i=0}^{j}a[i]\text{对于}j=0,1,\cdots,n-1$$

程序2-23 低效的前缀求和程序

```
template <class T>
void Inef(T a[], T b[], int n)
{// 计算前缀和
  for (int j = 0; j < n; j++)
    b[j] = Sum(a, j + 1);
}
```

以上已经得出函数Sum(a,n) 的执行步数为2n+3，函数Inef中的赋值语句每执行一次所需要的步数为2j+6。在这两个结果中分别加上了1个执行步。函数Inef中赋值语句的频率为n，但该

语句总的执行步数并不是(2j+6) n，而是：

$$\sum_{j=0}^{n-1}(2j+6)=n(n+5)$$

图2-8给出了该函数的完整分析。

语　句	s/e	频　率	总 步 数
void Inef(T a[], T b[], int n)	0	0	0
{	0	0	0
for (int j = 0; j < n; j++)	1	n+1	n+1
b[j] = Sum(a, j + 1);	2j+6	n	n(n+5)
}	0	0	0
总　计			n^2+6n+1

图2-8　程序2-23的执行步数

最好、最坏和平均操作数的概念可以很容易地扩充到执行步数应用中。下面的例子就阐述了这些概念。

例2-22　[顺序搜索] 图2-9和图2-10分别给出了函数SequentialSearch（见程序2-1）在最好和最坏情况下的执行步数分析。

语　句	s/e	频　率	总 步 数
int SequentialSearch(T a[],T& x, int n)	0	0	0
{	0	0	0
int i;	1	1	1
for (int i = 0; i < n && a[i] != x; i++)	1	1	1
if (i == n) return -1;	1	1	1
return i;	1	1	1
}	0	0	0
总　计			4

图2-9　最好情况下程序2-1的执行步数

语　句	s/e	频　率	总 步 数
int SequentialSearch(T a[],T& x, int n)	0	0	0
{	0	0	0
int i;	1	1	1
for (i = 0; i < n && a[i] != x; i++)	1	n+1	n+1
if (i == n) return -1;	1	1	1
return i;	1	0	0
}	0	0	0
总　计			n+3

图2-10　最坏情况下程序2-1的执行步数

为了分析一个成功查找的执行步数，假定数组a中的n个值都互不相同并且在一个成功的查找过程中，x与数组中任何元素相匹配的概率都是一样的。在这样的假设下，一个成功查找的平均执行步数为n个可能的成功查找的执行步数之和除以n。为了得到这个平均值，首先来分析一下当x=a[j]时的执行步数（如图2-11所示），其中j介于[0，n-1]范围内。

现在可以计算出成功查找的平均执行步数为：

$$t^{AVG}_{SequentialSearch}(n) = \frac{1}{n}\sum_{j=0}^{n-1}(j+4) = (n+7)/2$$

这个值稍小于一个不成功查找的执行步数的一半。

语　句	s/e	频　率	总 步 数
int SequentialSearch(T a[],T& x, int n)	0	0	0
{	0	0	0
int i;	1	1	1
for (i = 0; i < n && a[i] != x; i++)	1	j+1	j+1
if (i == n) return -1;	1	1	1
return i;	1	1	1
}	0	0	0
总　计			j+4

图2-11　当x=a[j]时程序2-1的执行步数

现在假设成功查找出现的概率为　80%，并且每个 a[i]被查找的机会仍然相同，则SequentialSearch的平均执行步数为：

0.8 * (成功查找的平均步数) + 0.2 * (不成功查找的执行步数)

= 0.8(n+7)/2 + 0.2(n+3)

= 0.6n + 3.4

例2-23　[**向有序数组中插入元素**] 函数Insert 的最好和最坏执行步数分别如图 2-12和图2-13所示。

语　句	s/e	频　率	总 步 数
void Insert(T a[], int& n, const T& x)	0	0	0
{	0	0	0
for (int i = n-1; i >= 0 && x < a[i]; i--)	1	1	1
a[i+1] = a[i];	0	0	0
a[i+1] = x;	1	1	1
n++;	1	1	1
}	0	0	0
总　计			3

图2-12　在最好情况下程序2-10的执行步数

语 句	s/e	频 率	总 步 数
void Insert(T a[], int& n, const T& x)	0	0	0
{	0	0	0
for (int i = n-1; i >= 0 && x < a[i]; i--)	1	n+1	n+1
a[i+1] = a[i];	1	n	n
a[i+1] = x;	1	1	1
n++;	1	1	1
}	0	0	0
总 计			2n+3

图2-13 在最坏情况下程序2-10的执行步数

为了计算平均执行步数，假定x 被插入到任何位置上的概率是一样的（注：共有n+1个可能的位置）。如果x 最终被插入到位置j 处，j≥0，则执行步数为2n-2j+3。所以平均执行步数为：

$$\frac{1}{n+1}\left(\sum_{j=0}^{n}(2n-2j+3)\right)=\frac{1}{n+1}\left[2\sum_{k=0}^{n}k+3(n+1)\right]=n+3$$

练习

8. 试给出可能影响一个程序时间复杂性的其他因素。

9. 在函数Sum（见程序1-8）的for循环中会执行多少次加法？

10. 函数Factorial（见程序1-7）会执行多少次乘法？

11. 修改程序2-6,把第一个for循环后面的代码替换为：释放a所占用的空间并把a更新到u中去。此外，把参数T a[ ]变成T* &a。为了使新的代码能够运行，对应于a的实际参数必须是一个动态分配的数组。试测试新的代码。

12. 创建一个输入数组a，使得函数Rearrange（见程序2-11）执行n-1次元素交换和n-1次行交换。

13. 函数Add（见程序2-19）中矩阵元素对之间共执行了多少次加法？

14. 函数Transpose（见程序2-22）共执行了多少次Swap操作？

15. 试确定函数Mult（见程序2-24）共执行了多少次乘法，该函数实现两个n×n矩阵的乘法。

程序2-24 两个n×n矩阵的乘法

```
template<class T>
void Mult(T **a, T **b, T **c, int n)
{// 两个 n x n 矩阵 a 和 b 相乘得到c.
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) {
      T sum = 0;
      for (int k = 0; k < n; k++)
        sum += a[i][k] * b[k][j];
      c[i][j] = sum;
      }
}
```

16. 试确定函数Mult (见程序2-25) 共执行了多少次乘法，该函数实现一个m×n矩阵与一个n×p 矩阵之间的乘法。

程序2-25 一个m×n矩阵与一个n×p矩阵之间的乘法

```
template<class T>
void Mult(T **a, T **b, T **c, int m, int n, int p)
{// m x n 矩阵 a 与 n x p 矩阵 b相乘得到c
  for (int i = 0; i < m; i++)
    for (int j = 0; j < p; j++) {
      T sum = 0;
      for (int k = 0; k < n; k++)
        sum += a[i][k] * b[k][j];
      c[i][j] = sum;
      }
}
```

17. 试确定函数 Perm(见程序1-10)共执行了多少次Swap操作？

18. 函数MinMax(见程序2-26) 用来查找数组a[0:n-1] 中的最大元素和最小元素。令n为实例特征。试问a 中元素之间的比较次数是多少？程序 2-27中给出了另一个查找最大和最小元素的函数。在最好和最坏情况下a 中元素之间比较次数分别是多少？试分析两个函数之间的相对性能。

程序2-26 查找最大和最小元素

```
template<class T>
bool MinMax(T a[], int n, int& Min, int& Max)
{// 寻找 a[0:n-1]中的最小元素和最大元素
 // 如果数组中的元素数目小于1，则返回 false
  if (n < 1) return false;
  Min = Max = 0; // 初始化
  for (int i = 1; i < n; i++) {
    if (a[Min] > a[i]) Min = i;
    if (a[Max] < a[i]) Max = i;
    }
  return true;
}
```

程序2-27 另一个查找最大和最小元素的函数

```
template<class T>
bool MinMax(T a[], int n, int& Min, int& Max)
{// 寻找 a[0:n-1]中的最小元素和最大元素
 // 如果数组中的元素数目小于1，则返回 false
  if (n < 1) return false;
  Min = Max = 0; // 初始化
  for (int i = 1; i < n; i++)
    if (a[Min] > a[i]) Min = i;
    else if (a[Max] < a[i]) Max = i;
```

```
  return true;
}
```

19. 递归函数SequentialSearch(见程序2-2)中数组a与x 之间共执行了多少次比较？

20. 程序2-28给出了另外一个迭代式顺序搜索函数。在最坏情况下 x 与a中元素之间执行了多少次比较？把这个比较次数与程序 2-1中相应的比较次数进行比较，哪一个函数将运行得更快？为什么？

程序2-28 另一个顺序搜索函数

```
template<class T>
int SequentialSearch(T a[ ], const T& x, int n)
{// 在未排序的数组 a[0:n-1]中查找 x
 // 如果找到，则返回相应位置，否则返回-1
  a[n] = x; // 假定该位置有效
  int i;
  for (i = 0; a[i] != x; i++);
  if (i == n) return -1;
  return i;
}
```

21. 1) 在程序2-29中所有合适的位置插入count计值语句。

2) 通过删除原执行语句对 1) 中所得到的程序进行简化。简化后的程序所计算出的 count值应与1中程序所计算出的count值相同。

3) 当程序结束时count的值是多少？可以假定count的初值为0。

4) 采用频率方法分析程序2-29的执行步数，列出执行步数表。

程序2-29 练习21的函数

```
void D(int x[], int n)
{
  for (int i = 0; i < n; i += 2)
    x[i] += 2;
  int i = 1;
  while (i <= n/2) {
     x[i] += x[i+1];
     i++;
     }
}
```

22. 分别对如下函数完成练习21：

1) Max(见程序1-31)。

2) MinMax(见程序2-26)。

3) MinMax(见程序2-27)，分析最坏情况下的执行步数。

4) Factorial(见程序1-7)。

5) PolyEval(见程序2-3)。

6) Horner(见程序2-4)。

7) Rank(见程序2-5)。

8) Perm(见程序1-10)。

9) SequentialSearch (见程序2-28)，分析最坏情况下的执行步数。

10) SelectionSort (见程序2-7)，分析最好和最坏情况下的执行步数。

11) SelectionSort (见程序2-12)，分析最好和最坏情况下的执行步数。

12) InsertionSort (见程序2-14)，分析最坏情况下的执行步数。

13) InsertionSort (见程序2-15)，分析最坏情况下的执行步数。

14) BubbleSort (见程序2-9)，分析最坏情况下的执行步数。

15) BubbleSort (见程序2-13)，分析最坏情况下的执行步数。

16) Mult (见程序2-24)。

23. 对如下函数完成练习21中的1、2和3:

1) Transpose(见程序2-22)

2) Inef(见程序2-23)

24. 计算如下函数的平均执行步数：

1) SequentialSearch(见程序2-2)

2) SequentialSearch(见程序2-28)

3) Insert (见程序2-10)

25. 1) 对于程序2-25完成练习21。

2) 在什么条件下适于交换最外层的两个for循环？

26. 试比较在最坏情况下，函数SelectionSort(见程序2-12)、函数InsertionSort(见程序2-15)以及函数Bubble Sort (见程序2-13) 中元素移动的次数。利用程序2-11完成按名次排序。请说明对于大型数组，这两种排序方法之间的相对性能。

27. 在最坏的情况下一个程序所需要的运行时间和内存都必须是最大的吗？证明你的结论。

2.4 渐进符号（O、Ω、Θ、o）

之所以要确定程序的操作计数和执行步数有两个重要的原因： 1）为了比较两个完成同一功能的程序的时间复杂性；2）为了预测随着实例特征的变化，程序运行时间的变化量。操作计数和执行步数都不能够非常精确地描述时间复杂性。在使用操作计数时，我们把注意力集中在某些“关键”的操作上，而忽略了所有其他操作。执行步数方法则试图通过关注所有的操作以便克服操作计数方法的不足，然而，“执行步”的概念本身就不精确。指令 x=y和x=y+z+(x/y)都可以被称为一步。正由于执行步数的不精确性，所以不便于用来进行比较。不过如果两个程序之间的执行步数相差非常大，比如一个为3n+3，一个为100n+10，则另当别论。我们可以非常保险地预测，一个执行步数为3n+3的程序将比执行步数为100n+10的程序需要更少的执行时间。但在这种情况下其实并不需要精确地知道执行步数为100n+10，类似于“大概为80n或85n或75n”就足以得到相同的结论。

如果有两个程序的时间复杂性分别为$c_1n^2 + c_2n$和c_3n，那么可以知道，对于足够大的n，复杂性为c_3n的程序将比复杂性为$c_1n^2 + c_2n$的程序运行得快。对于比较小的n值，两者都有可能成为较快的程序（取决于c_1，c_2和c_3）。如果c_1=1，c_2=2，c_3=100，那么当$n \le 98$时$c_1n^2 + c_2n \le c_3n$，当$n>98$时$c_1n^2 + c_2n > c_3n$；如果c_1=1，c_2=2，c_3=1000，那么当$n \le 998$时$c_1n^2 + c_2n \le c_3n$。因此，对于大多数情况，足以得出结论$c_1n^2 \le t_P^{WC}(n) \le c_2n^2$或$t_Q^{WC}(n, m) = c_1n + c_2m$，其中$c_1$和$c_2$为非负常数。

不管c_1、c_2和c_3的值是多少，总会存在一个门槛值，使得当n小于这个门槛值时，复杂性为

$c_3 n$的程序要比复杂性为 $c_1 n^2 + c_2 n$ 的程序运行得快。这个门槛值被称为均衡点（ breakeven point)。如果均衡点为0，那么复杂性为$c_3 n$ 的程序总是要比复杂性为$c_1 n^2 + c_2 n$ 的程序运行得快（或者至少一样快)。精确的均衡点不可能通过分析来确定，必须在计算机上实际运行程序才能确定，所以确定c_1、c_2 和c_3 的精确值并没有多大帮助。

前述讨论的目的是为了引入新的符号（或记号)，利用新符号可以写出关于程序时间和空间复杂性的具体公式（尽管不够精确)。这种符号被称为渐进符号（asymptotic notation)，它可以描述大型实例特征下时间或空间复杂性的具体表现。在下面的讨论中 $f(n)$ 表示一个程序的时间或空间复杂性，它是实例特征 n的函数。由于一个程序的时间和空间需求是一个非负值，所以可以假定对于n的所有取值，函数f的值非负。由于n表示一个实例特征，所以可以进一步假定$n \geq 0$。即将讨论的渐进符号允许我们对于足够大的n值，给出f的上限值和/或下限值。

2.4.1 大写O符号

大写O 符号给出了函数f的一个上限。

定义 **[大写O符号]** $f(n)=O(g(n))$ 当且仅当存在正的常数c 和n_0，使得对于所有的$n \geq n_0$，有$f(n) \leq c\, g(n)$。

上述定义表明，函数f 顶多是函数g 的c 倍，除非n 小于n_0。因此对于足够大的n (如$n \geq n_0$)，g 是f 的一个上限（不考虑常数因子c)。在为函数f 提供一个上限函数g 时，通常使用比较简单的函数形式。比较典型的形式是含有n 的单个项（带一个常数系数)。图2-14列出了一些常用的g 函数及其名称。对于图2-14中的对数函数$\log n$，没有给出对数基，原因是对于任何大于1的常数a 和b 都有$\log_a n = \log_b n / \log_b a$, 所以$\log_a n$ 和$\log_b n$ 都有一个相对的乘法系数$1/\log_b a$，其中a是一个常量。

函　数	名　称
1	常数
$\log n$	对数
n	线性
$n \log n$	n个$\log n$
n^2	平方
n^3	立方
2^n	指数
$n!$	阶乘

图2-14　常用的渐进函数

例2-24 **[线性函数]** 考察$f(n)=3n+2$。当$n \geq 2$时，$3n+2 \leq 3n+n=4n$，所以$f(n)=O(n)$，$f(n)$是一个线性变化的函数。采用其他方式也可以得到同样结论，例如，对于 $n>0$，有$3n+2 \leq 10n$，可以通过选择$c=10$以及$n_0>0$来满足大O 定义。此外，由于当$n \geq 1$时，有$3n+2 \leq 3n+2n=5n$，所以通过选择$c=5$以及$n_0=1$也可以满足大O 定义。用来满足大O 定义的c和n_0的值并不重要，因为只需说明$f(n)=O(g(n))$，公式中并未出现c和n_0。

对于函数$f(n)=3n+3$，当$n \geq 3$时，有$3n+3 \leq 3n+n=4n$，所以$f(n)= O(n)$。类似地，对于$n \geq$

n_0=6，有$f(n)$=100n+6≤100n+n=101n，因此，100n+6=O(n)。正如所期望的那样，3n+2, 3n+3以及100n+6都满足n的大O定义，也即它们都是线性函数（对于一定的n）。

例2-25 [平方函数] 假定$f(n) = 10n^2 + 4n + 2$。对于n≥2, 有$f(n)$≤$10n^2 + 5n$。由于当n≥5时有5n≤n^2，因此对于n≥n_0=5，$f(n)$ ≤$10n^2 + n^2 = 11n^2$，所以$f(n)$= O(n^2)。

考察另外一个具有平方复杂性的例子$f(n)$=$1000n^2 + 100n - 6$。可以很容易地看出对于所有n, 有$f(n)$ ≤$1000n^2 + 100n$。此外，对于n≥100, 有100n≤n^2，因此对于n≥n_0=100, 有$f(n)$<1001 n^2, 因而$f(n)$= O(n^2)。

例2-26 [指数函数] 考察一个具有指数复杂性的例子$f(n)$= 6 * 2^n + n^2。可以观察到对于n≥4, 有n^2 ≤2^n，所以对于n≥4，有$f(n)$≤ 6 * 2^n + 2^n = 7 * 2^n，因此6*2^n + n^2 = O(2^n)。

例2-27 [常数函数] 当$f(n)$是一个常数时，比如$f(n)$=9或$f(n)$=2033，可以记为$f(n)$=O(1)。这种写法的正确性很容易证明。例如，对于$f(n)$=9≤9*1，只要令c=9 以及n_0=0 即可得$f(n)$=O(1)。同样地，对于$f(n)$ = 2033≤2033*1，只要令c=2033以及n_0=0即可。

例2-28 [松散界限] 当n≥2时有3n+3≤$3n^2$，所以3n+3=O(n^2)，虽然n^2是3n+3的一个上限，但不是最小上限，因为可以找到一个更小的函数（在本例中为线性函数）来满足大O定义。

当n≥2时有$10n^2$ +4n+2≤$10n^4$，所以$10n^2$ +4n+2=O(n^4)，但n^4同样不是$10n^2$ +4n+2的最小上限。

类似地，6$n2^n$ +20=O($n^2 2^n$)，但$n^2 2^n$不是最小上限，因为可以找到一个更小的上限为 $n2^n$。因此，6$n2^n$ +20 = O($n2^n$)。

注意在前面例子的中所使用的推导策略是：用次数低的项目（如 n）替换次数高的项目（如n^2），直到剩下一个单项为止。

例2-29 [错误界限] 3n+2≠O(1)，因为不存在c>0及n_0，使得对于所有的n≥n_0，有3n+2<c。可以使用反证法来证明这个结论。假定存在这样的c及n_0，使得对于所有的n≥n_0，有3n+2<c，则可以得到n<(c-2)/3，因此当n>max{ n_0, (c-2)/3}时，将出现矛盾的结论。

为了证明$10n^2$ +4n+2≠O(n)，首先假定$10n^2$ +4n+2=O(n)，因此，总存在一个正数c和n_0，使得对于所有的n≥n_0，有$10n^2$ +4n+2≤cn。不等式两边同时除以n可得：对于所有的n≥n_0，有10n+4+2/n≤c。这个不等式不总是成立的，因为不等式的左边随着 n的增长而增大，而不等式的右边则保持不变。比如取n>max{ n_0, (c-4)/10}，将得到矛盾的结论。

$f(n)$= $3n^2 2^n$ +4$n2^n$ +8n^2 ≠O(2^n)。为了证明这个不等式，假定$f(n)$ = O(2^n)，因此，总存在一个c > 0 和n_0，使得对于所有的n≥n_0，有$f(n)$≤c*2^n。不等式两边同时除以2^n 可得：对于所有的n≥n_0，有$3n^2$ +4n+8n^2 /2^n ≤c。与上例一样，这个不等式的左边随着 n的增长而增大，而不等式的右边则是一个常数，因此对于一个“足够大”的n，不等式将不成立。

如例2-28所示，语句$f(n)$ = O($g(n)$) 仅表明对于所有的n≥n_0，$cg(n)$ 是$f(n)$ 的一个上限。它并未指出该上限是否为最小上限，n= O(n^2), n= O($n^{2.5}$), n= O(n^3) 和n= O(2^n)。为了使语句$f(n)$=O($g(n)$) 有实际意义，其中的$g(n)$ 应尽量地小。因此常见的是3n+3=O(n)，而不是3n+3=O(n^2)，尽管后者也是正确的。

注意$f(n)$ = O($g(n)$) 并不等价于O($g(n)$) = $f(n)$。实际上，O($g(n)$) = $f(n)$ 是无意义的。在这里，使用符号＝是不确切的，因为＝通常表示相等关系。可以通过把符号＝读作“是”而不是“等于”来避免这种矛盾。

定理2-1给出了一个非常有用的结论，利用该结论可以获取$f(n)$的序（即$f(n)=O(g(n))$中的$g(n)$），这里，$f(n)$是一个关于n的多项式。

定理2-1 如果$f(n)=a_m n^m+\cdots+a_1 n+a_0$且$a_m>0$，则$f(n)=O(n^m)$。

证明 对于所有的$n\geq 1$有：

$$f(n)\leq \sum_{i=0}^{m}|a_i|n^i$$

$$\leq n^m\sum_{0}^{m}|a_i|n^{i-m}$$

$$\leq n^m\sum_{0}^{m}|a_i|$$

所以$f(n)=O(n^m)$。

例2-30 把定理2-1应用到例2-24、2-25和2-27中去。对于例2-24中的三个函数，都有$m=1$，因此这三个函数的序均为$O(n)$。对于例2-25中的函数，$m=2$，因此所有函数的序均为$O(n^2)$。而对于例2-27中的两个常量来说，$m=0$，因此这两个常量的序均为$O(1)$。

可以对例2-29中所采用的策略进行扩充，以证明一个上限尽管满足大O定义，但它并不是真正需要的上限，这正是定理2-2所要做的事情。采用定理2-2通常要比采用大O定义更容易证明$f(n)=O(g(n))$。

定理2-2 [大O比率定理] 对于函数$f(n)$和$g(n)$，若$\lim_{n\to\infty} f(n)/g(n)$存在，则$f(n)=O(g(n))$当且仅存在确定的常数$c$，有$\lim_{n\to\infty} f(n)/g(n)\leq c$。

证明 如果$f(n)=O(g(n))$，则存在$c>0$及某个n_0，使得对于所有的$n\geq n_0$，有$f(n)/g(n)\leq c$，因此$\lim_{n\to\infty} f(n)/g(n)\leq c$。接下来假定$\lim_{n\to\infty} f(n)/g(n)\leq c$，它表明存在一个$n_0$，使得对于所有的$n\geq n_0$，有$f(n)\leq max\{1,c\}*g(n)$。

例2-31 因为$\lim_{n\to\infty}(3n+2)/n=3$，所以$3n+2=O(n)$；因为$\lim_{n\to\infty}(10n^2+4n+2)/n^2=10$，所以$10n^2+4n+2=O(n^2)$；因为$\lim_{n\to\infty}(6*2^n+2^n)/2^n=6$，所以$6*2^n+n^2=O(2^n)$；因为$\lim_{n\to\infty}(2n^2-3)/n^4=0$，所以$2n^2-3=O(n^4)$；因为$\lim_{n\to\infty}(3n^2+5)/n=\infty$，所以$3n^2+5\neq O(n)$。

2.4.2 Ω符号

Ω符号与大O符号类似，它用来估算函数f的下限值。

定义 [Ω符号] $f(n)=\Omega(g(n))$当且仅当存在正的常数c和n_0，使得对于所有的$n\geq n_0$，有$f(n)\geq cg(n)$。

使用定义$f(n)=\Omega(g(n))$是为了表明，函数f至少是函数g的c倍，除非n小于n_0。因此对于足够大的n（如$n\geq n_0$），g是f的一个下限（不考虑常数因子c）。与大O定义的应用一样，通常仅使用单项形式的g函数。

例2-32 对于所有的n，有$f(n)=3n+2>3n$，因此$f(n)=\Omega(n)$。同样地，$f(n)=3n+3>3n$，所以有

$f(n)=\Omega(n)$；$f(n)=100n+6>100n$，所以$100n+6=\Omega(n)$。因而$3n+2$，$3n+3$和$100n+6$都是带有下限的线性函数。

对于所有的$n\geq 0$，有$f(n)=10n^2+4n+2>10n^2$，因此$f(n)=\Omega(n^2)$。同样地，$1000n^2+100n-6=\Omega(n^2)$。由于$6*2^n+n^2>6*2^n$，所以$6*2^n+n^2=\Omega(2^n)$。

同样也可以得到$3n+3=\Omega(1)$，$10n^2+4n+2=\Omega(n)$，$10n^2+4n+2=\Omega(1)$，$6*2^n+n^2=\Omega(n^{100})$，$6*2^n+n^2=\Omega(n^{50.2})$，$6*2^n+n^2=\Omega(n^2)$，$6*2^n+n^2=\Omega(n)$和$6*2^n+n^2=\Omega(1)$。

为了说明$3n+2\neq\Omega(n^2)$，可以先假定$3n+2=\Omega(n^2)$，因此存在正数c和n_0，使得对于所有的$n\geq n_0$，有$3n+2\geq cn^2$，所以对于所有$n\geq n_0$，有$cn^2/(3n+2)\leq 1$，这个不等式不可能总成立，因为不等式的左边将会随着n 的增大而变得无限大。

与大O 定义中的情形一样，可能存在若干个函数$g(n)$满足$f(n)=\Omega(g(n))$。$g(n)$仅是$f(n)$的一个下限。为了使语句$f(n)=\Omega(g(n))$更有实际意义，其中的$g(n)$应足够地大。因此常用的是$3n+3=\Omega(n)$及$6*2n+n2=\Omega(2^n)$，而不是$3n+3=\Omega(1)$及$6*2^n+n^2=\Omega(1)$，尽管后者也是正确的。

定理2-3是与定理2-1相类似的一个定理，它是关于Ω符号的定理。

定理2-3 如果$f(n)=a_m n^m+\cdots+a_1 n+a_0$且$a_m>0$，则$f(n)=\Omega(n^m)$。

证明 见练习31。

例2-33 根据定理2-3可知，$3n+2=\Omega(n)$，$10n^2+4n+2=\Omega(n^2)$，$100n^4+3500n^2+82n+8=\Omega(n^4)$。

定理2-4是与定理2-2类似的一个定理，采用定理2-4通常要比采用Ω定义更容易证明$f(n)=\Omega(g(n))$。

定理2-4 [**Ω比率定理**] 对于函数$f(n)$和$g(n)$，若$\lim_{n\to\infty} g(n)/f(n)$存在，则$f(n)=\Omega(g(n))$对于确定的常数$c$，有$\lim_{n\to\infty} g(n)/f(n)\leq c$。

证明 见练习32。

例2-34 因为$\lim_{n\to\infty}(3n+2)=1/3$，所以$3n+2=\Omega(n)$；因为$\lim_{n\to\infty} n^2/(10n^2+4n+2)=0.1$，所以$10n^2+4n+2=\Omega(n^2)$。因为$\lim_{n\to\infty} 2^n/(6*2^n+n^2)=1/6$，所以$6*2^n+n^2=\Omega(2^n)$；因为$\lim_{n\to\infty} n/(6n^2+2)=0$，所以$6n^2+2=\Omega(n)$；因为$\lim_{n\to\infty} n^3/(3n^2+5)=\infty$，所以$3n^2+5\neq\Omega(n^3)$。

2.4.3 Θ符号

Θ符号适用于同一个函数g既可以作为f的上限也可以作为f的下限的情形。

定义［Θ符号］$f(n)=\Theta(g(n))$当且仅当存在正常数c_1，c_2和某个n_0，使得对于所有的$n\geq n_0$，有$c_1 g(n)\leq f(n)\leq c_2 g(n)$。

使用定义$f(n)=\Theta(g(n))$是为了表明，函数f介于函数g的c_1倍和c_2倍之间，除非n小于n_0。因此对于所有足够大的n (如$n\geq n_0$)，g既是f的上限也是f的下限（不考虑常数因子c）。与大O定义和Ω定义的应用一样，通常仅使用单项形式的g函数。

例2-35 从例2-24, 2-25, 2-26和2-32中可以得到：$3n+2=\Theta(n)$，$3n+3=\Theta(n)$, $100n+6=\Theta(n)$, $10n^2+4n+2=\Theta(n^2)$, $1000n^2+100n-6=\Theta(n^2)$, $6*2^n+n^2=\Theta(2^n)$。

由于对于$n\geq 16$，$\log_2 n<10*\log_2 n+4\leq 11*\log_2 n$，所以$10*\log_2 n+4=\Theta(\log_2 n)$。前面曾指出$\log_a n$等于$\log_b n$乘以一个常数，所以可以把$\Theta(\log_a n)$简写为$\Theta(\log n)$。

在例2-29中证明了$3n+2\neq O(1)$，所以$3n+2\neq\Theta(1)$。类似地，可以得到$3n+3\neq\Theta(1)$，$100n+6\neq\Theta(1)$。因为$3n+3\neq\Omega(n^2)$，所以$3n+3\neq\Theta(n^2)$；因为$10n^2+4n+2\neq O(n)$，所以$10n^2+4n+2\neq\Theta(n)$；因为$10n^2+4n+2\neq O(1)$，所以$10n^2+4n+2\neq\Theta(1)$。

因为$6*2^n+n^2\neq O(n^2)$，所以$6*2^n+n^2\neq\Theta(n^2)$。同样道理，$6*2^n+n^2\neq\Theta(n^{100})$，$6*2^n+n^2\neq\Theta(1)$。

正如前面所提到的，在实际应用过程中，仅使用乘法系数为1的g函数，所以几乎从来不采用$3n+3=O(3n)$，或$10=O(100)$，或$10n^2+4n+2=\Omega(4n^2)$，或$6*2^n+n^2=\Omega(6*2^n)$，或$6*2^n+n^2=\Theta(4*2^n)$，即使这些语句都是正确的。

定理2-5 如果$f(n)=a_m n^m+\cdots+a_1 n+a_0$且$a_m>0$，则$f(n)=\Theta(n^m)$。

证明 见练习31。

例2-36 根据定理2.5可知，$3n+2=\Theta(n)$，$10n^2+4n+2=\Theta(n^2)$，$100n^4+3500n^2+82n+8=(n^4)$。

定理2-6是与定理2-2和定理2-4类似。

定理2-6 [**Θ比率定理**] 对于函数$f(n)$和$g(n)$，若$\lim_{n\to\infty} f(n)/g(n)$及$\lim_{n\to\infty} g(n)/f(n)$存在，则$f(n)=\Theta(g(n))$当且仅当存在确定的常数$c$，有$\lim_{n\to\infty} f(n)/g(n)\leq c$及$\lim_{n\to\infty} g(n)/f(n)\leq c$。

证明 见练习32。

例2-37 因为$\lim_{n\to\infty}(3n+2)/n=3$且$\lim_{n\to\infty} n/(3n+2)=1/3<3$，所以$3n+2=\Theta(n)$；因为$\lim_{n\to\infty}(10n^2+4n+2)/n^2=10$且$\lim_{n\to\infty} n^2/(10n^2+4n+2)=0.1<10$，所以$10n^2+4n+2=\Theta(n^2)$；因为$\lim_{n\to\infty}(6*2^n+n^2)/2^n=6$且$\lim_{n\to\infty} 2^n/(6*2^n+n^2)=1/6<6$，所以$6*2^n+n^2=\Theta(2^n)$；因为$\lim_{n\to\infty}(6n^2+2)/n=\infty$，所以$6n^2+2\neq\Theta(n)$。

2.4.4 小写o符号

定义 [**小写o**] $f(n)=o(g(n))$当且仅当$f(n)=O(g(n))$且$f(n)\neq\Omega(g(n))$。

例2-38 [**小写o**] 因为$3n+2=O(n^2)$且$3n+2\neq\Omega(n^2)$，所以$3n+2=o(n^2)$，但$3n+2\neq o(n)$。类似地，$10n^2+4n+2=o(n^3)$，但$10n^2+4n+2\neq o(n^2)$。

2.4.5 特性

下面的定理可用于有关渐进符号的计算。

定理2-7 对于任一个实数$x>0$和任一个实数$\varepsilon>0$，下面的结论都是正确的：

1) 存在某个n_0使得对于任何$n\geq n_0$，有$(\log n)^x<(\log n)^{x+\varepsilon}$。
2) 存在某个n_0使得对于任何$n\geq n_0$，有$(\log n)^x<n$。
3) 存在某个n_0使得对于任何$n\geq n_0$，有$n^x<n^{x+\varepsilon}$
4) 对于任意实数y，存在某个n_0使得对于任何$n\geq n_0$，有$n^x(\log n)^y<n^{x+\varepsilon}$。
5) 存在某个n_0使得对于任何$n\geq n_0$，有$n^x<2^n$。

证明 可参考各个函数的定义。

例2-39 根据定理2-7，可以得到如下结论：$n^3+n^2\log n=\Theta(n^3)$；对于任意自然数k，$2^n/n^2=\Omega(n^k)$；$n^4+n^{2.5}\log^{20} n=\Theta(n^4)$；$2^n n^4\log^3 n+2^n n^4/\log n=\Theta(2^n n^4\log^3 n)$。

图2-15列出了一些常用的有关O，Ω和Θ的标记，在该表中除n以外所有符号均为正常数。

图2-16给出了一些关于和与积的有用的引用规则。其中，⊕可以是O、Ω、Θ之一。

图2-15和2-16为我们使用渐进符号来描述程序的时间复杂性（或执行步数）做好了准备。

$f(n)$		渐进符号
E1	c	$\oplus(1)$
E2	$\sum_{i=0}^{k} c_i n^i$	$\oplus(n^k)$
E3	$\sum_{i=1}^{n} i$	$\oplus(n^2)$
E4	$\sum_{i=1}^{n} i^2$	$\oplus(n^3)$
E5	$\sum_{i=1}^{n} i^k, k>0$	$\oplus(n^{k+1})$
E6	$\sum_{i=0}^{n} r^i, r>1$	$\oplus(r^n)$
E7	$n!$	$\oplus((n/e)^n)$
E8	$\sum_{i=1}^{n} 1/i$	$\oplus(\log n)$

图2-15 渐进标记

I1 $\{f(n)=\oplus(g(n))\} \rightarrow \sum_{n=a}^{v} f(n)=\oplus(\sum_{n=a}^{v} g(n))$

I2 $\{f_i(n)=\oplus(g_i(n)), 1\le i\le k\} \rightarrow \sum_{1}^{k} f_i(n)=\oplus(\max_{1\le i\le k}\{g_i(n)\})$

I3 $\{f_i(n)=\oplus(g_i(n)), 1\le i\le k\} \rightarrow \prod_{1}^{k} f_i(n)=\oplus(\prod_{1}^{k} g_i(n))$

I4 $\{f_1(n)=O(g_1(n)), f_2(n)=\Theta(g_2(n))\} \rightarrow f_1(n)+f_2(n)=O(g_1(n)+g_2(n))$

I5 $\{f_1(n)=\Theta(g_1(n)), f_2(n)=\Omega(g_2(n))\} \rightarrow f_1(n)+f_2(n)=\Omega(g_1(n)+g_2(n))$

I6 $\{f_1(n)=O(g(n)), f_2(n)=\Theta(g(n))\} \rightarrow f_1(n)+f_2(n)=\Theta(g(n))$

图2-16 关于O，Ω和Θ的引用规则

2.4.6 复杂性分析举例

让我们重新检查一下2.3.3节所分析的时间复杂性。对于函数Sum(见程序1-8)，已经知道$t_{Sum}(n)=2n+3$，所以$t_{Sum}(n)=\Theta(n)$。此外，$t_{Rsum}(n)=n(m+1)+2=\Theta(mn)$，$t_{Add}(m,n)=2mn+2n+1=\Theta(mn)$，$t_{Transpose}(n)=(n-1)(4n+2)/2=\Theta(n^2)$。

我们已经知道$t^{WC}_{SequentialSearch}(n)=n+3=\Theta(n)$。由于n+3是最坏情况下的复杂性，所以它是$t_{SequentialSearch}(n)$的一个上限，因此$t_{SequentialSearch}(n)=O(n)$。后面这个公式表明，存在正常数c和$n_0$，使得对于所有的$n\ge n_0$，函数SequentialSearch的计算时间受限于cn。注意$t_{SequentialSearch}(n)$实际上是一个多值函数，因为对于不同的n，它有不同的取值。此外，因为在最好的情况下，x=a[0]，此时，

$t_{SequentialSearch}(n)=4$，所以$t_{SequentialSearch}(n)=\Omega(1)$。$t^{AVG}_{SequentialSearch}(n)=\alpha(n+7)/2+(1-\alpha)(n+3)=\Theta(n)$，其中α是x存在于数组a中的概率。

尽管上一段中正确地使用了O、Ω、和Θ符号，但我们还是无法进行精确的执行步数分析。实际上，如果不需要确定精确的执行步数，可以很容易地确定渐进复杂性，步骤是首先确定程序中每条语句（或语句组）的渐进复杂性，然后把这些复杂性加起来。图2-17至图2-22给出了几个函数的渐进复杂性，但没有提供精确的执行步数分析。

语　句	s/e	频　率	总 步 数
T Sum(T a[], int n)	0	0	Θ(0)
{	0	0	Θ(0)
T tsum = 0;	1	1	Θ(1)
for(int i=0;i<n;i++)	1	n+1	Θ(n)
tsum += a[i];	1	n	Θ(n)
return tsum;	1	1	Θ(1)
}	0	0	Θ(0)

$$t_{sum}(n)=\Theta(\max\{g_i(n)\})=\Theta(n)$$

图2-17　函数Sum（程序1-8）的渐进复杂性

语　句	s/e	频　率	总 步 数
T Rsum(T a[], int n)	0	0	Θ(0)
{	0	0	Θ(0)
if(n)	1	n+1	Θ(n)
return Rsum(a,n-1) + a[n-1];	1	n	Θ(n)
return 0;	1	1	Θ(1)
}	0	0	Θ(0)

$$t_{Rsum}(n)=\Theta(n)$$

图2-18　函数Rsum（程序1-9）的渐进复杂性

语　句	s/e	频　率	总 步 数
void Add(T **a, ...)	0	0	Θ(0)
{	0	0	Θ(0)
for (int i = 0; i < rows; i++)	1	Θ(rows)	Θ(rows)
for (int j = 0; j < cols; j++)	1	Θ(rows*cols)	Θ(rows*cols)
c[i][j] = a[i][j] + b[i][j];	1	Θ(rows*cols)	Θ(rows*cols)
}	0	0	Θ(0)

$$t_{Add}(rows, cols)=\Theta(rows*cols)$$

图2-19　函数Add（程序2-19）的渐进复杂性

语 句	s/e	频 率	总 步 数
void Transpose(T **a, int rows)	0	0	$\Theta(0)$
{	0	0	$\Theta(0)$
for (int i = 0; i < rows; i++)	1	$\Theta(rows)$	$\Theta(rows)$
for (int j = i+1; j < rows; j++)	1	$\Theta(rows^2)$	$\Theta(rows^2)$
Swap(a[i][j], a[j][i]);	1	$\Theta(rows^2)$	$\Theta(rows^2)$
}	0	0	$\Theta(0)$

$$t_{Transpose}(rows) = \Theta(rows^2)$$

图2-20 函数Transpose（程序2-22）的渐进复杂性

语 句	s/e	频 率	总 步 数
void Inef(T a[], T b[], int n)	0	0	$\Theta(0)$
{	0	0	$\Theta(0)$
for (int j = 0; j < n; j++)	1	$\Theta(n)$	$\Theta(n)$
b[j] = Sum(a, j + 1);	2j+6	n	$\Theta(n^2)$
}	0	0	$\Theta(0)$

$$t_{Inef}(n) = \Theta(n^2)$$

图2-21 函数Inef（程序2-23）的渐进复杂性

语 句	s/e	频 率	总 步 数
int SequentialSearch(T a[],T& x, int n)	0	0	$\Theta(0)$
{	0	0	$\Theta(0)$
int i;	1	1	$\Theta(1)$
for (i = 0; i < n && a[i] != x; i++)	1	$\Omega(0)$, $O(0)$	$\Omega(0)$, $\Theta(0)$
if (i == n) return -1;	1	1	$\Theta(1)$
return i;	1	$\Omega(0)$, $O(0)$	
}	0	0	$\Theta(0)$

$$t_{SequentialSearch}(n) = \Omega(1)$$
$$t_{SequentialSearch}(n) = \Omega(n)$$

图2-22 函数SequentialSearch（程序2-1）的渐进复杂性

有时可以把$O(g(n))$、$\Omega(g(n))$和$\Theta(g(n))$分别解释成如下集合：

$O(g(n))=\{f(n) \mid f(n)= O(g(n))\}$

$\Omega(g(n))=\{f(n) \mid f(n)= \Omega(g(n))\}$

$\Theta(g(n))=\{f(n) \mid f(n)= \Theta(g(n))\}$

在这种解释下，诸如$O(g_1(n))=O(g_2(n))$和$\Theta(g_1(n))=\Theta(g_2(n))$这样的语句就有了明确的含义。在使用这种解释时，可以把$f(n)=O(g(n))$读作“$f(n)$是$g(n)$的一个大O成员”，另两种的读法类似。

按照执行步数来分析图2-17至2-22时，可把$t_p(n)=\Theta(g(n))$，$t_p(n)=O(g(n))$或$t_p(n)=\Omega(g(n))$看成一条程序P的语句（用于计算时间），因为每一步仅需要$\Theta(1)$的执行时间。

当有了使用表的经历以后，就会有要从全局角度来考察程序的渐进复杂性的需要。下面用几个例子来详细地阐述这种方法。

例2-40 [排列] 考察程序1-10中产生排列方式的代码。假定m=n-1。当k=m时，所需要的时间为$\Theta(n)$。当k<m时，将执行else语句，此时，for循环将被执行m-k+1次。由于每次循环所花费的时间为$\Theta(t_{Perm}(k+1, m))$，因此，当k<m时，$t_{Perm}(k, m)=\Theta((m-k+1)\ t_{Perm}(k+1, m))$。使用置换的方法，可以得到：$t_{Perm}(0, m)=\Theta((m+1)*(m+1)!)=\Theta(n*n!)$，其中n≥1。

例2-41 [折半搜索] 程序2-30是一个用来在有序数组a[0:n-1]中查找元素x的函数。变量left和right用来记录查找的起始点和结束点。开始时，将在0到n-1之间进行查找，所以left和right的初值分别为0和n-1。我们始终遵循以下规律：当且仅当x是a[left:right]中的元素时，x是a[0:n-1]中的元素。

程序2-30 折半搜索

```
template<class T>
int BinarySearch(T a[], const T& x, int n)
{// 在 a[0] <= a[1] <= ... <= a[n-1]中搜索 x
 // 如果找到，则返回所在位置，否则返回-1
   int left = 0; int right = n-1;
   while (left <= right) {
      int middle = (left + right)/2;
      if (x == a[middle]) return middle;
      if (x > a[middle]) left = middle + 1;
      else right = middle-1;
      }
   return -1; // 未找到x
}
```

搜索过程从x与数组中间元素的比较开始。如果x等于中间元素，则查找过程结束。如果x小于中间元素，则仅需要查找数组的左半部分，所以right被修改为middle-1。如果x大于中间元素，则仅需要在数组的右半部分进行查找，left将被修改为middle+1。这种搜索方法被称为折半搜索（binary search）。

While的每次循环（最后一次除外）都将以减半的比例缩小搜索的范围，所以该循环在最坏情况下需执行$\Theta(\log n)$次。由于每次循环需耗时$\Theta(1)$，因此在最坏情况下，总的时间复杂性为$\Theta(\log n)$。

例2-42 [插入排序] 程序2-15使用插入排序算法对n个元素进行排序。对于每个i值，最内部的for循环在最坏情况下时间复杂性为$\Theta(1)$，因此，在最坏情况下，程序2-15的时间复杂性最多为$\Theta(1+2+3+...+n)=\Theta(n^2)$。在最好的情况下，程序2-15的时间复杂性为$\Theta(n)$。程序的渐进复杂性可由$\Omega(n)$和$O(n^2)$给出。

小写o符号通常用于执行步数的分析。执行步数3n+O(n)表示3n加上上限为n的项。在进行这种分析时，可以忽略步数少于$\Theta(n)$的程序部分。

可以扩充O、Ω、Θ和o的定义，采用具有多个变量的函数。例如，$f(n, m)=O(g(n, m))$ 当且仅当存在正常量c, n_0和m_0，使得对于所有的$n \geq n_0$和所有的$m \geq m_0$，有$f(n, m) \leq cg(n, m)$。

练习

28. 仅使用O、Ω、Θ和o的定义来证明如下公式的正确性。不得使用定理 2-1至2-6或图2-15与2-16中的等式。

1) $5n^2 - 6n = \Theta(n^2)$
2) $n! = O(n^2)$
3) $2n^2 2^n + n\log n = \Theta(n^2 2^n)$
4) $\sum_{i=0}^{n} i^2 = \Theta(n^3)$
5) $\sum_{i=0}^{n} i^3 = \Theta(n^4)$
6) $n^{2^n} + 6 * 2^n = \Theta(n^{2^n})$
7) $n^3 + 10^6 n^2 = \Theta(n^3)$
8) $6n^3 / (\log n + 1) = O(n^3)$
9) $n^{1.001} + n\log n = \Theta(n^{1.001})$
10) $n^{k+\varepsilon} + n^k \log n = \Theta(n^{k+\varepsilon}),\ k \geq 0,\ \varepsilon > 0$

29. 采用定理2-2, 2-4和2-6完成练习28。

30. 证明以下等式不成立：

1) $10n^2 + 9 = O(n)$
2) $n^2 \log n = \Theta(n^2)$
3) $n^2 / \log n = \Theta(n^2)$
4) $n^3 2^n + 6n^2 3^n = O(n^3 2^n)$

31. 证明定理2-3和2-5。

32. 证明定理2-4和2-6。

33. 证明当且仅当$\lim_{n\to\infty} f(n)/g(n)=0$ 时$f(n) = o(g(n))$ 。

34. 证明图2-15中的等价性E5至E8是正确的。

35. 证明图2-16中的推理规则I5至I6是正确的。

36. 下面哪些规则是正确的？为什么？

1) $\{f(n) = O(F(n)), g(n) = O(G(n))\} \rightarrow f(n)/g(n) = O(F(n)/G(n))$
2) $\{f(n) = O(F(n)), g(n) = O(G(n))\} \rightarrow f(n)/g(n) = \Omega(F(n)/G(n))$
3) $\{f(n) = O(F(n)), g(n) = O(G(n))\} \rightarrow f(n)/g(n) = \Theta(F(n)/G(n))$
4) $\{f(n) = \Omega(F(n)), g(n) = \Omega(G(n))\} \rightarrow f(n)/g(n) = \Omega(F(n)/G(n))$
5) $\{f(n) = \Omega(F(n)), g(n) = \Omega(G(n))\} \rightarrow f(n)/g(n) = O(F(n)/G(n))$
6) $\{f(n) = \Omega(F(n)), g(n) = \Omega(G(n))\} \rightarrow f(n)/g(n) = \Theta(F(n)/G(n))$
7) $\{f(n) = \Theta(F(n)), g(n) = \Theta(G(n))\} \rightarrow f(n)/g(n) = \Theta(F(n)/G(n))$

8) $\{f(n)=\Theta(F(n)), g(n)=\Theta(G(n))\} \rightarrow f(n)/g(n)=\Omega(F(n)/G(n))$

9) $\{f(n)=\Theta(F(n)), g(n)=\Theta(G(n))\} \rightarrow f(n)/g(n)=O(F(n)/G(n))$

37. 计算以下函数的渐进复杂性，设计一个类似于图2-19至2-22的频率表。

1) Factorial(见程序1-7)
2) MinMax(见程序2-26)
3) MinMax(见程序2-27)
4) Mult(见程序2-24)
5) Mult(见程序2-25)
6) Max(见程序1-31)
7) PolyEval(见程序2-3)
8) Horner(见程序2-4)
9) Rank(见程序2-5)
10) Perm(见程序1-10)
11) SelectionSort(见程序2-7)
12) SelectionSort (见程序2-12)
13) InsertionSort (见程序2-14)
14) InsertionSort (见程序2-15)
15) BubbleSort (见程序2-9)
16) BubbleSort (见程序2-13)

2.5 实际复杂性

我们已经知道，一个程序的时间复杂性通常是其实例特征的函数，在确定程序的时间需求是如何随着实例特征的变化而变化时，这种函数将非常有用。我们也可以利用复杂性函数对两个执行相同任务的程序P和Q进行比较。假定程序P具有复杂性$\Theta(n)$，程序Q具有复杂性$\Theta(n^2)$，由此可以断定，对于“足够大”的n，程序P比程序Q快。为了说明这种推断的正确性，可以通过实际考察。对于某些常量c和所有的n，$n \geq n_1$，程序P的实际计算时间的上限为cn；对于某些常量d和所有的n，$n \geq n^2$，程序Q的实际计算时间的下限为dn^2。由于对于所有$n \geq c/d$，有$cn \leq dn^2$，因此每当$n \geq max\{n_1, n_2, c/d\}$时，程序$P$比程序$Q$快。

我们应该谨慎地使用上面推断中“足够大”的说法。在决定使用两个程序中的哪个程序时，必须了解所要处理的n是否真的足够大。如果程序P的实际运行时间为10^6n毫秒，程序Q的实际运行时间为n^2毫秒，若总有$n \leq 10^6$，那么优先使用的将是程序Q。

为了能体会各种函数是如何随着n的增长而变化的，可以仔细地研究图2-23和2-24。从图中可以看出，随着n的增长，$2n$的增长极快。事实上，如果程序需要2^n执行步，那么当n=40时，执行步数将大约为$1.1*10^{12}$。在一台每秒执行1 000 000 000步的计算机中，该程序大约需要执行18.3分钟；如果n=50，同样的程序在该台机器上将需要执行13天，当n=60时，需要执行310.56年；当n=100时，则需要执行$4*10^{13}$年。因此可以认定，具有指数复杂性的程序仅适合于小的n (典型地取$n \leq 40$)。

具有高次多项式复杂性的函数也必须限制使用。例如，如果程序需要n^{10}执行步，那么当n=10时，每秒执行1 000 000 000步的计算机需要10秒钟；当n=100时，需要3171年；n=1000时，将需要$3.17*10^{13}$年。如果程序的复杂性是n^3，则当n=1000时，需要执行1秒；n=10 000时，需

要110.67分钟；n=100 000时，需要11.57天。

log$n$	n	$n$log$n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4 096	65 536
5	32	160	1 024	32 768	4 294 967 296

图2-23 各种函数的值

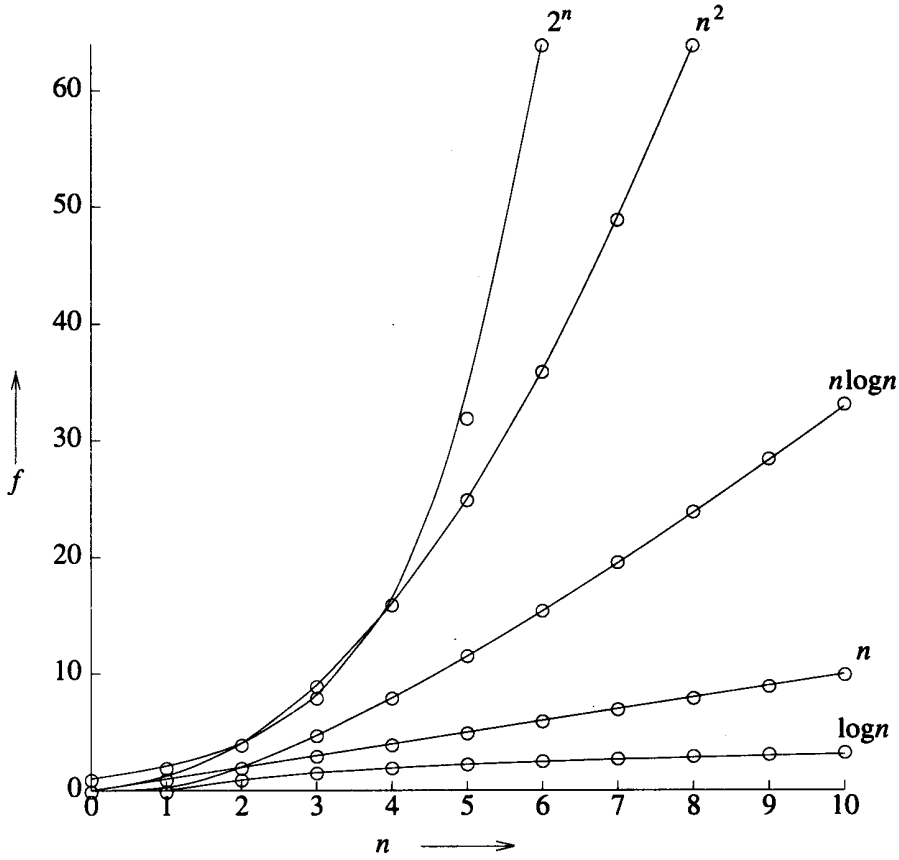

图2-24 各种函数的曲线

图2-25中给出了复杂性为$f(n)$的程序在每秒执行1 000 000 000条指令的计算机上执行时所需要的时间。应该注意到，目前只有世界上最快的计算机才能每秒执行 1 000 000 000条指令。从实际应用来看，对于相当大的n (比如n>100)，仅那些复杂性比较小(如n, $n\log n$, n^2, n^3) 的程序才是可行的，即使能够制造出每秒执行10^{12}条指令的计算机。 如果有这样的计算机，图2-25中的计算时间将分别减小1000倍。如n=100时，执行n^{10}条指令需耗时3.17年，执行2^n 条指令需耗时$4*10^{10}$年。

	f(n)						
n	n	$n\log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01us	.03us	.1us	1us	10us	10s	1us
20	.02us	.09us	.4us	8us	160us	2.84h	1ms
30	.03us	.15us	.9us	27us	810us	6.83d	1s
40	.04us	.21us	1.6us	64us	2.56ms	121d	18m
50	.05us	.28us	2.5us	125us	6.25ms	3.1y	13d
100	.10us	.66us	10us	1ms	100ms	3171y	$4*10^{13}$y
10^3	1us	9.96us	1ms	1s	16.67m	$3.17*10^{13}$y	$32*10^{283}$y
10^4	10us	130us	100ms	16.67m	115.7d	$3.17*10^{23}$y	
10^5	100us	1.66ms	10s	11.57d	3171y	$3.17*10^{33}$y	
10^6	1ms	19.92ms	16.67m	31.17y	$3.17*10^7$y	$3.17*10^{43}$y	

us=微秒=10^{-6}秒；ms=毫秒=10^{-3}秒；s=秒；m=分钟；h=小时；d=天；y=年

图2-25 每秒1 000 000 000条指令的机器上所需要的执行时间

练习

38. 设*A*和*B*是执行相同任务的程序，令$t_A(n)$和$t_B(n)$分别表示它们的运行时间。对于下面的每对数据，给出使程序*A*比程序*B*快的*n*的取值范围。

1) $t_A(n)=1000n$，$t_B(n)=10n^2$

2) $t_A(n)=2n^2$，$t_B(n)=n^3$

3) $t_A(n)=2^n$，$t_B(n)=100n$

4) $t_A(n)=1000n\log_2 n$，$t_B(n)=n^2$

39. 假定一台计算机每秒能执行1万亿条指令，重新给出图2-25中的数据。

40. 假定有某个程序和某台计算机，它们可以在“合理的时间”内解决规模为$n=N$的问题。建立一个表格来说明，对于同样的程序和速度快x倍的计算机，能在合理的时间内解决的问题的最大规模是多少（即最大的n值）。分别取x=10、100、1000、1 000 000及$t_A(n)=n$、n^2、n^3、n^5和2^n来完成练习。

2.6 性能测量

性能测量（performance measurement）主要关注于得到一个程序实际需要的空间和时间。按照前几节的说明，这些量与特定的编译器及编译器选项密切相关，同时也与执行程序的计算机密切相关。除非特别声明，本书中的所有性能值都是在 486/DX50 PC机上，用Borland C++5.01 for Windows95及缺省的编译器选项获得的。

我们忽略编译所需要的时间和空间是因为每个程序仅需编译一次（当然是在调试完成之后），而可以运行无数次。不过，如果测试的次数要比运行最终代码的次数多，则在程序测试期间，编译所需要的时间和空间也是很重要的。

基于以下原因，我们不能精确地测量一个程序运行时所需要的时间和空间：

- 指令空间和数据空间的大小是由编译器在编译时确定的，所以不必测量这些数据。
- 采用前几节介绍的方法，可以很准确地估算递归栈空间和变量动态分配所需要的空间。

为了得到一个程序的执行时间，需要一个定时机制。大多数C++产品都提供了相应的计时

函数。例如，Borland C++在它的time.h头文件中定义了clock()函数，该函数用于返回自程序启动以来所流逝的“滴答”数。把流逝的“滴答”数除以常量CLK_TCK，可以得到流逝的秒数。在PC计算机上，CLK_TCK=18.2。

假如希望测量程序2-15中的函数InsertionSort在最坏情况下所需要的时间，首先需要：

1) 确定需要测定其执行时间的n值

2) 对于上面的每个n，给出能导致最坏复杂性的测试数据。

2.6.1 选择实例的大小

可以根据以下两个因素来确定使用哪些n值：程序执行的时间及执行的次数。假定希望预测在最坏情况下，使用插入排序的方法对n个元素的数组进行排序所需要的时间。从例2-42中了解到，在最坏情况下，InsertionSort函数的复杂性为$\Theta(n^2)$，即n的平方函数。在理论上，如果知道任何三种n值所对应的执行时间，就可以确定这个平方函数，该函数可用来描述InsertionSort函数在最坏情况下的执行时间。利用所得到的平方函数，可以得到任何其他n值所对应的执行时间。在实践过程中，通常需要3个以上的n值，其原因如下：

1) 渐进分析仅给出了对于足够大的n值时程序的复杂性。对于小的n值，程序的运行时间可能并不满足渐进曲线。为了确定渐进曲线以外的点，需要使用多个n值。

2) 即使在满足渐进曲线的区间内，程序实际运行时间也可能不满足预定的渐进曲线，原因是在进行渐进分析时，忽略了许多低层次的时间需求。例如，一个程序的渐进复杂性为$\Theta(n^2)$，而它的实际复杂性可以是$c_1 n^2+c_2 n\log n+c_3 n+c_4$，或其他任何最高项为$c_1n^2$的函数，其中$c_1$为常量且$c_1>0$。

对于程序2-15，我们只期望获得n<100的渐进复杂性。所以，对于n>100的情况，可能只需要很少量的估算值。一个合理的选择是n=200, 300, 400, ..., 1000，这个选择并没有任何奥妙，也可以使用n=500, 1000, 1500, ..., 10000或n=512, 1024, 2048, ..., 2^{15}，后者将耗费更多的计算时间，而且会得到一些无法想象的时间值（相当巨大）。

对于[0, 100]范围内的n值，可以进行更精细的测量，因为我们并不很清楚渐进复杂性从何处开始有效。当然，如果测量结果表明，平方函数复杂性并不是从这个范围开始有效的，那么可对[100, 200]范围进行更细致的测量，如此进行下去，直到找到起始点。测算[0, 100]范围内的运行时间可以从n=0开始，此后，n每次递增10。

2.6.2 设计测试数据

对于许多程序，可以手工或用计算机来产生能导致最好和最坏复杂性的测试数据。然而，对于平均的复杂性，通常很难设计相应的数据。如对于InsertionSort，对任何n来说，能导致最坏复杂性的测试数据应是一个递减的序列，如n, n-1, n-2, ..., 1；导致最好复杂性的测试数据应是一个递增的序列，如0, 1, 2, ..., n-1。我们很难提供一组测试数据能使InsertionSort函数表现出平均的复杂性。

当不能给出能产生预期的复杂性的测试数据时，可以根据一些随机产生的测试数据所测量出的最小（最大，平均）时间来估计程序的最坏（最好，平均）复杂性。

2.6.3 进行实验

在确定了实例的大小并给出了测试数据以后，就可以编写程序来测量所期望的运行时间了。

对于插入排序，程序2-31给出了测试的过程。相应的测试数据见图2-26。

程序2-31 导致插入排序出现最坏复杂性的程序

```
#include <iostream.h>
#include <time.h>
#include "insort.h"
void main(void)
{
  int a[1000], step = 10;
  clock_t start, finish;
  for (int n = 0; n <= 1000; n += step) {
    // 获得对应于n 值的时间
    for (int i = 0; i < n; i++)
      a[i] = n - i; // 初始化
    start = clock( );
    InsertionSort(a, n);
    finish = clock( );
    cout << n << ' ' << (finish - start) / CLK_TCK << endl;
    if (n == 100) step = 100;
    }
  }
```

图2-26给出了这样的结论：排序100个元素以内的数组不需要任何时间，排序500~600个元素的数组所花费的时间是一样的。这个结论当然是错误的。对于其他的 n 值也有这种异常。问题出在对于计时函数clock () 来说，所需要的运行时间太小。而且，所有测量的精确度均为一个时钟“ 滴答 ”。由于在我们的计算机中， CLK_TCK = 18.2，因此测量的误差范围将是一个“ 滴答 ”时间1/18.2=0.055秒。对于n=1000，测试程序所报告的时间为6个“ 滴答 ”，因而实际的执行时间介于5到7个“ 滴答 ”之间。如果希望测量误差在 10%以内， finish-start至少应为10个时钟“ 滴答 ”或0.55秒，这时所得到的结果将与图2-26不同。

n	时间（s）	n	时间（s）
0	0	100	0
10	0	200	0.054945
20	0	300	0
30	0	400	0.054945
40	0	500	0.10989
50	0	600	0.109890
60	0	700	0.164835
70	0	800	0.164835
80	0	900	0.274725
90	0	1000	0.32967

图2-26 程序2-31的测试结果

为了提高测量的精确度，对于每个 n 值，可以重复排序若干次。由于排序会改变数组 a，因此需要在每次排序之前对该数组进行初始化。程序 2-32给出了新的测试程序。注意，现在所测量的时间为排序的时间、对a 进行初始化的时间以及while 循环所需要的额外时间。图2-27给出了实际的测量时间。

n	重复次数	总时间（s）	每次排序时间（s）
0	34228	0.549451	0.000016
10	10365	0.549451	0.000053
20	3525	0.549451	0.000156
30	1701	0.549451	0.000323
40	992	0.549451	0.000554
50	647	0.549451	0.000849
60	454	0.549451	0.001210
70	337	0.549451	0.001630
80	259	0.549451	0.002121
90	206	0.549451	0.002667
100	167	0.549451	0.003290
200	43	0.549451	0.012778
300	19	0.549451	0.028918
400	11	0.549451	0.049950
500	7	0.549451	0.078493
600	5	0.604396	0.120879
700	4	0.604396	0.151099
800	3	0.659341	0.219780
900	3	0.769231	0.256410
1000	2	0.604396	0.302198

图2-27 程序2-32的输出结果

通过注解程序2-32中的语句InsertionSort (a, n)，然后运行程序2-32，可以获得while 循环及初始化数组a所需要的时间。图2-28中给出了对于所选择的n 值，所得到的实际运行时间。从图2-27的每个时间中减去图2-28中的额外时间，可以得到InsertionSort 在最坏情况下的运行时间。可以看出，由于while 的条件实际被测试counter+1次，而while 循环体只执行了counter 次，因此，不精确性依然存在。不过，由于对于较小的 n 重复的次数很多，因此这种额外的开销可以被忽略。注意，对于较大的n，随着n 的加倍，图2-27中的时间相应地将变成4倍。这种情形是我们所期望的，因为程序最坏的复杂性为 $\Theta(n^2)$。

程序2-32 误差在10%以内的测试程序

```
#include <iostream.h>
#include <time.h>
#include "insort.h"
void main(void)
{
```

```
    int a[1000], n, i, step = 10;
    long counter;
    float seconds;
    clock_t start, finish;
    for (n = 0; n <= 1000; n += step) {
      // 获得对应于n 值的时间
      start = clock( ); counter = 0;
      while (clock( ) - start < 10) {
        counter++;
        for (i = 0; i < n; i++)
          a[i] = n - i; // 初始化
        InsertionSort(a, n);
        }
      finish = clock( );
      seconds = (finish - start) / CLK_TCK;
      cout << n << ' ' << counter << ' ' << seconds << ' ' << seconds / counter << endl;
      if (n == 100) step = 100;}
}
```

n	重 复 次 数	总时间（s）	每次排序时间（s）
0	36141	0.549451	0.000015
10	32321	0.549451	0.000017
50	19186	0.549451	0.000029
100	12999	0.549451	0.000042
500	3557	0.549451	0.000154
1000	1864	0.549451	0.000295

图2-28 图2-27测量中的额外开销

练习

41. 为什么程序2-31 的误差范围不在10%以内。

42. 利用程序2-32来获取两个不同的插入排序程序（见程序2-14和程序2-15）在最坏情况下所需要的运行时间。采用与程序 2-32相同的n 值。试比较调用Insert函数以及把Insert函数的代码合并到排序函数这两种情况下各自的优缺点。

43. 利用程序2-32来获取两个不同的冒泡排序程序（见程序 2-9和程序2-13）在最坏情况下所需要的运行时间。采用与程序 2-32相同的n 值。不过，你必须证实，程序 2-32中所给出的测试数据实际上也能使这两种冒泡排序函数产生最坏的复杂性。使用三列表格来给出你的结果，三列分别是： n、程序2-9、程序2-13。指出在最坏情形下，这两种冒泡排序函数的相对性能。

44. 1) 对于程序2-7和程序2-12中所给出的两个选择排序函数，设计能产生最坏复杂性的测试数据。

2) 对2-32进行适当的修改以测定这两种选择排序函数在最坏情况下所需要的时间。采用与

程序2-32相同的n 值。

3) 使用三列表格来给出你的结果，三列分别是：n、程序2-7、程序2-12。

4) 指出在最坏情形下，这两种选择排序函数的相对性能。

45. 比较插入排序（见程序2-34）、选择排序（见程序2-12）和冒泡排序（见程序2-13）在最坏情形下所需要的运行时间。为了一致，把程序2-13重写为一个函数。

1) 设计能使每种函数产生最坏复杂性的测试数据。

2) 使用1 中的数据及程序2-32中的测试程序来获取最坏情况下的运行时间。

3) 采用两种形式来描述这些时间：一是采用一个四列的表格，四列分别是：n、选择排序、冒泡排序、插入排序；二是采用一个显示三条曲线的图（每条曲线对应一种排序方法），图的x轴代表n 值，y 轴代表时间值。

4) 通过三种排序函数在最坏情形下的性能比较，能得出什么结论？

5) 对于每个n，测量额外的时间，并用图2-28的表格形式给出测试结果。从2 所得到的时间中减去这种额外开销，然后给出一个新的时间表和新的图。

6) 在减去额外开销后，在4 中所得到的结论是否发生了变化？

7) 利用已得到的数据，估计每种排序函数对2000、4000和10000个元素进行排序，在最坏情况下所需要的时间。

46. 修改程序2-32，以便获得InsertionSort函数（见程序2-15）的平均运行时间。要求如下：

1) 在每一次while 循环中，对0, 1, ..., n-1 的随机排列进行排序，这种随机排列是由一个随机排列产生器产生的。如果找不到这样的函数，可以用随机数生成器来编写，或简单地产生一个n 个数的随机序列。

2) 设置while循环，使得在一次循环中至少有20个随机排列被排序，并且至少需要耗费10个时钟“滴答”。

3) 用耗费的时间除以所排序的随机排列数目，得到平均排序时间。

47. 利用练习46中的策略来估算程序2-9和2-13中给出的冒泡排序函数的平均运行时间。采用与程序2-32相同的n 值。用表格形式给出测试结果。

48. 利用练习46中的策略来估算程序2-7和2-12中给出的选择排序函数的平均运行时间，采用与程序2-32相同的n 值。用表格形式给出测试结果。

49. 利用练习46中的策略来估算并比较程序2-12、程序2-13和2-15中所给出的排序函数的平均运行时间，采用与程序2-32相同的n 值。分别用表格和图的形式给出测试结果。

50. 编写测试程序来确定顺序搜索（见程序 2-1）和折半搜索（见程序 2-30）在搜索成功时所需要的平均时间。假定数组中每个元素被搜索的概率相同。用表格和图的形式给出结果。

51. 编写测试程序来确定顺序搜索（见程序2-1）和折半搜索（见程序2-30）在搜索成功时，最坏情况下所需要的时间。用表格和图的形式给出结果。

52. 对于n=10, 20, 30, ..., 100，确定函数Add（见程序2-19）的运行时间。用表格和图的形式给出测量结果。

53. 对于n=10, 20, 30, ..., 100，确定函数Transpose（见程序2-22）的运行时间。用表格和图的形式给出测量结果。

54. 对于n=10, 20, 30, ..., 100，确定函数Mult（见程序2-24）的运行时间。用表格和图的形式给出测量结果。

2.7 参考及推荐读物

下面的参考书给出了若干程序的渐进分析：

1) E.Horowitz, S.Sahni, *S.Rajasekaran. Fundamentals of Computer Algorithms/C++*. W.H.Freeman, 1997。

2) E.Horowitz, S.Sahni, D.Mehta. *Fundamentals of Data Structures in C++*. W.H.Freeman, 1995。

3) T.Cormen, C.Leiserson, R.Rivest. *Introduction to Algorithms*. McGraw-Hill, 1992。

4) G.Rawlins. *Compared to What: An Introduction to the Analysis of Algorithms*. W.H. Freeman, 1992。

5) B.Moret, H.Shapiro. *Algorithms from P to NP*第1卷: *Design and Efficiency*. Benjamin-Cummings, 1991。