第 1 章　基本概念

§1.1　概观：系统生命周期

本书读者应具备扎实的结构化程序设计技能。要获得这些技能，读者通常应学过程序设计基础一类课程。这类课程的培养目标就是传授结构化程序设计技能，但课程强调的是语言本身的语法形式与语句使用规则，学生在这个阶段通常只能编写很简单的程序，解决的问题不用说也是很简单的。这类简单问题，一般而言，只要直接选用程序设计语言提供的某语句也许就能完成求解，例如，用数组存储数据，再利用 `while` 循环语句，可能就足以解决这一阶段的许多问题了。

本书要指导读者向前迈一大步，大幅度提高编程能力，因为以后编写的程序，其规模要大很多，功能也要复杂得多。不用说，编写规模庞大而复杂的程序，不但需要更强有力的工具，还一定需要更高级的编程技术。我们希望在随后的学习过程，读者应扎实掌握数据的抽象思维方法，同时必须熟练掌握算法的规范声明、算法的性能分析、算法的性能评价等诸多技能。设置本章的目的就是要详细论述这些内容。此外，递归程序设计方法同样至关重要，读者也必须熟练掌握，因此也是本章讨论的内容，但论述得较为简明而且篇幅不很大。我们提请读者注意，如果读者以前对递归程序设计基础未给予足够重视，了解流于肤浅，那么必须仔细研读这方面内容，以后一定会深感大有益处。然而，在讨论各种工具与各项技术之前，我们必须强调，编程可不仅仅是写程序代码，即写完一条条程序语句就万事大吉了。与之截然相反，优秀的程序员有完全不同的观点。程序设计的首要问题，应该是首先把大规模程序系统分解成许许多多自成体系且相对独立的组成部件，然后再为各部分之间存在的相互调用，定义严格的调用格式。从系统观点来看形式各异的程序开发过程，实际上每种开发过程都有其生命周期，生命周期依次由需求、分析、设计、编码、验证诸多阶段构成。对于各阶段的论述，尽管可以分门别类，独立讨论，但各个阶段之间同时还会存在相当紧密的功能重叠与互相关联，为其划分时间顺序可以说仅仅是原始而粗略的。在选读材料与参考文献一节，我们列出了一些文献资料，供读者更加深入地了解系统的生命周期以及各阶段的详细含义。

(1) **需求阶段** 所有大规模程序设计项目，都是从确定规范声明开始，规范声明明确定义了项目的目标。需求用来描述程序员必须获得的信息，即，给定的条件（输入）应该是什么，生成的结果（输出）应该是什么。一般而言，刚开始的时候，规范声明往往粗略且粗糙，随后的求精过程应不断完善有关输入、输出的描述，直到包括全部细节，涵盖所有情形。

(2) **分析阶段** 仔细列出需求之后，就可以开始分析阶段的工作了。本阶段先把问题分解成规模适中且便于处理的各个部分。分析方法可以分为两类，一类是自底向上，一类是自顶向下。自底向上一类方法已遭摈弃，这类方法采用的策略不考虑结构信息，一开始就侧重编写代码的细节，显得杂乱无章。如果该类方法不幸被选用，而且程序员又缺乏大局观，那么最后的程序会成为相互之间松散连接又支离破碎的片段，这样，错误极可能蔓延不止，扩散到各处。打个比方，自底向上一类方法很像只用一个蓝图建各式房屋，根本不计房屋的功能和用途，而把所有房屋都看成由墙体、屋顶、水暖管道组成的呆板

建筑对象。这种做法实际上忽略了房屋的使用特性和居住特性，可以说完全缺乏全局考虑。虽然根本没人愿意住进这样建成的房屋，但还是有许多程序员，特别是新手程序员，却相信自己无需规划也可以编制出既优秀还不会出错的程序。

与自底向上一类方法不同，自顶向下方法一开始就要事先确定程序的指定目标，并利用该阶段所得成果，将程序分解成易于管理的组成成份。自顶向下方法会生成大量框图（diagrams），用于随后阶段，即系统设计阶段的设计依据。分析阶段根据问题种类的特点往往会提出多种候选方案，以后通过研究、比较，最后筛选出最佳方案。

(3) **设计阶段** 该阶段延续分析阶段的工作。设计人员从两方面进一步研究系统，这时，不但要考察程序所需的数据对象，还要考察针对数据对象而设置的各种操作。前一种考察的结果是创建抽象数据类型；后一种考察的结果更侧重算法的规范声明和算法的设计策略。以制订高校教学计划为例，这时，典型的数据对象应包括诸如学生、课程、教授等等；而典型的操作很可能包括在一种或多种数据对象之上完成插入、删除、查找等功能，特定的操作应包括向已开课程组成的集合之中插入新课程，还应包括查询某教授的主讲课程。

抽象数据类型和算法的规范声明与实现语言无关，因此编程的实现细节可以推迟。尽管目前必须确定每种数据对象所需的信息，不过这时却可以暂时忽略编写代码所需的实现细节。例如，假定已经确定了学生数据对象要包含姓名、社会保险号码、专业、电话号码，但这时尚未决定学生清单究竟该如何实现。到了后续章节，我们会了解到，实现学生清单实际上有多种数据结构可供挑选，例如，我们可以从数组、链表，树三种数据结构中按照需要来选择。设计阶段把实现细节尽量推迟有两方面好处，其一，可以为选择实现语言的种类留有余地，其二，可以留给我们充足的时间去选择最高效的实现语言。

(4) **求精与编写代码阶段** 到这一阶段，我们首先选择数据对象的（存储）表示，其次要实现各种操作的算法。上述先后次序很重要，因为算法的效率取决于数据对象的表示。也就是说，在确定数据表示之前，假如要考虑算法，那么算法应该与数据对象无关。

通常，进行到目前阶段，我们多半会发现目前完成的系统设计并不是最好的。一种可能是，我们这时得知，其他人也做过类似项目，而他们的方案比我们的方案更优越；另一种可能是，连我们自己也觉得现行方案欠佳，换一种设计方案也许可以更好。无论如何，如果原来的设计方案足够好，那么去吸收其它优点并非难事，可以比作锦上添花。实际上，避免过早地涉及编码细节，正是已经预计到这些情况都有可能出现，并事先做好了准备。如果现行方案不可行，就必须把全部方案推倒重来，由于这时没有过于沉重的包袱，所以，至少我们再提出新的设计方案，其代价就不至于过大，时间也不会花费太多，而且出错的可能也许更少了。因此，无论如何，即使要推倒重来，从代价方面来考虑，对我们多少也是个安慰。

(5) **正确性验证阶段** 本阶段工作有 3 方面内容，包括：① 证明程序正确，② 用合适的输入数据测试程序，③ 改正错误。这 3 方面内容业已经过深入、细致的研究，实施方法很多，本书显然不可能做全面彻底地讨论，所以，我们列出如下要点，并做一些简述。

- 正确性证明：程序的正确性证明与数学证明有类似之处，因而可以采用数学证明方法。然而，用数学方法证明程序的正确，不但耗时，而且对大规模程序设计项目也不适用。由于软件项目的开发时间有限，一般而言，要为所有程序都给出完全的数学证明并不现实。因而，选择那些已知是正确的算法，自然能大大减少出错机会。本书介绍的大多数算法，其正确性证明都是通过形式证明完成的，因而是实用的，也是可靠的。所以，我们要说，这些算法全都是程序员武器库中的利器。
- 测试：由于算法与特定语言无关，因而，证明程序的正确性可以早于编写代码阶段；然而，完成测试工作必须用到能够实际运行的代码，还必须使用真实的测试数据集。选取测试数据集的时候切记要谨慎，数据集应覆盖各种情形。新手最易犯的错误是：认为程序只要无语法错误就一定是正确的，显然，这样的想当然自然大错特错了。假如选取测试数据马马虎虎，比如只用一组数据去测试而根本不计其余，之后必将遗患无穷。合格的测试数据集应检查代码的所有片段，使无一遗漏，全部都应得到验证。举例来说，对于 **switch** 语句，测试数据必须检查所有 **case** 语句。

  测试的第一个目标是程序的正确性，这无疑至关重要，然而，程序的运行时间也同样重要。正确的程序，如果运行缓慢，仍无价值。本书的算法大多都有运行时间的理论估计，并给出了推导过程。对于运行时间仅涉及某段关键代码的情形，则应侧重该段代码的运行时间分析，本章同样也讨论这类局部估计问题。
- 纠错：如果一切进展顺利，正确性证明与测试结果会界定错误代码。读者应牢记，纠错的难易程度取决于早期设计阶段与编写代码阶段选择的决策。对于规模庞大、功能复杂的程序，假如忽视文档的编制和整理，加上代码又纠缠不清，象一盘意大利细面条，那就必然会成为程序员的恶梦，这是确定无疑的。去调试这样的程序，一次纠错极可能又会引入其它多处新错。与之截然相反，如果程序文档完整，模块自成体系，而且相互间参数调用关系清晰，那么调试工作将容易得多，这也是确定无疑的。因该说，如果可以去调试一个又一个独立的模块，之后再把它们合成为完整的系统，那么调试工作显然将更加容易了。

§1.2 指针和动态存储分配

§1.2.1 指针

指针是 C 语言的基本概念，C 语言中指针无处不在。实际上，每种数据类型 T，都有相应的指向 T 的指针类型。指针类型变量存放的值，实际上就是内存地址。指针类型有两个最基本的操作：

`&` 取地址操作

`*` 去引用（间接引用）操作

给定如下声明语句：

```
int i, *pi;
```

我们知道 `i` 是整型变量，而 `pi` 是指向整数的指针。如果令：

```
pi = &i;
```

则 &i 返回 i 的地址并把它赋值给 pi。要为 i 赋值，可以写

```
i = 10;
```

或者

```
*pi = 10;
```

以上两条赋值语句都把整数值 10 存储在 i 之中。第二条赋值语句 pi 前的 * 是去引用，10 并未存入指针里，而是存入指针 pi 指向的存储单元之中。

对指针还可以做其它操作。指针的值可以用来赋值给指针变量。指针是一个非负整数，因而 C 允许指针做算术运算，包括加、减、乘、除。指针之间可以做比较，结果返回大于、小于、相等三者之一。指针还可以通过强制显式地转换成整数。

指针变量的长度在不同的计算机中可以取不同值。有时，指针变量的长度在同一台计算机中也会不同。例如，指向 **char** 的指针变量长度也许比指向 **float** 的指针变量长度更长。C 用特殊的值 NULL 表示空指针。空指针不指向变量，也不指向函数。对于具体系统，空指针用整数值 0 表示，C 中 NULL 是一个宏，具体实现就定义为常量值 0。空指针可用在关系表达式中，表示布尔量“假”，因而，测试空指针的语句可以是：

```
if (pi == NULL)
```

或者可以是更简洁的形式：

```
if (!pi)
```

§1.2.2 动态存储分配

程序在运行时需要申请存储空间，用来存放信息，但在编程阶段，我们并不知道程序在运行时需要多大空间（例如，数组大小可能依赖于输入数据），也不想事先预留一块非常大的区域，因为其中很多空间也许根本就不会用到。针对这个问题，C 语言提供了一套机制，可以在程序运行时分配存储空间，这块区域称为*系统堆*（heap）。如果需要新的存储空间，我们可以调用函数 malloc 申请所需大小的一块内存空间。如果当前系统存在空闲内存，那么 malloc 函数返回指向这块空闲内存起始地址的指针；反之，如果当前系统没有空闲内存，则函数 malloc 返回指针 NULL。以后如果不再需要这块存储空间，可以调用函数 free 把它释放，交还给系统。一旦一块存储空间被释放，就不应再用它。程序 1.1 是申请、释放存储空间的例子，注意指针的用法。

程序 1.1 调用 malloc 的参量分别对应类型 **int** 和类型 **float** 的存储空间大小，返回值是指向这块存储空间第一个字节的指针。这个函数的返回类型对不同的系统可能不同，有些系统返回 **char** *，即指向 **char** 的指针。但 ANSI C 返回 **void** *。标记（**int** *）和（**float** *）是类型强制转换表达式，在程序 1.1 中本来可以缺省，这两个标记把返回类型变换为正确的类型。free 函数把以前用 malloc 申请的存储空间释放。在有些版本的 C 语言中，free 要求的参量类型是 **char** *，而 ANSI C 要求的参量类型是 **void** *。通常，free 的参量类型转换都是缺省的。

```
int i, *pi;
float f, *pf;
pi = (int *)malloc(sizeof(int));
pf = (float *)malloc(sizeof(float));
*pi = 1024;
*pf = 3.14;
printf("an␣integer␣=␣%d,␣a␣float␣=␣%f\n", *pi, *pf);
free(pi);
free(pf);
```

程序 1.1 分配、释放存储空间

如果存储空间不足，调用 malloc 会使申请失败，以下给出的代码更可靠，可以用来替换程序 1.1 中调用 malloc 的相应代码。

```
if ((pi =   (int *)malloc(sizeof(int)))   == NULL ||
    (pf = (float *)malloc(sizeof(float))) == NULL) {
    fprintf(stderr, "Insufficient␣memory");
    exit(EXIT_FAILURE);
}
```

或使用以下等价的代码

```
if (!(pi =   (int *)malloc(sizeof(int))) ||
    !(pf = (float *)malloc(sizeof(float)))) {
    fprintf(stderr, "Insufficient␣memory");
    exit(EXIT_FAILURE);
}
```

因为 malloc 在程序中经常出现，方便的做法是定义宏语句，在宏语句中调用 malloc，如果 malloc 失败则退出。以下是这种宏语句的一种实现：

```
#define MALLOC(p,s) \
  if (!(p) = malloc(s))) {\
      fprintf(stderr, "Insufficient␣memory"); \
      exit(EXIT_FAILURE); \
  }
```

现在，我们可以用两条语句替换程序 1.1 中调用 malloc 的语句。

```
MALLOC(pi, sizeof(int));
MALLOC(pf, sizeof(float));
```

在程序 1.1 中紧接 printf 语句之后插入如下一行：

```
pf = (float *) malloc(sizeof(float));
```

这时，现指针指向的存储空间，不再是存放 3.14 的存储单元。现在无法再访问原来那块存储空间。这是*悬空引用*（dangling reference）的一个例子。只要指向动态存储区域的指针丢掉了，原存储区域对程序而言，就丢掉了。在程序中使用指针和动态存储分配时，应牢记一点，如果不再需要一块动态存储空间，那么一定要把它还给系统。

§1.2.3 指针隐患

C 程序中让所有尚未指向实际目标的指针都取 NULL 值是好习惯。这样做，可以尽量避免访问一块尚未申请的空间，或访问一块我们并无权限访问的空间。有些计算机在涉及空指针操作时，返回 NULL，能够接着执行，而另一些系统，将直接对地址单元 0 操作，引发严重错误。

另一个好习惯是，在转换指针类型时，显式地使用强制类型转换，例如：

```
pi = malloc(sizeof(int));
     /* assign to pi a pointer to int */
pf = (float *) pi;
     /* casts an int pointer to a float pointer */
```

需要指出，在很多系统中，指针类型的大小和 **int** 类型的大小相同。由于 **int** 是函数返回的约定类型，一些程序员定义函数时会省略返回类型。函数的返回类型如果没有显式定义，那么这个约定的 **int** 返回类型以后可能被解释为指针。事实证明，这种习惯对某些系统是很危险的，因而，程序员应该明确指定函数的返回类型。

§1.3 算法形式规范

§1.3.1 综论

算法是计算机科学的基本概念。许多问题都有现成的求解算法，对于大规模计算机系统，设计高效算法是解决问题的核心。有鉴于此，我们必须首先来详细讨论算法概念。以下从算法定义开始。

定义 算法是指令的有限集合，顺序执行这些指令，可以完成特定的任务。同时，所有算法都遵循以下判据。

(1) **输入** 从外界获取零个或多个量。

(2) **输出** 产生至少一个量。

(3) **确定性** 每条指令清晰、无二义性。

(4) **有限性** 算法对所有情形都能在执行有限步之后结束。

(5) **有效性** 每条指令都是基本可执行的，意为借助纸、笔即可完成。判据 (3) 要求的确定性并不充分，每条指令还必须是能行的（feasible）。 □

计算理论范畴中的算法与程序有不同含义，计算理论中的程序无需满足以上的判据 (4)。例如，操作系统的工作模式是一个无限循环，无终止地等待为所有任务服务，这个系统程序从不结束，除非系统出错而崩溃。本书讨论的程序总会结束，因而，本书不区分算法和程序，这两个名词可以互换。

算法的描述方式可以多种多样，比如可以用自然语言，如英语，但这时必须保证每条指令都是确定的。借助图形也可以表示算法，称为流程图，但只适用小而简单的算法。本书描述算法都用 C 语言，不过有时会用英语与 C 语言的组合表示算法。以下给出两个例子，说明从问题描述到算法形成的转换过程。

例 1.1 (选择排序) 设计程序，将 $n \geq 1$ 个整数排序。以下是简单的求解过程：

```
From those integers that are currently unsorted,
find the smallest and place it next in the sorted list.
(在当前所有未排序的整数中，找出最小的一个，把它放在当前有序表的后一个位置。)
```

虽然这句话充分描述了排序问题，但它并不是算法，因为其中有几处疑问。例如，这句话没告诉我们这些整数开始时如何存放，也没告诉我们结果存放在哪里，更没告诉我们存放形式是什么。假定整数存放在数组中，数组名为 `list`，那么第 i 个整数应存放在第 i 个位置，即 `list[`i`]`，$0 \leq i < n$。程序 1.2 是构建算法的第一次尝试，注意它部分是 C 语言，部分是英语。

```
for (i=0; i<n; i++) {
    Examine list[i] to list[n-1] and suppose that the
    smallest integer is at list[min];

    Interchange list[i] and list[min];
}
```

程序 1.2 选择排序算法

要把程序 1.2 转换成真正的 C 语言，还必须实现两个已明确定义的子任务：一个是找出最小整数，一个是将当前最小值与 `list[`i`]` 交换。后一个子任务可以用函数（见程序 1.3）实现，也可以用宏实现。

```
void swap(int *x, int *y)
{/* both parameters are pointers to ints */
    int temp = *x; /* declares temp as an int and assignes
                      to it the contents of what x points to */
    *x = *y;       /* stores what y points to into the location
                      where x points */
    *y = temp;     /* places the contents of temp in location
                      pointed to by y */
}
```

程序 1.3 swap 函数

以下是这个函数 swap的用法。假定 a 和 b 都声明为 **int** 类型，要交换存放在两者中的值，调用：

```
swap(&a, &b);
```

传给 swap 的参数是 a 和 b 的地址。以下是交换两变量值的宏语句：

```
#define SWAP(x,y,t) ((t) = (x), (x) = (y), (y) = t)
```

两种实现各有优点，函数实现更易读，宏实现适用于所有变量类型。

现在，我们转向第一个子任务。假定当前的最小值在 list[i] 中，把它和 list[$i+1$], list[$i+2$], $\cdots$, list[$n-1$] 比较，只要找到更小值，就用它作最小值。这样，到 list[$n-1$] 整个工作就完成了。把所有这些工作组织在一起，就是函数 sort（见程序 1.4）。程序 1.4 是一个完整的程序，可以在计算机上运行。程序中用到定义在 <stlib.h> 中的函数 rand，它产生一系列随机数传给 sort。现在我们要问这个函数是否正确。

定理 1.1 对于 $n \geq 1$ 个元素的整数集合，函数 sort(list, n) 排序的结果是正确的。数据最后存放在在 list[0], $\cdots$, list[$n-1$] 中，且 list[0]$\leq$list[1]$\leq \cdots \leq$list[$n-1$]。

证明 当最外层 **for** 循环结束第 $i = q$ 次循环时，我们有 list[q]$\leq$list[r], $q < r < n$。接着，执行后续循环到 $i > q$，此时从 list[0] 到 list[q] 的内容不变。因此，当最外层 **for** 执行到最后一个循环时（即 $i = n-2$ 之后），有 list[0]$\leq$list[1]$\leq \cdots \leq$list[$n-1$]。 □

例 1.2 (折半查找) 假定 $n \geq 1$ 个有序整数存储在数组 list 之中，list[0]$\leq$list[1] $\leq \cdots \leq$ list[$n-1$]。我们想知道整数 searchnum 是否出现在这个表中，如果是，则返回下标 i，searchnum=list[i]；否则返回 -1。由于这个表有序，可以用以下方法查找给点值。

令 left、right 分别表示表中待查范围的左、右端点，初值为：left=0，right=$n-1$。令 middle=(left+right)/2，是表的中点下标值。 searchnum 和 list[middle] 比较的结果，有三种可能：

(1) searchnum<list[middle]。此时，如果 searchnum 在表中，它一定在位置 0 与 middle-1 之间，因此，把 right。设成 middle-1。

(2) searchnum=list[middle]。此时，返回 middle。

(3) searchnum>list[middle]。此时，如果 searchnum 在表中，它一定在位置 middle-1 与 $n-1$ 之间，因此，把 right 设成 middle-1。

当 searchnum 还没被查到，同时尚有没检查的其它整数，我们重新计算 middle，重复上述查找过程。程序 1.5 是这种查找策略的实现。算法包括两个子任务：(1) 表中是否还有未查找过的整数，(2) 比较 searchnum 和 list[middle]。

比较操作既可以用函数实现，也可以用宏实现。无论采用哪种实现，都需要分别为小于、等于、大于指定数值。我们遵循 C 语言库函数的习惯：

- 若前者小于后者，则返回负数 (-1)。

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE 101
#define SWAP(x,y,t) ((t) = (x), (x) = (y), (y) = t)

void sort(int [], int); /* selection sort */
void main(void)
{
  int i, n;
  int list[MAX_SIZE];
  printf("Enter␣the␣number␣of␣numbers␣to␣generate:␣");
  scanf("%d", &n);
  if (n < 1 || n > MAX_SIZE) {
    fprintf(stderr, "Improper␣value␣of␣n/n");
    exit(EXIT_FAILURE);
  }
  for (i = 0; i < n; i++) { /* randomly generate numbers */
    list[i] = rand() % 1000;
    printf("%d␣␣", list[i]);
  }
  sort(list, n);
  pritnf("\n␣Sorted␣array:\n␣");
  for (i = 0; i < n; i++)   /* print out sorted numbers */
    printf("%d␣␣", list[i]);
  printf("\n");
}

void sort(int list[], int n)
{
  int i, j, min, temp;
  for (i = 0; i < n-1; i++) {
    min = i;
    for (j = i+1; j < n; j++)
      if (list[j] < list[min])
        min = j;
    SWAP(list[i], list[min], temp);
  }
}
```

程序 1.4 选择排序

```
while (there are more intergers to check) {
  middle = (left + right) / 2;
  if (searchnum < list[middle])
    right = middle - 1;
  else if (searchnum == list[middle])
    return middle;
  else left = middle + 1;
}
```

程序 1.5 查找有序表

- 若两者相等，则返回 0。
- 若前者大于后者，则返回正数 (1)。

我们虽然给出函数（程序 1.6）和宏两种实现，以后总是用宏实现，因为它适应各种数据类型。

```
int compare(int x, int y)
{ /* compare x and y, return -1 for less than, 0 for equal,
     1 for greater */
  if (x < y) return -1;
  else if (x == y) return 0;
  else return 1;
}
```

程序 1.6 比较两个整数

宏实现：

```
#define COMPARE(x, y) (((x) < (y)) ? -1 : ((x) == (y)) ? 0 : 1)
```

我们现在着手解决第一个子任务：确定是否还有未查找的数据。回忆最初的算法，比较操作之后，数组的下标值可能向左移，也可能向右移，这么一直移下去，最终或者找到，或者下标交叉，即左边下标值大于右边下标值。本来左、右下标是全表的两个端点，查找在两者之间进行，一旦交叉，则说明再也没有待查找的数据元素了。把上述内容合成后，我们得到下面的折半查找程序（见程序 1.7）。

以上查找策略称为**折半查找**。 □

上面两个例子用 C 函数实现算法，实际上，把大程序分解为易于处理的多个部分，每部分用一个或多个函数实现，是函数机制的根本目标。分解之后，各函数令程序更易读，同时，由于各函数可以分别测试，程序正确性的几率也提高了。通常，我们在定义函数之前，要先声明函数，这样的好处是，编译程序在分析过程遇到一个函数调用，已经知道该函数的合法定义将出现在后续代码。C 语言中的函数可以一组一组分开编译，创建逻辑上功能相关的函数库。

```
int binsearch(int list[], searchnum, int left, int right)
{/* search list[0] <= list[1] <= ... <= list[n-1] for searchnum.
    Return its position if found. Otherwise return -1. */
  int middle;
  while (left <= right) {
    middle = (left + right)/2;
    switch(COMPARE(list[middle], searchnum)) {
    case -1: left = middle + 1;
      break;
    case 0: return middle;
    case 1: right = middle - 1;
    }
  }
  return -1;
}
```

程序 1.7 查找顺序表

§1.3.2 递归算法

刚入行的程序员常常把函数看作被其他函数调用的对象，函数执行自身代码，然后返回调用程序，这种看法是片面的。事实上，函数不但可以调用自身（直接递归），还可以调用其它函数，其它函数也可以再调用该函数（间接递归）。递归函数调用机制，不仅功能强大，还常常可以极大地简化问题，也就是说，如果不用递归方法，算法可能非常复杂。递归之所以可以大大简化问题，正是这里讨论递归的原因。

计算机专业的一些学生，经常会把递归方法视为某种神秘的技巧，看作仅适用于一小类特定问题，如计算阶乘、计算 Ackermann 函数一类问题。这种观点很不全面。实际上，任何可以用赋值语句、**if-else** 语句、**while** 语句实现的函数，都是可以递归实现的，而且递归实现通常要比循环实现更易理解。

那么，什么时候用递归表述算法更合适呢？这里仅给出一种判据：如果问题的表述本身就是递归定义，那么，自然而然，采用递归方法就很合适；求解阶乘如此，求解 Fibonacci 数列也是如此，求解二项式系数还是如此。二项式系数的定义是：

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

该式可用下式递归计算：

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$

以下通过两个例子论述递归算法的构造过程。第一个例子把例 1.2 的折半查找函数转为递归函数。第二个例子生成一个表中字符的所有置换。

例 1.3 (折半查找) 程序 1.7 是折半查找的循环实现，要把这个函数转化为递归函数，我们所做的工作是：(1) 判断递归结束而建立边界条件，(2) 实现递归调用，每次递归调用都向彻底

解决问题的目标前进一步。通过仔细阅读程序 1.7 的函数 binsearch，我们发现查找的结束有两种情形：其一是查找成功（ list[middle]=searchnum），其二是查找不成功（左右下标交叉）。对查找成功的情形，程序无需改动；但 **while** 语句应替换成等价的 **if** 语句。

为了让递归调用一步步接近最终结果，函数调用参量分别是新的 left 下标值和新的 right 下标值。程序 1.8 是递归实现的折半查找。注意，尽管代码改动了，递归函数被调用的形式与非递归的形式完全一致。 □

```
int binsearch(int list[], searchnum, int left, int right)
{/* search list[0] <= list[1] <= ... <= list[n-1] for searchnum.
    Return its position if found. Otherwise return -1. */
  int middle;
  if (left <= right) {
    middle = (left + right)/2;
    switch(COMPARE(list[middle], searchnum)) {
    case -1: return binsearch(list, searchnum, middle + 1, right);
    case  0: return middle;
    case  1: return binsearch(list, searchnum, left, middle - 1);
    }
  }
  return -1;
}
```

程序 1.8 折半查找的递归实现

例 1.4 (置换) 给定 $n \geq 1$ 个元素的集合，打印这个集合所有可能的置换。例如，给定集合 $\{a, b, c\}$，它的所有置换是 $\{(a, b, c), (a, c, b), (b, a, c), (b, c, a), (c, a, b), (c, b, a)\}$。容易看出，给定 n 个元素，共有 $n!$ 种置换。我们通过观察集合 $\{a, b, c, d\}$，得到生成所有置换的简单算法，以下是算法的构造过程：

(1) a 跟在 (b, c, d) 的所有置换之后。

(2) b 跟在 (a, c, d) 的所有置换之后。

(3) c 跟在 (a, b, d) 的所有置换之后。

(4) d 跟在 (a, b, c) 的所有置换之后。

“跟在所有 $\cdots$ 置换之后”是构造递归算法的关键，这句话启发我们，如果解决了 $n-1$ 个元素的置换问题，则 n 个元素的置换问题就可以解决了。由此，我们写出程序 1.9。程序中的 list 是字符数组。注意，开始调用函数的形式是 perm(list, 0, $n-1$)，这个函数递归生成所有置换，直到 $i = n$。

读者可以用三个元素的集合 $\{a, b, c\}$ 仿真程序 1.9 的执行。每次递归调用 perm 都生成参数 list、i、n 的局部新副本，每次调用 i 都会改变，而 n 不变。参数 list 是指向数组的指针，数组的值在调用过程也保持不变。 □

```
void perm(char *list, int i, int n)
{ /* generate all the permutations of list[i] to list[n] */
  int j, temp;
  if (i == n) {
    for (j=0; j<=n; j++) printf("%c", list[j]);
    printf("␣␣␣␣");
  } else {
  /* list[i] to list[n] has more than one permutation,
     generate these recursively */
    for (j = i; j <= n; j++) {
      SWAP(list[i], list[j], temp);
      perm(list, i+1, n);
      SWAP(list[i], list[j], temp);
    }
  }
}
```

程序 1.9 递归置换生成程序

在后续章节，我们还要给出更多递归函数，本书大量算法都是递归算法，特别在第 4 章的表算法与第 5 章的二叉树算法中，递归都会大量出现。

习题

上面的例子论述了如何将问题转化成程序，我们避开了有关数据抽象与算法设计策略的内容，仅专注于由问题的英语描述向函数的转换过程，或由循环算法向递归算法的转换。下列习题要求读者练习以上技能。对每个编程题，试着先确定算法，然后把它翻译成函数，再证明它是正确的。正确性的“证明”方法，可以是算法分析，也可以用恰当的测试数据。

1. 请看以下两个句子：

   (a) 使方程 $x^n + y^n = z^n$ 有正整数解x, y, z 的最大 n 值是 2 吗？

   (b) 把 5 除以 0 存入 x 然后转到语句 10。

   两句话都不能满足算法判据中的某一条，指出分别是哪一条。

2. 给定多项式

$$A(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0 x^0,$$

   求多项式在 x_0 处的值，可用 Horner 规则。

$$A(x_0) = (\cdots((a_n x_0 + a_{n-1})x_0 + \cdots + a_1)x_0 + a_0),$$

   Horner 规则使多项式求值所需乘法次数最少。写出用 Horner 规则求值的 C 程序。

3. 给定 n 个布尔变量 $x_1, \cdots, x_n$，我们希望打印所有可能的真值组合。例如，如果 $n = 2$，共有四种可能：(true, true), (false, true), (true, false), (false, false)。用 C 语言编程实现。

4. 写一个 C 程序，按升序打印 x, y, z 的整数值。

5. 鸽笼原理指出，如果函数 f 有 n 个不同的输入，且只有小于 n 个输出，则一定有两个输入 $a, b, a \neq b$，使 $f(a) = f(b)$。写一个 C 程序，找出其值域相等的 a 和 b。

6. 给定正整数 n，判定 n 是否等于其因子的总和，即判定 n 是否等于所有 t 的和，t 可以整除 n，$1 \leq t < n$。

7. 阶乘函数 $n!$ 定义为：若 $n \leq 1$，其值为 1；若 $n > 1$，其值为 $n \times (n-1)$。写出计算阶乘的递归函数和循环函数。

8. Fibonacci 数定义为：$f_0 = 0, f_1 = 1$, 以及当 $i > 1$ 时，$f_i = f_{i-1} + f_{i-2}$。写出计算 f_i 的 C 语言递归函数和循环函数。

9. 写出计算二项式系数的循环函数，再把它转为等价的递归函数。

10. Ackerman 函数 $A(m, n)$ 定义为：

$$A(m,n) = \begin{cases} n+1, & \text{if } m = 0; \\ A(m-1, 1), & \text{if } n = 0; \\ A(m-1, A(m, n-1)), & \text{otherwise.} \end{cases}$$

    这个函数的特性是：即使 m, n 很小，它的增长也非常迅速，因此得到广泛研究。写出它的递归实现和循环实现。

11. (**Hanoi 塔**) 有三根柱子，64 个直径不等的空心碟子套在第一根柱子上，下面的碟子都比上面的碟子大。传说一些和尚要把这些碟子从第一根柱子移到第三根柱子，移动时必须遵循以下规则：

    (a) 一次只能移动一个碟子。

    (b) 大碟子不能放在小碟子上。

    写一个递归程序，打印移动碟子的过程。

12. 令 S 是 n 个元素的集合，它的幂集是它所有子集的集合。例如，如果 $S = \{a, b, c\}$，那么 $\texttt{powerset}(S) = \{\{\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$。写出递归函数实现 $\texttt{powerset}(S)$。

§1.4 数据抽象

本书读者无疑熟悉 C 语言的基本数据类型，如 **`char`**，**`int`**，**`float`**，**`double`**，**`int`** 还可以有修饰符 **`long`**，**`unsigned`**。从真实世界抽象出的问题，最后都是由这些数据类型表示

的。除了这些基本数据类型，C 语言还提供两种聚合数据类型机制，一种是数组，一种是结构体。数组是相同数据类型的集体，数组中所有数据的类型都相同，无需显示给出，例如，`int list[5]`，定义了含有五个（相同的）整数的数组，数组下标的范围是 $0, \cdots, 4$。结构体可以是不同元素类型的集体，这些不同的元素类型必须显式给出。例如，

```
struct student {
    char lastName;
    int studentId;
    char grade;
};
```

在这个结构体中定义了三个成员域，两个成员域是字符型，一个成员域是整型，结构体名称是 `student`。C 语言结构体的详细解释在第 2 章中给出。

所有程序设计语言都至少提供一个最小的、预定义的数据类型集合，还要提供构造新类型的机制，或由用户自定义类型的机制。现在我们要问："数据类型是什么？"

定义 数据类型是数据对象和施加在数据对象上操作的聚合体。 □

无论程序中用到的是预定义数据类型，或自定义数据类型，都应考虑数据对象和数据操作两方面内容。例如，数据类型 **int** 的数据对象是 $\{0, +1, -1, +2, -2, \cdots,$ INT_MAX, INT_MIN$\}$，其中 INT_MAX 和 INT_MIN 是机器所能表示的最大整数和最小整数（这两个值定义在 C 语言的头文件 `limits.h` 之中）。整数操作有很多，当然包括算术运算 $+, -, *, /,$ %。其它操作还包括测试两数是否相等，以及赋值操作。这些操作都有操作名称。操作可以是前缀算符，如 `atoi`，或者是中缀算符，如 $+$。不管数据操作是由语言本身定义的，或者是由库函数定义的，其名称、参量、以及执行结果都应指定。

除了必须了解数据类型的操作之外，我们还应了解数据对象是如何表示的。例如，大多数计算机的 **char** 类型表示占用一个字节存储空间的比特串，而 **int** 可能占用 2 个或 4 个字节的存储空间，如果 **int** 占用 2 个字节，那么 $\texttt{INT\_MAX} = 2^{15} - 1 = 32767$。

了解数据类型中数据对象的表示格式，当然很有用处，但使用不当也有可能带来危险。如果已知数据类型的存储格式，编写算法时当然可以加以充分利用，然而，一旦该数据对象的格式改变了，那么程序代码也必须做相应改动。很多软件设计人员都发现，把数据对象的表示隐藏起来，不为用户所知，反倒是更好的设计策略。这样的话，用户只能通过提供的函数接口处理数据对象。只要某操作所提供的用户接口不变，设计者就可以修改表示方法，而用户并不需要修改代码。

定义 抽象数据类型（ADT）中的数据对象和数据操作的规范声明与数据对象的表示和数据操作的实现相互分离。 □

一些程序设计语言提供明显的机制支持区分规范声明和实现，例如，Ada 语言中的 `package` 和 C++ 语言中的类都提供这种支持，可供程序员直接用来定义抽象数据类型。尽管 C 语言并未明显地提供实现 ADT 的机制，我们还是可以利用 C 语言的现有机制构造类似的数据类型，信心十足地定义 ADT。

那么，ADT 之中数据操作的规范声明与数据操作的实现，其差别究竟是什么？规范声明包括所有函数的名称，它们的参量类型，以及返回结果的类型，还应包括函数的功能描述，但不涉及内部表示和实现细节。这样的需求界定及为重要，也就是说，抽象数据类型应独立于实现。数据类型中的函数还可以进一步分成以下类别:

(1) **创建函数/构造函数**：这类函数为特定类型创建新实例。

(2) **变换函数**：这类函数也为特定类型创建实例，但通常使用一个或多个其它实例。变换函数与构造函数的差别可以通过后续例子进一步澄清。

(3) **观察函数/报告函数**：这类函数提供数据类型的实例信息，但不修改实例。

常用 ADT 至少要包括以上类别中的一种函数。

本书全书不断强调规范声明和实现的区别。为方便读者理解两者之间的差异，我们从一个 ADT 的定义开始。这个定义有助于读者理解问题的实质，这里我们无需讨论数据对象的表示和数据操作实现细节。在 ADT 的定义清楚无误之后，我们才接着讨论数据对象的表示和操作的具体实现，这两方面内容是数据结构的核心。以下引出 ADT 的记法。

例 1.5 (抽象数据类型 NaturalNumber) 这是第一个 ADT 例子，我们要花些篇幅解释记法。ADT 1.1 定义了自然数的抽象数据类型 NaturalNumber。

```
ADT NaturalNumber
  数据对象：有序整数数列，范围从0到机器能够表示的最大整数(INT_MAX)
  成员函数：
    以下 x,y ∈ NaturalNumber; TRUE, FALSE ∈ Boolean,
    且 +, -, <, == 是整数类型的操作符。
    NaturalNumber Zero()          ::=  0
    Boolean IsZero(x)             ::=  if (x) return FALSE
                                       else return TRUE
    Boolean Equal(x, y)           ::=  if (x==y) return TRUE
                                       else return FALSE
    NaturalNumber Successor(x)    ::=  if (x==INT_MAX) return x
                                       else return x + 1
    NaturalNumber Add(x, y)       ::=  if ((x+y)<INT_MAX) return x + y
                                       else return INT_MAX
    NaturalNumber Subtract(x, y) ::=  if (x<y) return 0
                                       else return x - y
end NaturalNumber
```

ADT 1.1: 抽象数据类型 NaturalNumber

这个自然数抽象数据类型，从定义关键字 **ADT** 开始，主要内容分两节：数据对象和成员函数。数据对象实际正是计算机系统中的整数类型，但此时并未显式指明整数的表示。成员函数的定义要复杂一些。首先，定义中用符号 x，y 记 NaturalNumber 的两个元素，再用 TRUE，FALSE 记 Boolean 类型元素。接着定义了整数的加、减、相等、小于函数。这个例子说明，要定义新的数据类型，我们可能会用到其它数据类型的操作。每个函数的形式为：返回结果类型位于函数名称的左边，定义部分位于右边，中间是符号 ::=，意思是“定义为”。

第一个函数的名称是 Zero，无参量，返回自然数零，是构造函数。函数 Successor(x) 返回自然数序列中 x 的下一个，这个函数是变换函数的例子。注意，如果数列中不再有后继元素了，就是说，x 本身就是 INT_MAX 的话，那么这个函数返回 INT_MAX。有些程序员这时更倾向于返回出错信息，也是可以的。其它两个变换函数是 Add 和 Subtract，当然可以在上述越界情形返回出错信息，但我们选择返回 NaturalNumber 中的一个元素。 □

本书后续的抽象数据类型定义，将遵循 ADT 1.1 给出的形式，不过后续 ADT 中的函数定义形式不一定是 C 语言语句，有可能会用结构化的自然语言（英语）解释函数的功能，原因在于，引入 ADT 的目的是隐藏实现细节，因而在 ADT 的定义中，无需拘泥于 C 语言。有时，ADT 中的成员函数与相应的 C 语言函数（具体实现）会有差异，甚至参数个数也不相同，为避免混淆，ADT 中的成员函数都以大写字母开头，而 C 语言函数都以小写字母开头。

习题

参照 ADT 1.1，写出以下抽象数据类型。

1. 为自然数ADT 加上如下成员函数：Predecessor, IsGreater, Multiply, Divide。
   (前驱、大于、乘法、除法)
2. 构造集合 ADT Set，遵循标准数学定义，成员函数包括：Create, Insert, Remove, IsIn, Union, IIntersection, Difference。
   (创建、插入、删除、属于、并，交，差)
3. 构造 ADT Bag，可以有重复元素的集合。至少包括成员函数：Create, Insert, Remove, IsIn。
   (创建、插入、删除、属于)
4. 构造 ADT Boolean，至少包括成员函数：And, Or, Not, Xor, Equivalent, Implies。
   (与、或、非、异或、等价、蕴含)

§1.5 性能分析

本书的一个主要目标是教会读者判断程序性能的优劣。判断程序性能优劣的标准很多，至少包括如下几点：

(1) 程序是否符合任务的规范说明。

(2) 程序是否正确。

(3) 是否有配套文档，说明程序的用法和原理。

(4) 程序是否根据逻辑关系分解成能有效执行的函数。

(5) 程序代码是否易读。

以上判据至关重要，对构建大规模程序系统，显得更为关键；然而，若要指出如何达到上述标准却并非易事。我们只能说，以上判据可以指导程序员遵循良好的习惯和风格，而且这样做有益于系统开发，能使开发过程按良好的步调推进。程序员必须在实践中不断摸索，从自身的经验中提炼出优秀的编程技能，培养出良好的编程风格。本书后续内容包括大量实例，我们希望这些内容有助于读者达到期望的目标。除了上述一般性判据之外，以下两条判据更加具体：

(6) 程序是否能够高效使用主存和辅存。

(7) 程序的运行时间是否令人满意。

这最后两条判据用来评价程序的性能，可以分成两方面讨论。第一方面的性能估计与机器无关的空间代价和时间代价，称为*性能分析*，其研究内容是计算机科学的一个重要分支，属*复杂性理论*研究的核心问题。第二方面称为*性能度量*，即获取程序在真实环境的实际运行时间。通过性能度量能够获得程序运行的时间，有助于发现耗费时间较多的程序片段。本节用来专门讨论性能分析，下节内容讨论性能度量。以下首先给出空间复杂度与时间复杂度的定义。

定义 *空间复杂度*是程序运行所需的存储空间；*时间复杂度*是程序的运行时间。 □

§1.5.1 空间复杂度

程序运行所需空间包括两部分：

(1) **定长空间需求**：即与程序输入、输出无关的空间，包括指令空间（存储代码的空间）、简单变量的存储空间、定长结构变量（如结构体）的存储空间、常量存储空间。

(2) **变长空间需求**：即与求解的问题实例相关的结构化变量所占空间大小。如果是递归程序，还应加上递归时所需工作空间的大小。给定问题实例 I，程序 P 的变长空间记为 $S_P(I)$。$S_P(I)$ 通常为自变量定义在 I 的*特征*空间之上的一个数学函数。这些特征常常表现为关于 I 的输入、输出个数、长度、或取值。例如，若输入是长度为 n 的数组，则 n 是一个实例特征。如果 n 是唯一的实例特征，那么 $S_P(I)$ 可以具体化为 $S_P(n)$。

任何程序所需空间总量是：

$$S(P) = c + S_P(I)$$

其中的 c 表示定长空间需求。分析一个程序的空间复杂度，通常仅考察变长空间需求；分析多个程序的空间复杂度也是这样。下面看几个例子。

例 1.6 函数 abc (见程序 1.10) 的输入是三个简单变量，输出返回一个简单变量。根据以上分类，这个函数只有定长空间需求，因此，$S_{\text{abc}}(I) = 0$。 □

```
float abc(float a, float b, float c)
{
  return a+b+b*c+(a+b-c)/(a+b)+4.00;
}
```

程序 1.10 简单算术函数

例 1.7 程序 1.11 的输出只有一个简单变量，但输入比上例增加了一个数组，这里，变长空间需求依赖于数组参量是如何传给函数的。不同语言的处理各不相同，对于 Pascal 语言，数组参量是按值传送，就是说，这个函数在执行时，整个数组被复制到函数的临时存储空间。对应这样一类语言，程序 1.11 的变长空间需求是 $S_{\text{sum}}(I) = S_{\text{sum}}(n) = n$，$n$ 是数组长度。C 语言的参数传递方法虽然也是传值调用，但如果传递的是数组参量，C 只传递数组第一个元素的首地址，并不复制整个数组，因而，$S_{\text{sum}}(n) = 0$。 □

```
float sum(float list[], int n)
{
  float tempsum = 0;
  int i;
  for (i = 0; i< n; i++)
    tempsum += list[i];
  return tempsum;
}
```

程序 1.11 数组求和的循环实现

例 1.8 程序 1.12 也是增加了一个数组，不过这个求和程序是递归程序，编译器要为每次递归调用保存参量、局部变量、返回地址。

```
float sum(float list[], int n)
{
  if (n) return rsum(list, n-1) + list[n-1];
  return 0;
}
```

程序 1.12 数组求和的递归实现

本例中，一次递归调用所需的空间是两个参量的字节数加上返回地址的字节数。假定整数和指针所需字节数都是 4，图 1.1 示出每次递归调用所需的字节数。

类别	名称	字节数
参量：数组指针	`list[]`	4
参量：整数	n	4
返回地址：(内部使用)		4
每次递归所需空间总和		12

图 1.1 程序 1.12 每次递归调用所需空间

如果数组的成员个数是 $n =$ `MAX_SIZE`，那么变长空间总和是 $S_{\text{rsum}}(\texttt{MAX\_SIZE}) = 12 \times \texttt{MAX\_SIZE}$。当 `MAX_SIZE` 取1000 时，这个递归程序的变长空间需求为 $12 \times 1000 = 12000$ 字节。比较求和程序的循环实现和递归实现，前者无变长空间需求，而后者的开销要大许多。 □

习题

1. 计算 §1.3 习题 7 求解阶乘的循环函数和递归函数的空间复杂度。
2. 计算 §1.3 习题 8 求解 Fibonacci 数列的循环函数和递归函数的空间复杂度。
3. 计算 §1.3 习题 9 求解二项式系数的循环函数和递归函数的空间复杂度。
4. 计算 §1.3 习题 5 函数的空间复杂度(鸽笼原理)。
5. 计算 §1.3 习题 12 函数的空间复杂度(幂集问题)。

§1.5.2 时间复杂度

程序 P 的时间开销记为 $T(P)$，是编译时间和运行（执行）时间的总和。编译时间与定长空间需求类似，同样与实例特征无关；而且，程序经检验确定其正确性之后，编译结果可不断执行。所以时间复杂度仅考虑程序的执行时间 T_P。

准确计算 T_P 需要详细考察编译器的工作属性，即程序源代码到目标代码的翻译过程。我们考虑一个简单程序，假定只做加、减运算，令 n 是实例特征，T_P 可用下式表示：

$$T_P(n) = c_a\,\mathrm{ADD}(n) + c_s\,\mathrm{SUB}(n) + C_l\,\mathrm{LDA}(n) + C_{st}\,\mathrm{STA}(n),$$

其中 $\mathrm{ADD}(n), \mathrm{SUB}(n), \mathrm{LDA}(n), \mathrm{STA}(n)$ 分别是加、减、取、存操作关于实例特征 n 所需执行的次数，c_a, c_s, c_l, c_{st} 分别对应各个操作一次执行的时间常量。

如此繁琐的估计，实在是太过份了，更好的做法是直接用系统时钟测量程序的运行时间，后续内容留有篇幅专门介绍这种方法。现在介绍的方法，是统计程序执行时所需各操作的次数。这种方法的计数结果与机器无关，但首先要把程序分成独立的程序步。

定义 *程序步*是与实例特征无关的、根据语法或语义划分的程序片段。 □

注意，程序步的计算量随表示的不同而不同。例如，

```
a = 2;
```

是一个程序步，

```
a = 2*b + 3*c/d - e + f/g/a/b/c;
```

是另一个程序步，前者简单，所需计算时间较少，后者复杂，所需计算时间较多。两者的共同点在于，它们都被看作一个程序步，且所需计算时间与实例特征无关。

下面我们在程序中设一个全局变量 count，用来累计程序或函数的程序步，count 的初值设为 0，语句插入在累加可执行语句次数的地方。

例 1.9 (数表求和，循环实现) 本例统计求和程序 1.11 的程序步个数，程序 1.13 给出 count 语句的位置。注意，只有可执行语句需要计数，其它如程序首部与第二句变量声明不在计数之列。

```
float sum(float list[], int n)
{
  float tempsum = 0; count++; /* for assignment */
  int i;
  for (i = 0; i< n; i++) {
    count++; /* for the for loop */
    tempsum += list[i]; count++; /* for assignment */
  }
  count++; /* last execution of for */
  count++; /* for return */
  retur tempsum;
}
```

程序 1.13 程序 1.11 加入 count 语句

由于我们的目的是获取计数值，因此，程序 1.13 中那些可执行语句可以去掉，这样得到简化的程序 1.14，使得计数的算术过程更加清晰。程序 1.14 中 count 的初值是 0，最后结果是 $2n+3$，所以求和程序的程序步数目为 $2n+3$。 □

```
float sum(float list[], int n)
{
  float tempsum = 0;
  int i;
  for (i = 0; i < n; i++)
    count += 2;
  count += 3;
  return 0;
}
```

程序 1.14 程序 1.13 的简化

例 1.10 (数表求和，递归实现) 本例统计数表求和递归程序的程序步数目。程序 1.15 在原来的程序 1.12 加入 count 语句。

```
float rsum(float list[], int n)
{
  count++; /*for if conditoinal */
  if (n) {
    count++; /* for return and rsum invocation */
    reutrn rsum(list, n-1) + list[n-1];
  }
  count++;
  return list[0];
}
```

程序 1.15 程序 1.12 加入 count 语句

我们先确定 $n = 0$ 时的边界条件，在程序 1.15 中，当 $n = 0$，只有 **if** 条件语句和第二条 **return** 语句执行，因此程序步为 2。当 $n > 0$，**if** 条件语句和第一条 **return** 语句执行，因此，当 $n > 0$，每次递归调用为计数贡献 2。最后，由于共有 n 次递归调用，加上 $n = 0$ 的一次，这个程序的程序步总数是 $2n + 2$。

这个递归程序的程序步比相应循环程序的程序步少，初看令人惊奇，但不要忘记，程序步的计数仅告诉我们程序以多少程序步执行，并未告诉我们每一程序步的执行时间。实际上，递归程序的执行，一般而言，要比循环程序慢，递归程序虽然程序步少于循环程序，但花的时间却要多一些。 □

例 1.11 (矩阵加法) 程序 1.16 是两维数组求和，我们要统计这个程序的程序步数目。数组 a 和数组 b 相加，结果存入数组 c，每个数组的维数都是 rows × rows。程序 1.17 在 add 函数中加入程序步计数。和上面几个例子一样，程序步计数同样是关于输入量的长度，本例中是 rows 和 cols。为了让程序更易读，我们把同一个循环体中的 count 语句合在一起，给出了形式更简单的程序 1.18。

在程序 1.18 中，count 的初值是 0，最后结果是 $2\text{rows} \cdot \text{cols} + 2\text{rows} + 1$。根据这个结果，如果矩阵的行数比列数大很多，那么应把矩阵转置。 □

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
         int c[][MAX_SIZE], int rows, int cols)
{
  int i, j;
  for (i=0; i<rows; i++)
    for (j=0; j<cols; j++)
      c[i][j] = a[i][j] + b[i][j];
}
```

程序 1.16 矩阵加法

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
         int c[][MAX_SIZE], int rows, int cols)
{
  int i, j;
  for (i=0; i<rows; i++) {
    count++; /* for i for loop */
    for (j=0; j<cols; j++) {
      count++; /* for j for loop */
      c[i][j] = a[i][j] + b[i][j];
      count++; /* for assignment statement */
    }
    count++; /* last time of j for loop */
  }
  count++; /* last time of i for loop */
}
```

程序 1.17 矩阵加法加入 count 语句

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
         int c[][MAX_SIZE], int rows, int cols)
{
  int i, j;
  for (i=0; i<rows; i++) {
    for (j=0; j<cols; j++)
      count +=2;
    count+=2;
  }
  count++;
}
```

程序 1.18 程序 1.17 的简化

上面给出的例子，我们把 count 语句嵌入函数之中，这样的好处是，实际运行以上程序，能得到关于各种实例特征的程序步数目。下面我们介绍表方法，同样可以获得程序步数目，这种方法称为“*程序步/执行*”(stps/execution)，或简称为 s/e。接下来，我们用这种方法找出每条语句的程序步计数，这个计数称为*频率*。不可执行语句的频率是 0，把 s/e 与频率相乘，得到每条语句的程序步计数，把每条语句的程序步计数加起来，结果就是整个函数的程序步计数。乍一看，这种做法挺麻烦，其实并非如此。我们用表方法重做上面三个例子。

例 1.12 (数表求和，循环实现) 图 1.2 是程序 1.11 的程序步计数表。造表的过程是这样的。首先填写每条语句的 s/e 栏，然后填频率栏。第 5 行的 **for** 循环语句略复杂些，由于从 0 开始，i 在等于 n 时结束，所以频率是 $n+1$，而第 6 行的循环体语句只执行 n 次，i==n 时不执行。得到每条语句的程序步个数之后，整个函数的程序步数目就是它们的总和。 □

语句	s/e	频率	程序步个数
`float sum(float list[], int n)`	0	0	0
{	0	0	0
`  float tempsum = 0;`	1	1	1
`  int i;`	0	0	0
`  for (i = 0; i< n; i++)`	1	$n+1$	$n+1$
`    tempsum += list[i];`	1	n	n
`  return tempsum;`	1	0	1
}	0	0	0
程序步总数			$2n+3$

图 1.2 程序 1.11 的程序步计数表

例 1.13 (数表求和，递归实现) 图 1.3 是程序 1.12 的程序步计数表。 □

语句	s/e	频率	程序步个数
`float sum(float list[], int n)`	0	0	0
{	0	0	0
`  if (n)`	1	$n+1$	$n+1$
`    return rsum(list, n-1) + list[n-1];`	1	n	n
`  return 0;`	1	1	1
}	0	0	0
程序步总数			$2n+2$

图 1.3 程序 1.12 的程序步计数表

例 1.14 (矩阵加法) 图 1.4 是矩阵加法函数的程序步计数表。 □

语句	s/e	频率	程序步个数
`void add(int a[][MAX_SIZE] ... )`	0	0	0
{	0	0	0
`  int i, j;`	0	0	0
`  for (i=0; i<rows; i++)`	1	$rows+1$	$rows+1$
`    for (j=0; j<cols; j++)`	1	$rows\cdot(cols+1)$	$rows\cdot cols+rows$
`      c[i][j] = a[i][j] + b[i][j];`	1	$rows\cdot cols$	$rows\cdot cols$
}	0	0	0
程序步总数			$2rows\cdot cols+2rows+1$

图 1.4 矩阵加法的程序步计数表

小结

程序的时间复杂度度量主要由实现程序功能的程序步确定。程序步数目本身也是实例特征的数学函数。每种指定的实例可以取多种特征（例如，输入量的个数、输出量的个数、输入/输出的量值大小，等等），程序步数目由这些特征的某子集计算而得。通常，我们在特征中选取意义重要的个体。例如，如果我们关心计算（运行）时间（即时间复杂度）随输入个数增加的规律，那么，程序步数目的计算仅仅是输入个数的数学函数。再如，如果我们关心计算时间随输入量值变化的规律，那么，程序步数目的计算仅仅是输入量值的数学函数。所以，在统计程序步之前，事先必须确定问题实例特征的具体细节，它将成为程序步计算表达式中的自变量。在统计求和函数 `sum` 的程序步时，时间复杂度测量是关于 n 的数学函数，n 是所加数据元素的个数；计算 `add` 的程序步，用到的特征是待加矩阵的行数和列数。

只有在相应特征（$n, m, p, q, r, \ldots$）选定后，才可以定义究竟什么是程序步。程序步是与特征（$n, m, p, q, r, \ldots$）无关的计算单位。所以，10 次加法是一个程序步；100 次乘法也可以是一个程序步。但 n 次加法却不是一个程序步，同理，$m/2$ 次加法不是一个程序步，$p+q$ 次减法也不是。

以上讨论的计算复杂度例子都相当简单，由初等数学函数确定，依据足够简单的实例特征，例如数表的长度、矩阵的行数和列数。然而，多数程序的时间复杂度估计并非如此简单，如果仅考虑输入、输出的个数，或其它容易定义的特征，是远远不够的。我们考虑程序 1.7 的顺序表查找函数 `binsearch`，要估计程序步，表长 n 是一个自然特征。我们选这个特征的目的，是希望考察程序随 n 变化的规律，但是，这个特征是不充分的。对同一个 n，如果待查元素 `searchnum` 在表中的位置不同，程序的程序步数目就不相同。当遇到类似情形，即选定参数不足以唯一确定程序步数目时，我们只好把程序步细化为三类：最优情形、最差情形、平均情形，通过这种细分排除困难。

对于给定的实例参数，*最优情形*程序步数目是程序执行的最少程序步数目；*最差情形*程序步数目是程序执行的最大程序步数目；*平均情形*程序步数目是程序执行的平均程序步数目。

习题

1. 在 §1.3 习题 2 (多项式的 Horner 法求值)中插入计数语句，写出程序步总数的公式。
2. 在 §1.3 习题 3 (真值表)中插入计数语句，写出程序步总数的公式。
3. 在 §1.3 习题 4 中插入计数语句，写出程序步总数的公式。
4. (a) 在程序 1.19 中插入计数语句。
   (b) 简化上小题结果，删除原可执行语句。
   (c) 给出程序结束时的 `count` 值。
   (d) 写出这个程序计数表。

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
         int c[][MAX_SIZE], int rows, int cols)
{
  int i, j;
  for (i=0; i<rows; i++) {
    for (j=0; j<cols; j++)
      printf("%d", matrix[i][j]);
    printf("\n");
  }
}
```

程序 1.19 打印矩阵

5. 对程序 1.20，重复习题 4 的方法。

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
         int c[][MAX_SIZE], int rows, int cols)
{
  int i, j, k;
  for (i=0; i<MAX_SIZE; i++)
    for (j=0; j<MAX_SIZE; j++) {
      c[i][j] = 0;
      for (k=0; k<MAX_SIZE; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
}
```

程序 1.20 矩阵乘法函数

6. 对程序 1.21，重复习题 4 的方法。

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
         int c[][MAX_SIZE], int rowsa, int colsb, int colsa)
{
  int i, j, k;
  for (i=0; i<rowsa; i++)
    for (j=0; j<colsb; j++) {
      c[i][j] = 0;
      for (k=0; k<colsa; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
}
```

程序 1.21 矩阵点乘函数

7. 对程序 1.22，重复习题 4 的方法。

```
void transpose(int a[][MAX_SIZE])
{
  int i, j, temp;
  for (i=0; i<MAX_SIZE; i++)
    for (j=i+1; j<MAX_SIZE; j++)
      SWAP(a[i][j], a[j][i], temp);
}
```

程序 1.22 矩阵转置函数

§1.5.3 渐近记号（O, Ω, Θ）

统计程序步的动机，是比较实现同一功能的两个程序的时间复杂度，预测实例特征变化时程序的运行时间随之变化的规律。

有证据表明，企盼算出程序步的精确值，无论是最差情形还是平均情形，都可能异乎寻常地困难。由于程序步概念本身的不精确性，花费大量精力计算其精确值，实际上并无可取之处。（例如：$x = y$ 和 $x = y + z + (x/y) + (x * y * z - x/z)$ 都是一个程序步。）正是由于程序步意义的不精确，即使用其精确值作比较，结论也不准确，因此没有多大用处。只有在程序步数目相差很大时，例如 $3n + 3$ 比 $100n + 10$，相比才有意义。就是说，程序步为 $3n + 3$ 的程序，比之程序步为 $100n + 10$ 的程序，所需运行时间总要少一些。即使这是事实，程序步数目也不需要精确到 $100n + 10$，用于比较的程序步数目，即使是 $80n$，或 $85n$，或 $75n$ 左右，已足以达到同样的比较结果。

对多数情形，能够给出如下论断就可以了，如 $c_1 n \le T_P(n) \le c_2 n^2$ 或 $T_Q(m, n) = c_1 n + c_2 m$，其中 c_1, c_2 是非负常量。原因在于，如果有两个程序，一个复杂度是 $c_1 n^2 + c_2 n$，另一个复杂度是 $c_3 n$，那么，复杂度为 $c_3 n$ 的程序，比之复杂度为 $c_1 n^2 + c_2 n$ 的程序，当 n 充分大时，会运行得更快。但是，当 n 较小时，无论哪个程序都可能比另一个运行得更快些（取决于常量 c_1, c_2, c_3 的取值）。如果 $c_1 = 1, c_2 = 2, c_3 = 100$，那么，当 $n \le 98$ 时，$c_1 n^2 + c_2 n \le c_3 n$；当 $n > 98$ 时，$c_1 n^2 + c_2 n > c_3 n$。如果 $c_1 = 1, c_2 = 2, c_3 = 1000$，那么，当 $n \le 998$ 时，$c_1 n^2 + c_2 n \le c_3 n$。

无论 c_1, c_2, c_3 如何取值，总存在一个 n 值，之后，复杂度为 $c_3 n$ 的程序总会比复杂度为 $c_1 n^2 + c_2 n$ 的程序运行得更快。这个 n 值称为*失衡点*(break even point)。如果失衡点是0，那么，复杂度为 $c_3 n$ 的程序运行得总是更快(或者至少一样快)。失衡点的精确值不能由解析法获得，只有程序的实际运行才能确定。一旦知道了一定存在一个失衡点，那么，形式地得到程序的复杂度 $c_1 n^2 + c_2 n$ 和 $c_3 n$，其中 c_1, c_2, c_3 是任意常数，我们所需的信息已经足够了，不再需要刻意找出 c_1, c_2, c_3 的精确值。

上述讨论是本节内容的动机，现在介绍一些术语，使我们可以推出有意义的（不是精确的）论断，刻划程序的时间复杂度和空间复杂度。在本章后续内容，f 和 g 都是非负的数学函数。

定义 (大 O 记号) $f(n) = O(g(n))$（读作“$f(n)$ 是 $g(n)$ 的大 O”）当且仅当存在正常量 c 和 n_0，使当 $n \ge n_0$ 时，$f(n) \le cg(n)$。 □

例 1.15 $3n+2=O(n)$ 因为对所有 $n\geq 2$ 有 $3n+2\leq 4n$。$3n+3=O(n)$ 因为对所有 $n\geq 3$ 有 $3n+2\leq 4n$。$100n+6=O(n)$ 因为对所有 $n\geq 10$ 有 $100n+6\leq 101n$。$10n^2+4n+2=O(n^2)$ 因为对所有 $n\geq 5$ 有 $10n^2+4n+2\leq 11n^2$。$1000n^2+100n-6=O(n^2)$ 因为对所有 $n\geq 100$ 有 $1000n^2+100n-6\leq 1001n^2$。$6\cdot 2^n+n^2=O(2^n)$ 因为对所有 $n\geq 4$ 有 $6\cdot 2^n+n^2\leq 7\cdot 2^n$。$3n+3=O(n^2)$ 因为对所有 $n\geq 2$ 有 $3n+3\leq 3n^2$。$10n^2+4n+2=O(n^4)$ 因为对所有 $n\geq 2$ 有 $10n^2+4n+2\leq 10n^4$。$3n+2\neq O(1)$ 因为对所有常量 c 和 $n\geq n_0$ 都不可能使$3n+2$ 小于常量 c。$10n^2+4n+2\neq O(n)$。 □

我们用 $O(1)$ 表恒定的时间复杂度。$O(n)$ 称为线性的，$O(n^2)$ 称为 2 次的，$O(n^3)$ 称为 3 次的，$O(2^n)$ 称为指数的。如果一个算法的时间复杂度是 $O(\log n)$，那么对充分大的 n，该算法比复杂度为 $O(n)$ 的算法运行更快。同理，$O(n\log n)$ 比 $O(n^2)$ 快，但不如 $O(n)$ 快。这七种计算时间，$O(1), O(\log n), O(n), O(n\log n), O(n^2), O(n^3)$, 和 $O(2^n)$，将在本书不断出现。

上例说明，语句 $f(n)=O(g(n))$ 仅指出 $g(n)$ 是当 $n\geq n_0$ 时，$f(n)$ 所能取得的上界值，而并未涉及这个上界有多好（接近）。注意，我们甚至可以写 $n=O(n^2), n=O(n^{2.5}), n=O(n^3), n=O(2^n)$，等等。然而，为使语句 $f(n)=O(g(n))$ 提供尽量多的信息，$g(n)$ 应该是关于 n 的、满足关系 $f(n)=O(g(n))$ 的尽量小的数学函数。所以，尽管 $3n+3=O(n^2)$ 也不算错，但今后绝不这样写，我们总是写 $3n+3=O(n)$。

根据大 O 记号的定义，我们明确指出，$f(n)=O(g(n))$ 与 $O(g(n))=f(n)$ 截然不同。实际上，$O(g(n))=f(n)$ 毫无意义。这里用的 $=$ 和常用的“等于”记号形式相同，却意义各异。为区分两者的意义，$=$ 在这里的读作“是”，而不应读作“等于”。

如果 $f(n)$ 是关于 n 的多项式数学函数，以下定理 1.2 有助于计算 $f(n)$ 的阶数，这个阶数就是式 $f(n)=O(g(n))$ 中的 $g(n)$。

定理 1.2 如果 $f(n)=a_m n^m+\cdots a_1 n+a_0$，那么 $f(n)=O(n^m)$。

证明 当 $n\geq 1$，有

$$\begin{aligned} f(n) &\leq \sum_{i=0}^{m}|a_i|n^i \\ &\leq n^m\sum_{i=0}^{m}|a_i|n^{i-m} \\ &\leq n^m\sum_{i=0}^{m}|a_i|. \end{aligned}$$

所以，$f(n)=O(n^m)$。 □

定义 (大 Ω 记号) $f(n)=\Omega(g(n))$（读作“$f(n)$ 是 $g(n)$ 的大 Ω”）当且仅当存在正常量 c 和 n_0，使当 $n\geq n_0$ 时，$f(n)\geq cg(n)$。 □

例 1.16 $3n+2=\Omega(n)$ 因为对所有 $n\geq 1$ 有 $3n+2\geq 3n$。（实际上，不等式对 $n\geq 0$ 也成立，但根据 Ω 定义，需要 $n_0>0$。）$3n+3=\Omega(n)$ 因为对所有 $n\geq 1$ 有 $3n+2\geq 3n$。$100n+6=\Omega(n)$ 因为对所有 $n\geq 1$ 有 $100n+6\geq 100n$。$10n^2+4n+2=\Omega(n^2)$ 因为对所有 $n\geq 1$ 有 $10n^2+4n+2\geq n^2$。$6\cdot 2^n+n^2=\Omega(2^n)$ 因为对所有 $n\geq 1$ 有 $6\cdot 2^n+n^2\geq 2^n$。通过观察，也

有 $3n+3=\Omega(1)$；$10n^2+4n+2=\Omega(n)$；$10n^2+4n+2=\Omega(1)$；$6\cdot 2^n+n^2=\Omega(n^{100})$；$6\cdot 2^n+n^2=\Omega(n^{50.2})$；$6\cdot 2^n+n^2=\Omega(n^2)$；$6\cdot 2^n+n^2=\Omega(n)$；$6\cdot 2^n+n^2=\Omega(1)$。 □

定理 1.3 如果 $f(n)=a_m n^m+\cdots a_1 n+a_0$，则 $f(n)=\Omega(n^m)$。

证明 留作习题。 □

定义 (大 Θ 记号) $f(n)=\Theta(g(n))$（读作“$f(n)$ 是 $g(n)$ 的大 Θ”）当且仅当存在正常量 c_1, c_2, 和 n_0，使当 $n\geq n_0$ 时，$c_1 g(n)\leq f(n)\leq c_2 g(n)$。 □

例 1.17 $3n+2=\Theta(n)$ 因为对所有 $n\geq 2$ 有 $3n+2\geq 3n$，对所有 $n\geq 2$ 有 $3n+2\leq 4n$，因此 $c_1=3, c_2=4, n_0=2$。$3n+2=\Theta(n)$；$10n^2+4n+2=\Theta(n^2)$；$6\cdot 2^n+n^2=\Theta(2^n)$；还有，$10\log n+4=\Theta(\log n)$；$3n+2=\Theta(1)$；$3n+3=\Theta(n^2)$；$10n^2+4n+2=\Theta(n)$；$10n^2+4n+2=\Theta(1)$；$6\cdot 2^n+n^2=\Theta(n^2)$；$6\cdot 2^n+n^2=\Theta(n^{100})$；$6\cdot 2^n+n^2=\Theta(1)$； □

大 Θ 记号比大 O 记号与大 Ω 记号都精确，$f(n)=\Theta(g(n))$ 当且仅当 $g(n)$ 既是 $f(n)$ 的上界又是 $f(n)$ 的下界。

注意，在以上三个例子中，为了便利，$g(n)$ 的系数全部取为 1，实际使用时通常也是这样。我们几乎永远不会写 $3n+3=O(3n)$，$10=O(100)$，$10n^2+4n+2=\Omega(4n^2)$，$6\cdot 2^n+n^2=\Omega(6\cdot 2^n)$，$6\cdot 2^n+n^2=\Omega(4\cdot 2^n)$，尽管这些写法都不算错。

定理 1.4 如果 $f(n)=a_m n^m+\cdots a_1 n+a_0$，则 $f(n)=\Theta(n^m)$。

证明 留作习题。 □

我们现在重新考察上一节的复杂度分析例子。程序 1.12 中的函数 `sum` 已算出 $T_{\text{sum}}(n)=2n+3$，因而，$T_{\text{sum}}(n)=\Theta(n)$。由于 $T_{\text{rsum}}(n)=2n+2$，所以有 $T_{\text{rsum}}(n)=\Theta(n)$；还有，$T_{\text{add}}(rows,cols)=2rows\cdot cols+2rows+1=\Theta(rows\cdot cols)$。

尽管上一段推导说明 O, Ω, Θ 记号都正确，读者可能还是会问：“如果程序步已经精确地算出来了，那么这些记号又有什么用处呢？”我们的回答是，渐近复杂度（就是用这些记号表示的复杂度）分析要容易得多，而且并不需要计算程序步数目。渐近复杂度分析的方法是，首先确定程序中每条语句（或语句组）的渐近复杂度，然后再把它们加起来。

例 1.18 (矩阵加法的复杂度) 图 1.5 中的表是用表方法构造的，构造方法与图 1.4 的程序步计算表相似。不过，这里并没有列出精确的程序步，而是列出相应的渐近表示，对不可执行语句，填入的计数值是 0。实际上，构造图 1.5 所示表格，要比构造图 1.4 所示表格容易。例如，第 5 行语句，程序步数目是 $rows\cdot(cols+1)$，其计算比渐近复杂度 $\Theta(rows\cdot cols)$ 的计算难度要大很多。只要把每条语句的渐近复杂度加起来，就能得到整个函数的渐近复杂度。甚至还有更简单的做法，由于程序行数是常量(即与实例特征无关)，只要挑出语句行中渐近复杂度的最大值，就是整个函数的渐近复杂度。不管用那张方法，都能得到正确的结果 $\Theta(rows\cdot cols)$。 □

例 1.19 (折半查找) 我们现在计算程序 1.7 中折半查找函数 `binsearch` 的时间复杂度。我们用表长 n 作实例特征。**`while`** 语句每次循环耗时 $\Theta(1)$，可以证明，**`while`** 语句的循环次数最多是 $\lceil\log_2(n+1)\rceil$。由于是渐近分析，不必用如此精确的最差情形循环次数。每次循环，除了

语句	渐近复杂度
`void add(int a[][MAX_SIZE] ... )`	0
{	0
`  int i, j;`	0
`  for (i=0; i<rows; i++)`	$\Theta(rows)$
`    for (j=0; j<cols; j++)`	$\Theta(rows \cdot cols)$
`      c[i][j] = a[i][j] + b[i][j];`	$\Theta(rows \cdot cols)$
}	0
总计	$\Theta(rows \cdot cols)$

图 1.5 矩阵加法的程序步计数表

最后一次，待查找的表长都会减少一半，也就是说，`right-left+1` 每次都会减少大约一半。所以，最差情形的循环次数是 $\Theta(\log n)$，再考虑到每次循环耗时 $\Theta(1)$，我们得到，`binsearch` 在最差情形的整体复杂度是 $\Theta(\log n)$。注意，最佳情形的复杂度是 $\Theta(1)$，这时，**`while`** 循环在第一次查找就找到了 `searchnum`。 □

例 1.20 (置换) 考虑程序 1.9 中的函数 `perm`。当 $i = n$ 时，耗时 $\Theta(n)$。当 $i < n$ 时，进入 **`else`** 语句，里面的 **`for`** 语句要进入 $n-i+1$ 次，每次循环耗时 $\Theta(n + T_{\text{perm}}(i+1,n))$。所以，当 $i < n$ 时，$T_{\text{perm}}(i,n) = \Theta((n-i+1)(n+T_{\text{perm}}(i+1,n)))$。因为当 $i+1 \le n$ 时，$T_{\text{perm}}(i+1,n)$ 至少是 n，所以对 $i < n$，有 $T_{\text{perm}}(i,n) = \Theta((n-i+1)T_{\text{perm}}(i+1,n))$。求解这个递推关系，得到 $T_{\text{perm}}(i,n) = \Theta(n \cdot n!)$。 □

例 1.21 (魔方) 最后一个复杂度分析例子取自趣味数学的魔方问题。*魔方*是 $n \times n$ 的矩阵，每个单元取整数值，范围从 1 到 n^2，要求每行、每列，以及两条主对角线的和都相等。图 1.6 是 $n = 5$ 的魔方，相等的和数是 65。

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

图 1.6 5 阶魔方

Coxeter 提出如下生成奇数阶(n是奇数)魔方的方法：

开始时，在魔方第一行的中间一格放 1。然后重复以下步骤：移动到左上一格，把当前的数加 1 放在这个位置。如果移动时超出魔方范围，则想象与当前状态完全相同的另一个魔方，(对齐)紧靠在超出的那条边界线上，因而可以继续。如果移动到的格子已经放置过数字，则从这个格子的位置向正下方移动一格。直到把所有格子

都放满数字为止。

图 1.6 就是根据 Coxeter 规则生成的。程序 1.23 是 Coxeter 算法的实现。令 n 表示魔方阶数（即程序中的 `size`）。**`if`** 语句做错误检查，耗时 $\Theta(1)$。两个嵌套的 **`for`** 循环复杂度是 $\Theta(n^2)$，下一个 **`for`** 循环每次耗时 $\Theta(1)$，共循环 $\Theta(n^2)$ 次，所以复杂度是 $\Theta(n^2)$。输出魔方的 **`for`** 循环耗时也是 $\Theta(n^2)$。综上所述，整个程序 1.23 的渐近复杂度是 $\Theta(n^2)$。 □

在后续章节程序的复杂度分析中，我们通常只关心复杂度上界，因此要把分析结果限制在大 O 记号。我们这样做的原因是，算法的主流分析结果大多给出 O 记号结果。在后续分析过程，如果得到的复杂度界既是上界也是下界，则可以用大 Θ 记号代替大 O 记号。

习题

1. 证明以下各小题

   (a) $5n^2 - 6n = \Theta(n^2)$

   (b) $n! = O(n^n)$

   (c) $2n^2 + n\log n = \Theta(n^2)$

   (d) $\sum_{i=0}^{n} i^2 = \Theta(n^3)$

   (e) $\sum_{i=0}^{n} i^3 = \Theta(n^4)$

   (f) $n^{2^n} + 6 \cdot 2^n = \Theta(n^{2^n})$

   (g) $n^3 + 10^6 n^2 = \Theta(n^3)$

   (h) $6n^3 / (\log n + 1) = O(n^3)$

   (i) $n^{1.001} + n\log n = \Theta(n^{1.001})$

   (j) 对所有 $k \geq 1$，$n^k + n + n^k \log n = \Theta(n^k \log n)$

   (k) $10n^3 + 15n^4 + 100n^2 2^n = O(n^2 2^n)$

2. 证明以下各小题不正确

   (a) $10n^2 + 9 = O(n)$

   (b) $n^2 \log n = \Theta(n^2)$

   (c) $n^2 / \log n = \Theta(n^2)$

   (d) $n^3 2^n + 6n^2 3^n = O(n^2 2^n)$

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

3. 证明定理 1.3。

4. 证明定理 1.4。

5. 计算程序 1.19 的最差情形复杂度。

```
#include <stdio.h>
#define MAX_SIZE 15             /* maximum size of square */
void main(void)
{ /* construct a magic square, iteratively */
  int square[MAX_SIZE][MAX_SIZE];
  int i, j, row, col;           /* indexes */
  int count;                    /* counter */
  int size;                     /* square size */
  printf("Enter␣the␣size␣of␣the␣square:␣");
  scanf("%d", &size);
  /* check for input errors */
  if (size<1 || size>MAX_SIZE+1) {
    fprintf(stderr, "Error!␣Size␣is␣out␣of␣range\n");
    exit(EXIT_FAILURE);
  }
  if (!(size%2)) {
    fprintf(stderr, "Error!␣Size␣is␣even\n");
    exit(EXIT_FAILURE);
  }
  for (i=0; i<size; i++)
    for (j=0; j<size; j++) square[i][j] = 0;
  square[0][(size-1)/2] = 1;    /* middle of first row */
  i = 0; j = (size-1)/2;        /* i and j are current position */
  for (count=2; count<=size*size; count++) {
    row = (i-1<0) ? (size-1) : (i-1);       /* up */
    col = (j-1<0) ? (size-1) : (j-1);       /* left */
    if (square[row][col]) i = (++i) % size; /* down */
    else { i = row; j = (j-1<0) ? (size-1) : --j; } /*square is unoccupied */
    square[i][j] = count;
  }
  /* output the magic square */
  printf("␣Magic␣Square␣of␣size␣%d␣:␣\n\n", size);
  for (i=0; i<size; i++) {
    for (j=0; j<size; j++) printf("%5d", square[i][j]);
    printf("\n");
  }
  printf("\n\n");
}
```

程序 1.23 魔方程序

6. 计算程序 1.22 的最差情形复杂度。

7. 用不同的 n 比较函数 n^2 和 $20n+4$。确定什么时候第二个函数变得比第一个函数要慢。

8. 参考程序 1.23，编写生成魔方的递归程序。

§1.5.4 实际复杂度

我们知道，程序的时间复杂度是某些实例特征的数学函数，用来揭示该程序本身的时间需求随实例特征变化的规律。复杂度函数还可以用来比较完成同样功能的两个程序 P, Q 的运行效率。假定 P 的复杂度是 $\Theta(n)$，Q 的复杂度是 $\Theta(n^2)$，可以断言，对“充分大的”n，程序 P 比程序 Q 运行得更快。以下论证这个事实。我们注意到，存在某常量 c，当 $n \geq n_1$ 时，程序 P 的实际运行时间上界是 cn；另外，存在某常量 d，当 $n \geq n_2$ 时，程序 Q 的实际运行时间下界是 dn^2。由于当 $n \geq c/d$ 时，$cn \leq dn^2$，所以，只要 $n \geq \max\{n_1, n_2, c/d\}$，程序 P 就会比程序 Q 运行得更快。

读者应重视上述论证中的“充分大”。要在以上两个程序中选一个运行，一定要确定 n 是否真得“充分大”。如果程序 P 实际的运行时间是 $10^6 n$ 毫秒，而程序 Q的运行时间是 n^2 毫秒，并且其它因素相同，那么，当 $n \leq 10^6$ 时，理应选程序 Q。

读者对各数学函数的变化率应有直观印象，请仔细研究图 1.7 和图 1.8。我们看到，函数 2^n 关于 n 增长得非常快。事实上，如果一个程序需执行 2^n 程序步，那么当 $n = 40$ 时，程序步数目约为 1.1×10^{12}，假定一台计算机每秒可执行 10 亿步，那么完成所有这些程序步需要 18.3 秒。当 $n = 50$，在同一台计算机上需要执行 13 天；$n = 60$ 需要约 310.56 年；$n = 100$ 需要约 4×10^{13} 年。所以，具有指数复杂度的程序，仅限于 $n(\leq 40)$ 很小的情形。

$\log_2 n$	n	$n \log_2 n$	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	4096	65536
5	32	160	1024	32768	4294967296

图 1.7 函数值

程序如果具有高阶多项式复杂度，也没有实际用途。假定还是在上述那台计算机上运行程序，若程序步数目是 n^{10}，运行需要 10 秒；$n = 100$ 需要 3,171 年；$n = 1000$ 需要 3.17×10^{13} 年。如果程序的时间复杂度是 n^3 程序步，$n = 1000$ 需要 1 秒；$n = 10000$ 需要 110.67 秒；$n = 100000$ 需要 11.57 天。

图 1.9 列出了在“10 亿次程序步 / 每秒”计算机上，执行复杂度 为 $f(n)$ 的程序所需时间。读者应该了解，目前最快的计算机，执行速度也就是大约每秒 10 亿条指令。据此，在现实世界中，当 $n(> 100)$ 较大时，只有那些时间复杂度小(如 $n, n\log n, n^2, n^3$)的程序是能行的。即使未来建造更快的计算机，情况也基本如此。假定我们有每秒执行 10^{12} 条指令的计算机，那

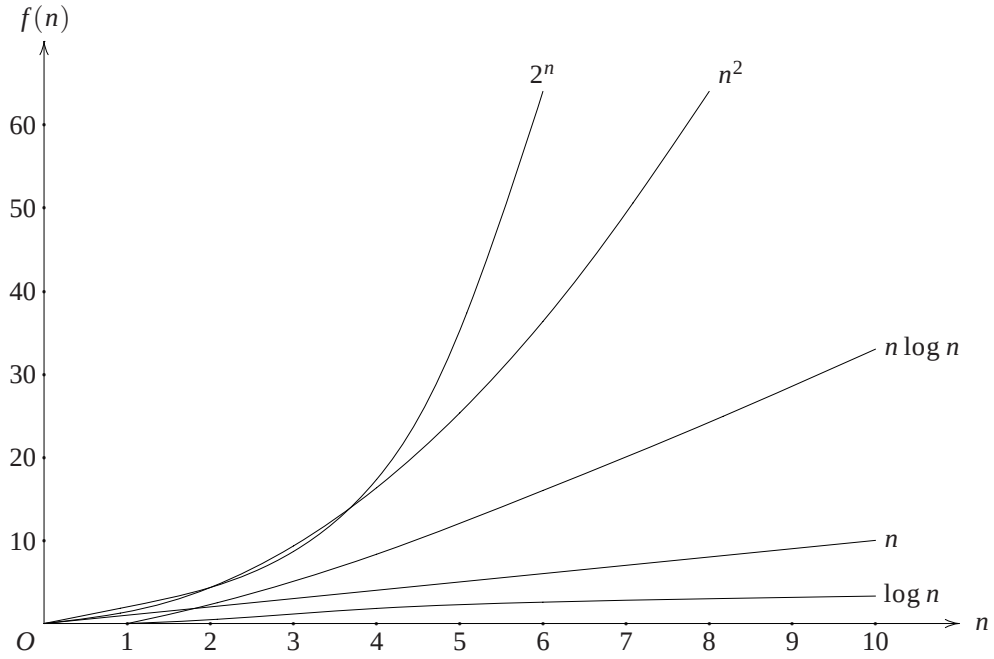

图 1.8 函数值作图

	$f(n)$						
n	n	$n\log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01 μs	.03 μs	.1 μs	1 μs	10 μs	10 s	1 μs
20	.02 μs	.09 μs	.4 μs	8 μs	160 μs	2.84 h	1 ms
30	.03 μs	.15 μs	.9 μs	27 μs	810 μs	6.83 d	1 s
40	.04 μs	.21 μs	1.6 μs	64 μs	2.56 ms	121 d	18 m
50	.05 μs	.28 μs	2.5 μs	125 μs	6.25 ms	3.1 y	13 d
100	.10 μs	.66 μs	10 μs	1 ms	100 ms	3171 y	4×10^{13} y
10^3	1 μs	9.96 μs	1 ms	1 s	16.67 m	3.17×10^{13} y	32×10^{283} y
10^4	10 μs	130 μs	100 ms	16.67 m	115.7 d	3.17×10^{23} y	
10^5	100 μs	1.66 ms	10 s	11.57 d	3171 y	3.17×10^{33} y	
10^6	1 ms	19.92 ms	16.67 m	31.71 y	3.17×10^{7} y	3.17×10^{43} y	

μs = 微秒($=10^{-6}$ 秒)，ms = 毫秒($=10^{-3}$ 秒)，
s = 秒，m = 分钟，h = 小时，d = 天，y = 年。

图 1.9 在“10 亿次程序步 / 每秒”计算机上的程序运行时间表

么图 1.9 中的数据只不过减少为千分之一，当 $n = 100$，复杂度为 n^{10} 的程序要运行 3.17 年，复杂度为 2^n 的程序要运行 4×10^{10} 年。

§1.6 性能度量

§1.6.1 定时

性能分析用来评价算法的空间、时间复杂度，是强有力的工具，然而，考察算法在计算机上的实际执行效率，有时也不可或缺，这种需求将我们从分析领地带到测量领地。本节专门讨论测量算法运行时间的方法。

获得时间事件是 C 语言标准库提供的功能之一。头文件 `time.h` 中包含相应的函数声明。C 语言程序可以用两种方法获得时间事件信息，图 1.10 列出了这两种方法的主要差别。

	方法一	方法一
启动定时钟	`start=clock()`	`start=time(NULL)`
停止定时钟	`stop=clock()`	`stop=time(NULL)`
返回类型	`clock_t`	`time_t`

返回秒数	`duration=`
	`((double)(stop-start))/`
	`CLOCK_PER_SEC`
		`duration=`
		`(double)difftime(stop, start)`

图 1.10 C 语言的事件定时

方法一调用 `clock` 函数，返回处理机内部的绝对时钟周期，这个时钟周期从以前的某时刻开始计时。为算出程序的执行时间，需两次调用 `clock`，然后用结束时的返回值减去开始时的返回值。因为所得结果可以是任意的合法数值类型，所以用强制类型转换把它转成 **`double`** 型。另外，这种表示时钟周期是的内部时钟，还要除以“每秒时钟周期数”才是秒数，在 ANSI C 中，这个每秒时钟周期数是内部常量 `CLOCKS_PER_SEC`。方法一得到的结果非常精确。方法二的优点是不涉及每秒时钟周期数，用法更简单一些。

方法二用 `time` 函数，返回值是按秒计时的时间，类型是内部类型 `time_t`。与调用 `clock` 不同，调用 `time` 需要传入一个参量，它指定返回时间的存储单元，我们这里不需要保留这个返回值，所以参量是 `NULL`。与方法一相同，在开始计算时间时调用一次 `time_t`，结束时再调用一次，然后把这两个结果传给 `difftime` 函数，它返回测量所需的时间差。由于这个返回类型是 `time_t`，我们在打印输出前把它强制转换为 **`double`**。

例 1.22 (选择排序的最差情形性能) 当待排序数据呈逆序时，选择排序算法运行在最差情形。就是说，开始时数据按降序排列，而排序的目标是得到升序排列的数据。在本例中，我们分别取数组长度 0, 10, 20, 90, 100, 200, . . ., 1000。程序 1.24 是时间测量程序代码（`sort` 函数已在程序 1.4 给出，程序 1.24 包括的头文件 `selectionSort.h` 指得就是这个程序）。

```
#include <stdio.h>
#include <time.h>
#include "SelectionSort.h"
#define MAX_SIZE 1001
void main(void)
{
  int i, n, step=10;
  int a[MAX_SIZE];
  double duration;
  clock_t start;

  /* times for n=0, 10, ..., 100, 200, ..., 1000 */
  printf("␣␣␣␣n␣␣␣time\n");
  for (n=0; n<=1000; n+=step)
    { /* get time for size n */

      /* initialize with worst-case data */
      for (i=0; i<n; i++)
        a[i] = n - i;

      start = clock();
      sort(a, n);
      duration = ((double)(clock()-start))/CLOCK_PER_SEC;
      printf("%6d␣␣␣␣%f\n", n, duration);
      if (n==100) step = 100;
    }
}
```

程序 1.24 第一个选择排序的时间测量程序

程序中用 **for** 语句控制数组长度，每次循环时，都先准备一个逆序数组。在 sort 语句的前后调用 clock。令人惊讶的是，对每个 n，时间间隔输出结果都是 0! 究竟哪里出错了呢？尽管程序 1.24 逻辑上是正确的，但是本例测量的时间间隔太短了。程序 1.24 的测量误差是±1 个时钟周期，只有程序的运行时间远远大于 1 个时钟周期时，测量结果才是正确的。程序 1.25 是比程序 1.24 更精确的选择排序测量程序，程序中，排序所需时间延长到一秒(1000 个时钟周期)。不过这个程序也有不精确之处，例如排序前的数组初始化时间也包括在内了，但是，初始化时间比排序的实际时间要小($O(n)$ 比 $O(n^2)$)。如果初始化时间不容忽略，那么还需测量初始化时间，然后在程序 1.25 的输出结果中减去这段时间。

程序 1.25 的输出结果见图 1.11 和图 1.12。图 1.12 中的曲线与图 1.8 中的 n^2 曲线吻合，说明实际运行结果与分析结果一致。 □

```
#include <stdio.h>
#include <time.h>
#include "SelectionSort.h"
#define MAX_SIZE 1001
void main(void)
{
  int i, n, step=10;
  int a[MAX_SIZE];
  double duration;
  clock_t start;
  long repetitions;

  /* times for n=0, 10, ..., 100, 200, ..., 1000 */
  printf("␣␣␣␣n␣␣␣time\n");
  for (n=0; n<=1000; n+=step)
    {
      /* get time for size n */
      repetitions = 0;
      start = clock();
      do {
        repetitions++;

        /* initialize with worst-case data */
        for (i=0; i<n; i++)
          a[i] = n - i;

        sort(a, n);
      } while (clock()-start<1000);
      /* repeat until enough time has elapsed */

      duration = ((double)(clock()-start))/CLOCK_PER_SEC;
      duration /= repetitions;
      printf("%6d␣␣␣%9d␣␣␣%f\n", n, repetitions, duration);
      if (n==100) step = 100;
    }
}
```

程序 1.25 更精确的选择排序的时间测量程序

n	重复次数	时间
0	8690714	0.000000
10	2370915	0.000000
20	604948	0.000002
30	329505	0.000003
40	205605	0.000005
50	145353	0.000007
60	110206	0.000009
70	85037	0.000012
80	65751	0.000015
90	54012	0.000019
100	44058	0.000023
200	12582	0.000079
300	5780	0.000173
400	3344	0.000299
500	2096	0.000477
600	1516	0.000660
700	1106	0.000904
800	852	0.001174
900	681	0.001468
1000	550	0.001818

图 1.11 选择排序在最差情形的性能(时间单位是秒)

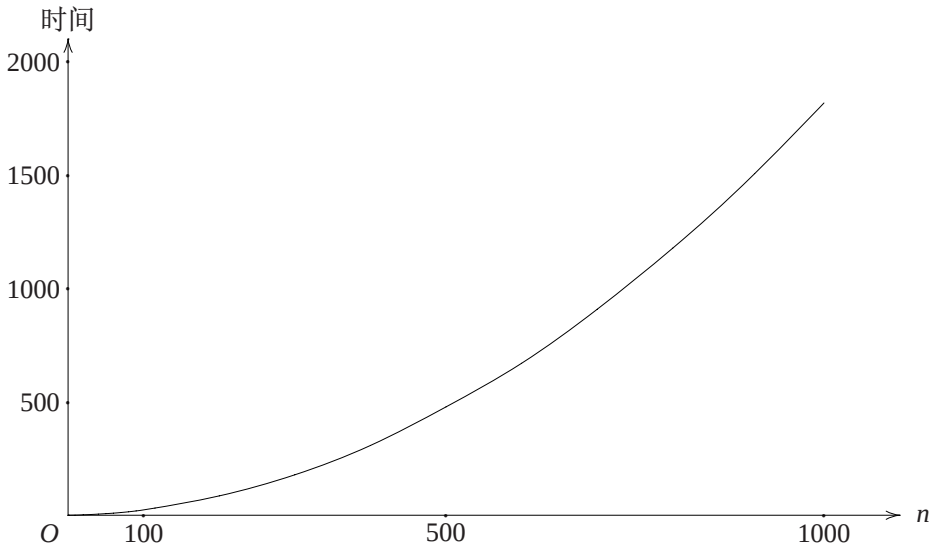

图 1.12 选择排序在最差情形的性能图

§1.6.2 生成测试数据

分析程序的最差情形，需要构造特殊的测试数据，一般来说，这并不容易。对于某些问题，编写程序确实可以生成满足需要的数据，但对于另一些问题，即使借助计算机也很难获得合适的测试数据。这时，我们只能采用如下方法，即首先针对每个实例特征的可能取值集合，生成大量的随机测试数据，然后利用这些数据进行测试，最后把测试结果中的最大值看作最差情形的估计结果。

对于平均情形的时间测量，容易想到的方法是，先利用所有可能的实例特征进行测试，然后取其平均值。不过，一般而言，这种方法并不是万能的。对于顺序查找和折半查找，这种方法确实可行，然而，用这种方法测试排序程序的平均性能根本行不通。假定每个关键字都不同，那么对任意给定的 n，不同的排列个数是 $n!$，显然，要测试所有这些数据是不可能的。

生成测试平均情形性能的数据要比最差情形困难得多，所以，通常的做法是采用随机数测试方法，然后估计平均时间。

用随机数方法测试程序性能，无论最差情形还是平均情形，通常选取的实例特征数据集都要远远小于实际实例特征数据集，因而，我们期望首先分析待测试算法的特点，并据此决定在性能测试时应用特殊的数据类别。这是一类特定的算法分析问题，但我们不在这里深入讨论了。

习题

下列习题都需要构造定时程序。你必须自己选择合适的数组大小，并选择合适的定时语句。把所得结果整理成图、表，最后作出总结。

1. 重做例 1.22。要求测量时间精确到 10%，用原例中的那些 n 值，把测量结果画成关于 n 的函数图像。

2. 比较程序 1.11 与程序 1.12 中，数组求和函数的循环实现与递归实现的最差情形性能。

3. 比较程序 1.7 与程序 1.8 中，折半查找函数的循环实现与递归实现的最差情形性能。

4. (a) 把程序 1.26[1] 中循环实现的顺序查找函数翻译成等价的递归函数。

   (b) 分析函数的最差情形复杂度。

   (c) 测量递归的顺序查找函数的最差情形性能，并和循环函数的最差情形性能相比较。

5. 测试程序 1.16 中 `add` 的最差情形性能。

6. 测试程序 1.20 中 `mult` 的最差情形性能。

[1]原文如此，书中无此程序——译者注。

§1.7 参考文献和选读材料

文献 [1] 是 C 语言入门的好教材。文献 [2] 包括更全面的的软件测试与调试技术。

[1] Narain Gehani *C: An advanced introduction*. Silicon Press, NJ, 1995.

[2] J. Falk C. Kaner and H. Nguyen. *Testing Computer Software*. John Wiley, New York, NY, 2nd edition, 1999.

文献 [3–6] 包含大量程序的渐近复杂度分析结果。

[3] S. Sahni E. Horowitz and S. Rajasekaran. *Fundamentals of Computer Algorithms*. W.H.Freedman and Co., New York, 1998.

[4] Dinesh P. Mehta and Sartaj Sahni. *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.)*. Chapman & Hall / CRC, 2004.

[5] C. Leiserson T. Cormen and R. Rivest. *Introduction to Algorithms*. McGraw-Hill, New York, NY, 2002.

[6] G. Rawlins. *Compared to What: An Introduction to the Analysis of Algorithms*. W.H.Freedman and Co., NY, 1992.