第 2 章 程序设计基本策略与方法

递归、逐步求精、分治是基本的算法（程序）设计策略与方法。许多复杂问题，使用它们都可迎刃而解。这几种策略与方法在后面要经常使用，这里先介绍它们的基本思想，进一步的例子将在后面的章节中见到。做为基础，我们先介绍算法的概念。

§2.1 算法的基本概念

算法是与程序设计和数据结构密切相关的一个问题。简单地讲，算法是解决问题的策略、规则、方法。算法的具体描述形式很多，但计算机程序是对算法的一种精确描述，而且可在计算机上运行（通过适当的编译/解释）。数据结构的操作的实现方法就是个算法问题，但该类问题是针对数据结构的，在给定数据结构上进行的。一般的算法问题是直接面向应用的，它涉及到数据结构的应用，但它的范围比数据结构中的操作更广。

§2.1.1 算法的概念

(一) 什么是算法

一个**算法**是规则的有穷序列，它为解决某一特定类型的问题规定了一个运算过程，并且具有下列特性：

- 有穷性：一个算法必须在执行有穷步之后结束。
- 确定性：算法的每一步必须是确切定义的，对于每种情况，有待执行的每种动作必须严格地和清楚地规定。
- 可行性：算法应该是可行的，这意味着算法中所有有待实现的运算都是基本的，即它们都可由相应的基本计算装置所理解，并可通过有穷次运算即可完成。
- 输入：一个算法有零个或多个输入，它们是在算法所需的初始量或被加工的对象的表示。这些输入取自特写的对象集合。
- 输出：一个算法有一个或多个输出，它们是与输入有特定关系的量。

算法实质上是特定问题的可行的求解方法、规则、步骤。在算法的定义中，并没有规定算法的描述方法，所以它的描述方法可以是任意的，可以用自然语言描述，也可用数学方法描述，也可用某种计算机语言。若用计算机语言描述，就成了**程序**。但一个程

序并不一定满足算法的定义（如某些程序，比如说操作系统，是不会终止的，除非给出了专门的终止信号）。

(二) 什么是“好”的算法

通常，从下列几个方面衡量算法的优劣：

正确性。也称有效性，是指算法能满足具体问题的要求。亦即对任何合法的输入，算法都会得出正确的结果。确认正确性的根本的方法是进行形式化的证明。但对一些较复杂的问题，这是一件相当困难的事。许多计算机科学工作者正致力于这方面的研究，目前尚处于初级阶段，因此，实际中常常用测试的方法验证正确性。测试是指用精心选定的输入（测试用例）去运行算法，检查其结果是否正确。但正如著名的计算机科学家E ·Dijkstra 所说的那样，“测试只能指出有错误，而不能指出不存在错误”。

可读性。指算法被理解的难易程度。人们现在常把算法的可读性放在比较重要的位置，主要是因为晦涩难懂的算法不易交流、推广使用，也难以修改、扩展与调试，而且可能隐藏较多的错误。可读性实质上强调的是：越简单的东西越美。

健壮性（鲁棒性）。是指对非法输入的抵抗能力。它强调的是，如果输入非法数据，算法应能加以识别并做出处理，而不是产生误动作或陷入瘫痪。

时间复杂度与空间复杂度。时间复杂度是指算法的运行的时间消耗，同一个问题，不同的算法可能有不同的时间复杂度。问题规模较大时，时间复杂度就变得十分重要。尽管计算机的运行速度提高很快，但可以证明，这种提高无法满足问题规模加大带来的速度要求。所以追求高速算法仍然是件重要事情。

空间复杂度是指算法运行所需的存贮空间的多少。相比起来，这个问题并没有时间复杂度那样严重，但这并不是因为计算机的存贮空间是海量的，而是由人们面临的问题的本质决定的。时间复杂度与空间复杂度往往是一对矛盾。常常可以用空间换取速度，反之亦然。

§ 2.1.2 算法时间复杂度与空间复杂度

(一) 相关因素

简单地讲，算法的时间复杂度是指算法的运行的时间消耗的大小，但这里的“运行”，是指抽象的运行，并不是指在具体机上的运行。一个算法在具体计算机上的运行速度，不仅取决于算法本身，而且与目标计算机（运行算法的计算机）的速度有关，如果算法是用非目标指令（如汇编语言、高级语言等）描述的，则运行速度还与描述语言的编译器所生成的目标代码的质量（或解释器本身的质量）有关。

算法的时间复杂度指算法自身的抽象运行的时间消耗，与运行算法的目标计算机及描述算法的工具无关。一般而言，算法的时间复杂度与下列因素有关：

- 问题性质：有的问题比较复杂，而有的比较简单；
- 问题规模：同一问题，同一算法，其处理的对象的规模不同，则耗时也不同；

- 算法性质：同一问题，不同的解决方法，效率也不同。

一个算法的时间复杂度通常用函数描述（时间复杂度函数）。时间复杂度函数通常可以描述为输入规模的函数。输入规模可以是输入量或输出量，或两者之和，也可以是某种测度（如数组维数、图的边数等），它可用一个或多个量代表。所以时间复杂度可能是单自变量函数，也可以是多自变量函数。在下面的讨论中，为简明起见，用 f(n)表示时间复杂度，其中，n 表示问题规模。

下面举一个简单的例子。

[例] 考虑计算三次多项式

$ax^3 + bx^2 + cx + d$

一种直接的计算方法是：

```
s=a*x*x*x + b*x*x + c*x + d;
```

该算法共执行 6 次乘法，3 次加法。

考察另一种解法：

```
s=a;
s=s*x+b;
s=s*x+c;
s=s*x+d;
```

该算法共进行 3 次乘法，3 次加法。显然该算法的计算量少于前一种方法，尽管其看上去较复杂。第二种算法基于的原理是逐步提取共因子：

$a*x*x*x + b*x*x + c*x + d$

$= (ax+b)x^2 + cx + d$

$=((ax+b)x + c)x + d$

该例子表明，同一问题，采用不同的算法，效率是不同的。

> 对 n 次（即任意次）多项式，上面问题的第一种方法是行不通的（会造成不定的程序源码，即源程序动态改变），而第二种方法总是可行的。

(二) 复杂度的渐近表示

时空（时间与空间）复杂度的精确表示是件非常困难的事，许多算法的时间复杂度函数难以给出解析形式，即使是能够给出，也可能是个相当复杂的方程，方程的解法本身也可能是相当复杂的，所以，人们往往放弃求确切的时空复杂度的函数的企图，而通常用某些种简单函数近似表示，这就是时空复杂度的渐近表示。

为了说明渐近表示法，我们重温一下数学分析中的大 O 的定义：

设 T(n)和 f(n)均为正整数 n 的函数，则

$T(n) = O(f(n))$

表示存在一个正整数 M 和 n_0，使得当 $n \geq n_0$ 时，有

$| T(n) | \leqslant M | f(n) |$

根据极限的定义知，上式成立表示 $\lim\limits_{n->\infty} \dfrac{T(n)}{f(n)}$ = M， f(n)是 T(n)的一个上界函数（当 n 足够大时）。一般而言，T(n)的上界函数可能很多，但我们总希望能找到一个形式简单且较接近的上界。

常用的上界函数有

$\log(n)$, n^m, a^n, $n!$

下面给出几种常见的函数的比较关系：即当 n 充分大时，下列关系成立：

$O(a) < O(\log n) < O(n) < O(n.\log n) < O(n^2) < ... < O(n^m) < O(a^n) < O(n!) < O(n^n)$

这里，n 为自变量，a 和 m 为常数。

有相当多的一些算法，它们的时间复杂度可以表示为一个输入量的高次多项式，为此，给出一个相关的定理：

定理 2-1 若 $A(n) = a_m n^m + \ldots + a_1 n + a_0$ 是关于 n 的一个 m 次多项式，则

$$A(n) = O(n^m)$$

证：$| A(n) | = | a_m n^m + \ldots + a_1 n + a_0 |$

$$\leqslant (|a_m| + \ldots + |a_1| \frac{1}{n^{m-1}} + |a_0| \frac{1}{n^m}) n^m$$

$$\leqslant (|a_m| + \ldots + |a_1| + |a_0|) n^m$$

只要取 $n_0=1$， $M \geqslant |a_m| + \ldots + |a_1| + |a_0|$，就有

$$|A(n)| \leqslant Mn^m$$

从而有

$$|A(n)| = O(n^m)$$

证毕。

该定理指出，m 次多项式与 n^m 同阶，可用 n^m 逼近。

(三) 算法的时间复杂度的分类

通常，将算法按时间复杂度分为两大类：

多项式时间复杂度算法：渐近函数为多项式，或变化率不超过多项式。

指数时间复杂度算法：渐近函数变化率超过多项式，这类算法也称 **NP-困难**的算法。

§2.1.3 算法时间复杂度的度量

给定一个算法，如何求它的时间复杂度函数呢？对于用过程型程序设计语言或相应的伪码描述的算法，我们以“语句频度”为基础给出算法时间复杂度的度量。

语句频度（Frequency Count）是指语句被重复执行的次数。即对某语句，若在算法

的一次运行中它被执行 n 次，则它的语句频度为 n。语句频度也可称为程序步数。

我们用**算法中的各语句的语句频度之和**表示**算法的时间复杂度。**

这里的“语句”，是指描述算法的基本语句，它的执行，在语法意义上讲应是不可再分割的。据此，循环语句的整体、函数调用语句等不能算作基本语句，因为它们还包括循环体或函数体。

在一个算法中，有些语句可能很少被执行，且与程序规模无关，在估算时间复杂度时，可以不考虑它们。一般只考虑与程序规模有关的语句的语句频度。

显然，我们给出的时间复杂度估算方法是种近似方法，但从效果来讲，这种方法还是很合适的。

有时候，某些语句的频度是随机量，它可能依赖于具体输入值或应用方式，这些对算法本身而言是不确定的。此时，需用概率论方法解决。在这种情况下，一般要分别讨论时间复杂度在最好情况下、最差情况及一般情况下的值，分别称为最好性态，最差性态和平均性态。

有些算法是用递归方法描述的，这类算法的时间复杂度一般较容易用递归方程表示，此时，就涉及到递归方程求解问题。

[例 1.8] 考虑下列子程序，它的功能是求出数组 a 中各数的大小次序号，存入数组 b 中的对应位置。例如，若 a[]={4, 2, 3, 9, 6, 8, 7}，则程序结束后，数组 b 内容为 b[]={3, 1, 2, 7, 4, 6, 5}，表示 4，2，3，9，6，8，7 的大小次序分别为 3，1，2，7，4，6，5。

```
void OrderNumbers(int *a,  int n, int *b)
{
  int i, j;

  for (i=0; i<n;  i++) b[i]= 1;   ………………………..(1)：  n
  for (i=0 ; i<n-1;  i++)   ………………………………(2)：n-1
    for (j=i+1; j<n ; j++)   ………………………………(3)：∑_{i=0}^{n-1} (n-i-1)
      if (a[i] < a[j])   ……………………………… ….(4)：∑_{i=0}^{n-1} (n-i-1)
        b[j]++;        …………………………………..(5)：最多为∑_{i=0}^{n-1} (n-i-1)
      else b[i]++;
   return ;
  }
```

分析该程序的时间复杂度，其由两个并列的循环(1)(2)组成，而(2)又嵌套循环(3)。

各语句的的语句频度标在语句的右侧。总的语句频度最大为：

$$n + (n-1) + 3\sum_{i=0}^{n-1} (n-i-1)$$

$$= 2n-1 + 3(n-1)n/2$$
$$= 3n^2/2 + 7n/2 - 1$$
$$= O(n^2)$$

从这里看出，实际计算语句频度时，可以不考虑循环语句本身和条件语句本身，而只考虑它们所包含的循环体和条件体。

§2.2 穷举/试探法

可以说，穷举法是最直观最“笨”的一种方法，它的基本思想是，分别列举出各种可能解，测试（试探）其是否满足条件（是否是欲求的解），若是，则输出之。

能用穷举法解决的问题，它们的可能解（结果）应该是可列举的。因此，解结构一般应为离散结构。

这里举一个解不定方程的例子。不定方程（组）是指因为独立方程个数少于变量个数而导致方程有多解的方程。例如，2x+3y=20 就是一个不定方程。解这个方程，就是求出所有的解。不定方程一般都有限定条件，我们这里考虑正整数解的情况。解这个方程，一个简单的做法是，让 x 和 y 分别遍取 0 到 20 内的正整数，并代入方程计算，若值为 20，则表示找到一组解。具体的程序片断如下。

```
for (i=0; i<=20; i++)
   for (j=0; j<=20; j++)
      if (2*i + 3*j == 20 ) cout<<"\n"<<i<<' '<<j;
```

列举所有“可能解”是穷举法的关键。为了避免陷入重复试探，应保证列举过的可能解，在后面不再被列举。要做到这点，一般有两种方式，一是按规则列举，使得每次所做的列举都与以前不同。例如，上面求不定方程的程序就是这样。另一方式是盲目列举，但要随时检查当前的列举是否重复。检查重复一般需要记下以前的所有列举（至少记下它们的状态），这需要消耗存储空间。

§2.3 递推法与迭代法

递推法与迭代法是两种风格类似的方法，它们都是基于分步递增方式进行求解，在

许多其他更深入的算法设计方法中，都要结合这两种方法。

§2.3.1 递推法

递推法是针对这样一类问题：问题的解决可以分为若干步骤，每个步骤都产生一个子解（部分结果），每个子解都是由前面若干子解生成，它们都与问题规模相关，是相对于所相关的问题规模的完整解。不同的子解，其所相关的问题规模也随子解不同而递增。我们把这种由前面的子解得出后面的子解的规则称为**递推关系**。

例如，对于 Fibonacci（1175~？，意大利数学家）数列：

1 1 2 3 5 8 13 21 34 …

设 f(n)表示数列中第 n 项，则有：

f(1) = 1

f(2) = 1

f(k) = f(k-1) + f(k-2)

这就是一个递推关系，项的序号就是问题规模，第 2 项后的任一项都是根据它前两项的值导出。如果我们现在求 f(n)，那么可以按递推关系依次求 f(1)、f(2)、…、f(n-1)、f(n)，这里，每个 f(k)都是相对问题规模 k 的完整解。显然，每步递推，都需要保存前两项的值。具体的计算程序如下。

```
int Fibonacci(int n)
{
  int f1, f2, f3;
  int k;

  f1=1;
  f2=1;
  if (n==1) return f1;
  if (n==2) return f2;

  for (k=3; k<=n; k++)
  {
    f3 = f1+f2;
    f1 = f2; f2=f3;
  }

  return f3;
}
```

递推法的运用有如下两个关键：

■ 寻找递推关系：这是最重要的问题。递推关系有解析和非解析两种。解析递推关系是指能用一般数学公式描述的关系，也称**递推公式**。例如，Fibonacci 数列的递推关系就是解析的。非解析递推关系是指不能用一般的数学公式描述的关系，这类关系的描述，也许本身就是一个过程。下面将要介绍的分治法中的例子“循环赛日程安排”就是一个非解析递推关系的例子。另外，递推关系必须有始基，即最小子解（针对初始规模的子解）的值。没有始基，递推计算不能进行。例如，Fibonacci 数列的递推关系中，f(1)=1 和 f(2)=1 就是始基。

■ 递推计算：在确立了递推关系后，如何实现递推计算也是一个重要问题。如果递推关系依赖于较多的子解，则在递推计算中必须保存这些子解，如何有效地保存子解就成了递推计算的关键。递推计算的具体实现，可有非递归和递归两种。非递归是自底向上计算，即按子解规模的先小后大的次序递推计算，上面的 Fibonacci 数列生成就是非递归计算。递归计算是指使用递归法计算，其形式上是自顶向下。关于递归法，我们将在后面集中讨论。

§2.3.2 迭代法

迭代法和递推法类似，也是递增求解，但不同的是，递推法中，每步得到的解都是相对于对应问题规模的完整解，但对迭代法，中间步骤得到的解一般只是“近似解”，并不代表子问题的解（常常没有明确的子问题）。

迭代法比较经典的例子是数学近似计算，如方程求根，开方计算等。下面我们举一个用迭代法进行开方计算的例子。

考虑计算 $\sqrt{a}$ 。设 $\sqrt{a}$ =x，则 $x^2-1 = a-1$，若把 x 看作未知数，则问题就转化为解方程 $x^2-1 = a-1$，变换该方程得：(x-1)(x+1) = a-1，进一步有：

$$x = 1+\frac{a-1}{1+x}$$

当 x 是准确解时，代入上式，则左右两边相等。若 x 是近似值，则左右两边不等，但若 x 的误差越小，则左右两边相差越小。在求近似解的情况下，当左右两边相差足够小（比如说，只在小数点后第 n 位出现不同），则可认为当前 x 就是可接受的近似解。

将上式看作一个恒等式时，可以用 1+(a-1)/(1+x)代入上式右边的 x，得

$$x=1+\frac{a-1}{1+1+\frac{a-1}{1+x}}$$

在该式的基础上再用 1+(a-1)/(1+x)代入：

$$x=1+\frac{a-1}{1+1+\frac{a-1}{1+1+\frac{a-1}{1+x}}}$$

这样继续代入下去，就得到一个多级的连分式

$$x = 1 + \cfrac{a-1}{1+1+\cfrac{a-1}{1+1+\cfrac{a-1}{1+1+\cfrac{a-1}{\cdots}}}}$$

显然，若连分式级数越大，x 所处的位置就越深，从而，忽略分数(a-1)/(1+x)时，对整个式子的值的影响就越小，也就是说，级数足够大时，可将此式中的最底层的分数式去掉，然后做为 x 的近似值。而此式的计算，可按下面的递推式进行：

$$x_0 = 1$$

$$x_n = 1 + \frac{a-1}{1+x_{n-1}}$$

具体的程序实现，就是个简单的递推程序了，但要注意，一般情况下，递推的次数是受精度控制的。下面是具体的程序。

```
float Sqrt(float a)
{
    float precision=0.0001; //定义解的精度
    float x, x0;


    x=1;
    do
    {
      x0=x;
      x = 1+(a-1)/(x+1);
    }while (x-x0>precision || x0-x>precision);
      //当最近两个近似解的差的绝对值小于给定精度时结束
  return x;
}
```

该问题具有一定的普遍意义：连分式是迭代收敛的，可直接做为迭代式。在设计迭代式时，要特别注意收敛问题，有许多式子并不迭代收敛，所以不能用作迭代式。

与递推法类似，迭代关系也有时难以用解析方法表示，这时需要按规则迭代。

§2.4 递归

§2.4.1 递归与递归程序的概念

递归(Recursion)是一种描述与解决问题的基本方法，用来解决一类可归纳描述的问题，或说可分解为结构**自相似**的问题。所谓结构自相似，是指构成问题的部分与问题本身在结构上相似。这类问题具有的特点是：整个问题的解决，可以分为两大部分：

第一部分：是一些特殊（基础）情况，有直接的解法，即始基；

第二部分：这部分与原问题“相似”，可用类似（与整个问题的解法类似）的方法解决（即递归），但比原问题的规模小。

由于第二部分比整个问题的规模小，所以，每次递归，第二部分规模都在缩小，这样，如果最终缩小为第一部分的情况，则结束递归。因此，第一和第二部分应密切配合，以满足这个条件：**第二部分规模每次递归都在缩小，总有一次会缩小为第一部分的情况**。否则，程序不会终止。

这类问题在数学中很常见。例如，计算阶乘：

$$f(n) = n! = \begin{cases} 1 & \text{：第一部分，n = 0的情况，直接计算} \\ n.f(n-1) & \text{：第二部分，n > 0情况，递归} \end{cases}$$

在该式中，f(n-1)的计算与原问题 f(n)的计算“相似”，只是规模较小。在看算术求和：

$$s(n) = \sum_{k=0}^{n} k = \begin{cases} 0 & \text{：n = 0} \\ s(n-1)+n & \text{：n > 0} \end{cases}$$

也有许多非算术计算问题可用递归方法解决。例如，设一维数组 A 的元素 $A[k_1]$~$A[k_2]$中存放整数，现要求出它们中最大者，递归求法为：

a) 若 $k_1=k_2$，则 $A[k_1]$即为所求；

b) 若 $k_1<k_2$，则先按类似的方法，求出 $A[k_1+1]$~$A[k_2]$中最大者，设其为 m，然后，比较 $A[k_1]$和 m，二者中最大者即为所求。

在计算机程序中，这类递归问题，可使用递归程序解决。

一个可按名调用的程序模块称为一个子程序（函数或过程）。在一个子程序中，可以调用其他子程序或自己。如果一个子程序直接调用自己，则称该子程序为**直接递归程序**；若一个子程序 A 调用了另一子程序 B，而 B 直接调用 A，或 B 所调用的其他子程序（包括各级子程序）中的某个（些）调用了 A，则称 A 是一个**间接递归程序**。

用递归程序解决递归问题时，将整个递归问题描述为一个以问题规模为参数的子程序（即递归程序）。在程序中，用分支语句分别实现递归的两大部分。对第二部分中的“按

类似方法解决”，主要通过用不同的参数实现。

例如，关于阶乘的计算，可通过下列的递归程序：

```
long Fact(int n)
{
  if (n<=0) return 1;    //第一部份，始基
  else return n*Fact(n-1);    //第二部份，用实参 n-1 递归调用
}
```

再如，在一维数组中求最大值的程序为：

```
int GetMax(int a[], int k1, int k2)
{
  int m;

  if ( k1 == k2) return a[k1];
  m = GetMax(a, k1+1, k2);
  if (a[k1] > m) return a[k1];
  else return m;

}
```

在计算机程序设计中，从逻辑功能讲，递归相当于循环。这是因为，递归程序通过直接或间接地调用自己，实现了自己的重复执行。因此，许多循环问题，都可以较容易地用递归实现，反之亦然。正因为这样，有些计算机程序设计语言只支持递归，如 Prolog 和 Lisp；也有许多语言不支持递归，如基本 BASIC、COBOL 及 FORTRAN 等。

例如，计算阶乘，可用我们熟知的循环方法：

```
long Fact(int n)
{
  int i; long t;

  t=1;
  for ( i=1; i<n; i++) t = t*i;
  return t;
}
```

§ 2.4.2 递归程序设计要点

递归是一种概念性/思想性的技术，没有一定的规程。要很好地运用它，一般需要很

好的数学抽象能力。下面就递归程序设计，归纳几个要点。

划分问题、寻找递归：许多问题，并没有明显的递归解法，这就需要寻找递归方案。首先要试着看能否把问题的处理分为两大类情况。一类情况是可以直接解决（不需要递归，为基本情况），另一类情况可按与原问题的解法（即现在正使用的解法）类似的解法进行（称为递归情况）。

设计函数、确定参数：使用递归，必须设计为子程序（函数或过程）的形式。子程序的参数尤为重要，决定着递归的实现。递归子程序中的参数，除了应具有一般子程序所需要的参数外，还需要递归参数。这里，**递归参数**是指反映递归问题的规模或情况的参数，依据它们，就可以把整个问题的解决划分为两大类情况（如上一点阐述），而且，在第二类情况中，“按类似方法处理”的实现，是通过用不同的递归参数调用自己实现的。

例如，阶乘的递归程序中，long Fact(ing n)是函数原型，n 是递归参数，反映问题的规模，也是划分情况的标准（n<=0 时为一种情况，其余为另一种）；再如，对在数组中求最大数的程序 int GetMax(int a[], int k1, int k2)，递归参数是 k1 和 k2. 当 k1==k2 时，为基本情况，其他为递归情况。在这递归情况中，用实在参数 k1+1 和 k2 调用自己，即 GetMax(a, k1+1, k2).

从应用角度看，递归参数往往不是子程序接口参数所必需的，或不符合所需的形式，此时，可以对递归程序进行“封装”，将递归参数改造为所需的形式，或将其隐蔽起来。例如，对上面给出的 GetMax(int a[], int k1, int k2)，如果我们期望函数参数中只指出数组 a[]中从 a[0]起的元素个数，则可以如下封装：

```
int GetMax(int a[], int n)
{//返回 a[0]~a[n-1]中的最大者
  return GetMax(a, 0, n-1);
}
```

设置边界、控制递归：递归调用相当于循环执行，因此，必须处理好循环条件。递归中，第一类情况（非递归的情况）就起到循环条件的作用，这是因为，满足第一类情况时，就不递归调用了，本次调用即可结束。把第一类情况的判别条件称为递归控制条件。递归控制条件相当重要，如果出现错误，往往导致无限循环（从而很快耗尽内存）。

最基本的一点是，要确保第一类情况能在某一次出现（条件成立）。要做到这一点，就应该保证在第二类情况中，递归参数的选择，应能使其向递归控制条件发展。

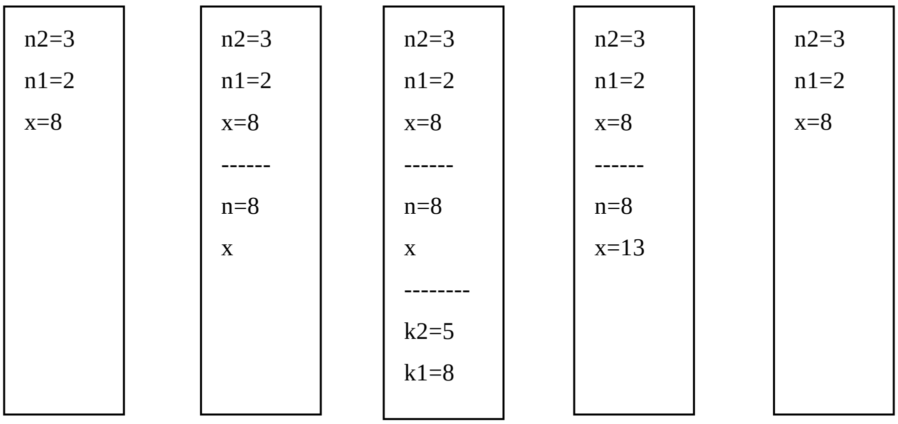
例如，在阶乘计算中，递归控制条件为 if (n<=0)，递归调用为 Fact(n-1),其每执行一次，实在参数就在上一次的基础上减 1，因此，总有某一次，能满足递归条件。在求数组中最大元素的程序中，递归条件是 if (k1 == k2)，递归调用为 GetMax(a, k1+1, k2)，其每执行一次，第二个实在参数就增加 1，向第三个参数靠拢，因此，总有一次，第二与第三参数会相等，即满足递归控制条件。

§ 2.4.3 递归程序执行机理

要彻底理解递归程序，最好的方法是先理解递归程序的执行机理。递归程序的执行方式与一般的子程序调用方式类似，所以先要分析一般子程序的调用（执行）方式。

(一) 一般子程序的执行方式

对大多数的程序设计语言，子程序的调用执行都基于先进后出存储设备----栈。程序运行时，设置一个栈，称为运行栈。每调用一次子程序，就要为被调程序的各实在参数和临时变量分配存储空间。这种存储空间一般是分配在运行栈中的。子程序结束后，进行相应的退栈，释放本次调用所分配的栈空间。



(a) (b) (c) (d) (e)

(修改注：下面这段不需要修改)

(a): 调用 P1(2,3),执行到(4)之前 **(b)**: 执行(4)，进入 P2(x)，执行到 P2 的(10)之前

(c) : 执行 P2 的(10)，调用 P3，执行到 P3 的(15)之前 **(d)**: 执行完 P3，返回到 P2 的(11)

(e): 执行完 P2，返回到 P1 的(5)

图 2-1 子程序调用的运行栈示例

例如，设有下面的子程序：

```
[1]  void P1(int n1, int n2)
[2]  {
[3]    int x=8;
[4]    P2(x);
[5]    … …
[6]  }

[7]  int P2(int n)
[8]  {
```

```
[9]     int x;
[10]   x=P3(n, 5);
[11]   return x;
[12] }

[13] int P3(int k1, int k2)
[14] {
[15]     return k1+k2;
[16] }
```

现假定调用 P1，即执行：P1(2,3)，则运行栈的情况如图 2-0.

(二) 递归子程序的执行方式

递归子程序的执行（调用），仍然遵循一般子程序的执行方式，需要注意的是，将每次递归调用都视为不同子程序的调用，即每次递归调用，都将递归子程序的实在参数和各临时变量进栈，做为栈中的新的一项。据此可知下面的子程序是个不能终止的过程（相当于无限循环），且会无休止地进栈，直至可用内存耗尽。

(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)
n=4	n=4	n=4	n=4	n=4	n=4	n=4	n=4	n=4	(返回24)
	n=3	n=3	n=3	n=3	n=3	n=3	n=3	(返回6)	
		n=2	n=2	n=2	n=2	n=2	(返回2)		
			n=1	n=1	n=1	(返回1)			
				n=0	(返回1)				

（修改注：请作者核 g、h、i、j 中返回值是否正确？？？？？？--答：正确，因为 0 和 1 的阶乘都是 1）

(a): 调用 Fact(4);
(b): 在 Fact(4)中调用 Fact(3);
(c): 在 Fact(3)中调用 Fact(2);
(d): 在 Fact(2)中调用 Fact(1);
(e): 在 Fact(1)中调用 Fact(0),返回 1;
(f): 返回到 Fact(1)，计算出了 Fact(0);
(g): 返回到 Fact(2)，计算出了 Fact(1);
(h): 返回到 Fact(3)，计算出了 Fact(2);
(i): 返回到 Fact(4)，计算出了 Fact(3)
(j): 返回到主调，计算出了 Fact(4)

图 2-2 递归程序执行方式示例

```
void P1(int n)
{
  P1(n-1);
}
```

图 2-0 给出了求阶乘子程序 Fact(n)的执行过程中栈的变化情况（设调用 Fact(4)）

§2.4.4 Hanoi 塔问题与运行图

递归程序的执行过程，也可以很形象地用运行图表示，下面通过 Hanoi 塔问题说明。

Hanoi 塔问题出现在 19 世纪的欧洲，用来比喻世界的寿命。问题是这样的，设有三根钻石杆，分别编为 A 号 、B 号和 C 号，在 A 号杆上堆叠 64 个金盘，每个盘大小（半径）不同，只允许小盘在大盘之上。最底层的盘最大。现在要求将 A 号杆上的盘全部移到 C 号杆，在移的过程中可以借助 B 号杆（以 B 号杆为中转）。规定每次移一个盘，并且在任何时刻，只能移最上面的盘，且大盘不能放在小盘上面。

我们现在考虑编写计算机程序，使其给出移盘的过程。很显然，将 n 个盘子从 A 移到 C，可以按如下步骤进行：

- 借助 C，将 A 上的最上面的 n-1 个盘子按规定移到 B
- 将 A 上的最底层的一个盘子移到 C
- 借助 A，将 B 上的 n-1 个盘子按规定移到 C

这是个递归过程。为了编程方便，我们给每个盘子一个自然数编号，位于最上面的盘子的编号为 1，其余编号从上到下依次为 2、3、…n。具体的程序如下：

```
void MoveDisks(int n, char fromP, char tempP, char toP)
{//将 fromP 上的 n 个盘子借助 tempP 移到 toP 上
 if (n>0) //n<=0 时，没有盘子可移，无动作
 {
    MoveDisks(n-1, fromP, toP, tempP);   //借助 toP 将 fromP 上的 n-1 个盘移到 tempP
    cout<<'\n'<<fromP<<":"<<n<<"-->"<<toP;   //将 n 号盘从 fromP 移到 toP
    MoveDisks(n-1,tempP,fromP,toP);   //借助 fromP 将 tempP 上的 n-1 个盘移到 toP
 }
}
```

例如，若盘子的数目为 3，则该程序输出的移动过程为：

A:1→C　//表示将 A 上的 1 号盘移到 C 上，余类推。

A:2→B

C:1→B

A:3→C

B:1→A

B:2→C

A:1→C

分析上面的递归程序，设移动 n 个盘子所需要的移动次数为 T(n)，则显然有

$T(0) = 0$

$T(n) = 2T(n-1) + 1$

这是一个递归方程。可以证明：

$T(n) = 2^n - 1$

这样，移动 64 个盘，需要的移动次数是

$2^{64} - 1 \approx 1.6 \times 10^{19}$

设每秒钟移动一个盘，一年约有 3.2×10^7 秒，则全部完成约需 5×10^{11} 年（500 亿年）。天文学家估计，宇宙的年龄为 2×10^{10} 年。这样，按规定移完这 64 个盘，需要 25 倍宇宙的年龄的时间，所以，19 世纪的人认为这个时间大于地球的寿命。即使现在的微处理器主频达到 4G（1G 为 2^{30}），平均每条指令需要 4 个周期，则每秒也不过执行 1G 的移动动作，这样，$2^{64}-1$ 个动作约需要 2^{34} 秒，约合 198841 天（约 545 年）。

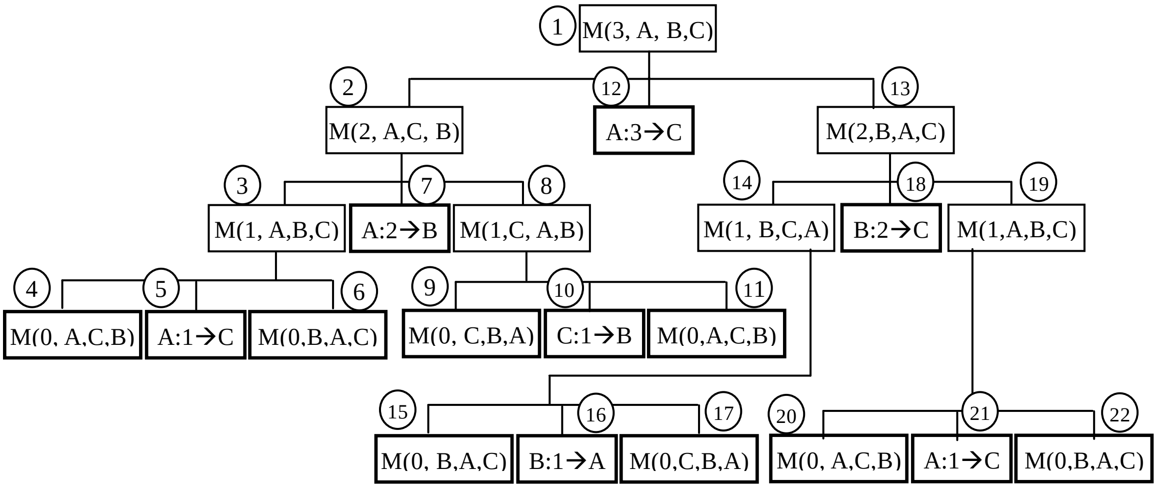


图 2-3 Hanoi 塔程序的运行图

为了方便直观地描述递归程序的执行轨迹，引入运行图。运行图是由结点和连接结点的边构成。每个结点代表一个子程序调用或一段程序的执行。如果某结点 A 代表子程序调用，则它可以有子结点（用从 A 到子结点的边表示这种关系），每个子结点同样代表 A 中一个子程序调用或一段程序的执行。A 的各子结点构成的序列（从左到右）就代表 A 所代表的子程序的执行。

图 2-0 给出执行 MoveDisks(3,'A', 'B', 'C')的运行图。为了方便，图中用 M 代替 MoveDisks，并且代表钻石杆的字符 A、B、C 也不加引号。粗体框表示终结点，即不再包含有效的子程序调用；方框上的圆圈表示执行次序。

§2.5 逐步求精

§2.5.1 基本思想

逐步求精是在 20 世纪 70 年代随着结构化程序设计技术/思想而产生的。在程序设计方面，逐步求精是一种十分有效的基本策略（或思维方式）。

对一些较复杂问题，往往一次写出它的程序实现很困难，而人的思维是由大到小、由外到内、由表及里、自顶向下、由粗到精来解决问题的。这种思想就称为**逐步求精。**

简单地讲，逐步求精方法，是一种逐步“划分”的方法，即将问题的解决，先用几个大/粗的模块的组合表示。对这些模块，先不考虑它们的内部实现，只规定其功能。然后再按类似方法继续划分这些模块，直到它们都变为程序设计语句。在这种划分中，应遵循下列规则：

- **保证模块的粒度应逐步变小**。粒度越大/粗，“说明性”越强，越远离程序设计语言，但越容易给出（设计）；
- **保证当前正确**。对每次划分，若假定各模块都可正确实现，则它们的当前组合（即划分方式）是整个问题的正确实现；

对逐步求精的描述，一般采用“伪码”。所谓**伪码**，是指不完全的程序代码，它一般以程序设计语言(典型的是 C/Pascal 之类的结构化程序设计语言)的流程控制语句（如 while, for, if 等）为主体，夹杂自然语言的描述。

§2.5.2 应用示例

下面用一个简单例子说明逐步求精方法。

考虑求矩阵鞍点的问题。所谓矩阵鞍点，是指满足这样条件的矩阵元素：它是所在行上的最小元素，同时是所在列上的最大元素。可以证明，一个矩阵可以有多个鞍点，但它们的值均相等。

显然，求鞍点的一个直接的方法是，检查矩阵中每个元素是否为鞍点，用伪码描述为（设矩阵名为 a，有 n 行 m 列，元素下标从 0 起）：

```
for (i=0; i<n; i++)
  for (j=0; j<m; j++)
  {
    判断 a[i][j]是否为鞍点；
    if (a[i][j]  是鞍点)
        输出 a[i][j];
  }
```

这是算法的最初版本。这里，关键问题是判断 a[i][j]是否为鞍点，所以关键是细化

模块“判断 a[i][j]是否为鞍点”。解决该问题，先检查 a[i][j]是否为 i 行上的最小者，若是，则继续检查其是否为 j 列上最大者，若是，则为鞍点。其他情况都不是鞍点。该过程的伪码描述为：

```
isSaddle = 0;
检查 a[i][j]是否为 i 行上最小者；
if (是)
{
检查 a[i][j]是否为 j 列上最大者；
if (是) isSaddle = 1;
}
```

这段程序结束后，isSaddle 为非 0 时表示 a[i][j]为鞍点，否则不是鞍点。在这里，有两个模块需要细化：

a) 检查 a[i][j]是否为 i 行上最小者，这可以先找出 i 行上最小者的下标，然后与 a[i][j]比较即可：

```
kk=0;
for (k=1; k<m; k++)
   if (a[i][k] < a[i][kk] ) kk=k;
if (a[i][kk]==a[i][j]) a[i][j]为 i 行上最小者；
```

请思考，上面 if 语句是否可以改为这样的形式(对下一段也有类似情况)：“if (kk==j) a[i][j]为 i 行上最小者”

b) 检查 a[i][j]是否为 j 行上最大者：

```
kk=0;
for (k=1; k<m; k++)
   if (a[k][j] > a[kk][j] ) kk=k;
if (a[kk][j]==a[i][j]) a[i][j]为 j 列上最大者；
```

将这两段程序代入上面的“判断 a[i][j]是否为鞍点”的程序：

```
isSaddle = 0;
kk=0;
for (k=1; k<m; k++)
   if (a[i][k] < a[i][kk] ) kk=k;

if (a[i][kk]==a[i][j])
{
```

```
kk=0;
for (k=1; k<m; k++)
     if (a[k][j] > a[kk][j] ) kk=k;
if (a[kk][j]==a[i][j]) isSaddle = 1;
}
```

将该段程序代入最前面的求鞍点的伪码中，即得完整的求鞍点的程序片段：

```
for (i=0; i<n; i++)
   for (j=0; j<m; j++)
   {
     isSaddle = 0;
     kk=0;
     for (k=1; k<m; k++)
     if (a[i][k] < a[i][kk] ) kk=k;

     if (a[i][kk]==a[i][j])
     {
     kk=0;
     for (k=1; k<m; k++)
         if (a[k][j] > a[kk][j] ) kk=k;
     if (a[kk][j]==a[i][j]) isSaddle = 1;
   }
   if (isSaddle)    输出 a[i][j];
  } //for (j=0; ….)
```

§2. 6 分治法

§2.6.1 基本思想

分治是指“分而治之”（Divide and conquer）。对某些问题，直接求解是很困难的，但若把它们分解为一些较小的问题（子问题），然后各个击破，最后综合起来，就得到整个问题的解决。这就是分治法的基本思想。

使用分治法的前提是问题是可分治的。所谓**可分治**，是指满足下列几点:

可分解：问题能分解为更小更容易解决的子问题；

可综合：由子问题的综合，可得到整个问题的解决。

分解的方法一般有两大类：

平面分解：问题能划分为若干相互独立的子问题。这里的相互独立是指各子问题的解决，不相互依赖。下面要介绍的顺序统计问题就是典型的平面分解。前面的 Hanoi 塔问题，也属于平面分解。

迭代分解：将待解决问题分解为子问题 s_1、s_2、…、s_n，其中，s_n是问题的最终解决，s_i的解决，需在 s_{i-1}的基础上或在 s_1~s_{i-1}中的某些的基础上解决。下面要介绍的循环赛日程安排就属于这种迭代分解。

归并排序（迭代分解）和快速排序（平面分解）也是典型的分治法例子，这将在后面的章节中介绍。

§2.6.2 平面分治法示例----顺序统计

顺序统计是指在 n 个数的集合中（这里设为一维数组），找出第 k 小的数（即从小到大排行第 k，最小元素为第 1 小）。一个直观的解法是，将这 n 个数排序，然后取其中第 k 个即可。但是，一般情况下，排序的时间复杂度不能低于 O(nlogn)。能否不通过排序直接获得解呢？。通过分治法就可以解决这个问题。

假定 n 个元素在数组 a 中（a[0]~a[n-1]中），那么，分治法的具体做法是，先在 a 中任选一个元素 x，以 x 为基准，将 a[0]~a[n-1]分为三部分，第一部分元素都小于 x，第二部分（称为划分点）元素是 x，第三部分元素都大于或等于 x。设第一部分中的元素个数为 m，那么，若 k≤m，则第 k 小元素在第一部分中，而且也是第一部分中的第 k 小元素，可按类似的方法在第一部分中找；若 k=m+1，则第二部分中的 x 即为所求；若 k>m+1，则第 k 小元素在第三部分中，且为第三部分中的第(k-m-1)小元素，也可按类似方法在第三部分中找。

设 Partition(a,p1,p2)实现这种划分（称为分割算法，其将 a[0]~a[n-1]分为上述的三部分，返回划分点的顺序号），则顺序统计问题可方便地用递归程序实现：

```
long OrderStatistics(long a[], long p1, long p2, long k)
{//在 a[p1]~a[p2]中，找出第 k 小者，并返回其值

  long p,num;

  if (k<1 || k>p2-p1+1) return -1; //k 超出范围时返回特殊标记（这里为简单用-1 做特殊标记）
  if (p1>=p2) return a[p1];    //若 a[p1]~a[p2]只有一个元素（或参数不合法），则返回该元素
  p = Partition(a,p1,p2);    //划分，返回划分点
  num=p-p1; //求出划分点 p 前面的到 p1 的元素个数

  if (k==num+1) return a[p]; //第 k 小元素为分割点
  if (k<=num) return OrderStatistics(a, p1, p-1, k); //第 k 小元素在前部
  return OrderStatistics(a, p+1, p2,k-num-1); //第 k 小元素在后部
}
```

该算法也可用非递归程序实现，具体留作练习。

下面的问题是，如何实现分割算法 Partition(a, p1,p2)。设置两个指示器 i 和 j，它们的初值分别为 p1 和 p2，且把 a[p1]送入工作单元 x 中保存（选第一个记录为基准），然后进入分割：比较 a[j]和 x，若 a[j]>=x，则 j 减 1 后继续与 x 比较，直至 a[j]＜x，然后，将 a[j]移至 a[i]的位置，令 i 加 1，接着进行 a[i]和 x 的比较，若 a[i]<=x，则令 x 加 1，然后继续比较，直至满足 a[i]＞x，此时，将 a[i]移至 a[j]的位置，令 j 减 1，之后，一个回合的分割完成，然后，重复上述过程，直至 i==j。此时 i 便是记录 x 所应在的位置。至此，一趟分割完成。下面给出该过程针对 A[] = {10，2，9，12，8，15，6，16，18}的例子。

↑[ ]　2　9　12　8　15　6　16　18↑	初态：a[0]=10 传到 x 中，i 与 j 分别指向头和尾
6 ↑2　9　12　8　15　[ ]↑ **16　18**	右→左：18 和 16 大于 x，j 连续左移 2 步，遇到 6 停止，将 6 移到 i 处，且 i 前进一步；
6　2　9 ↑[ ]　8　15↑ **12　16　18**	左→右：2 与 9 小于 x，i 连续右移 2 步，遇到 12 停止，将 12 移到 j 处，且 j 左移一步；
6　2　9　8 ↑[]↑ **15　12　16　18**	右→左：15 大于 x，j 左移一步，遇到 8 停止，将 8 移到 i 处，且 i 前进一步；
6　2　9　8 ↑[10]↑**15　12　16　18**	左→右：i==j，停止分割，将 x 移到 i 处

下面是具体的程序：

```
long Partition (long a[], long p1, long p2)
{//对 a[p1]~a[p2]进行分割，返回分割点的序号
  long i, j;
  int x;

  i =p1;
  j =p2;
  x =a[i];
  while (i<j)
  {
    while ( a[j]>=x &&   i<j ) j-- ;   //右→左扫描
    if (i<j)   {a[i]=a[j]; i++;}
    while (a[i]<=x && i<j ) i++; //左→右扫描
    if (i<j) {a[j] =a[i]; j--;}
  }
  a[i]=x;
  return i;
```

```
    }
```

该程序也可用在快速排序算法中。

注意，该程序结束后，原数组的元素次序也被改变了（副作用!）。这样也很难顺便求出第 k 小元素在原数组中的序号。如果要在该算法中避免该问题，则需要在原数组的复制品上工作。

§2.6.3 迭代分治法示例----循环赛日程安排*

我们这里考虑用分治法解决体育比赛中循环赛的赛程安排。

设有 n 支篮球队参加循环赛，共进行 n-1 天比赛，每队每天必参加且仅参加一次比赛。这里，设 $n=2^k$，k 为正整数。可以证明，当 n 满足此条件，该问题有解。

为了编程方便，我们设 n 支球队编号分别为 1,2,…,n. 该问题实质上是求 1 到 n 这 n 个自然数的 n-1 个全排列，每个排列对应于一天的赛程，若第 k 号排列位置上的数为 s, 则表示在该天球队 k 和球队 s 比赛。显然，由于规定一支球队一天只参加一场，所以，对每个全排列，若第 k 号排列位置上的数为 s，则在这个排列中，第 s 号排列位置上的数必为 k. 另外，对任意两天的赛事（即任意两个排列），它们的每个对应排列位置都不能有相同的数（否则就重复比赛了）

	第 1 天
球队 1	2
球队 2	1

(a) 2 支球队赛程

	第 1 天	第 2 天	第 3 天
球队 1	2	3	4
球队 2	1	4	3
球队 3	4	1	2
球队 4	3	2	1

(b) 4 支球队赛程

	第 1 天	第 2 天	第 3 天	第 4 天	第 5 天	第 6 天	第 7 天
球队 1	2	3	4	5	6	7	8
球队 2	1	4	3	6	7	8	5
球队 3	4	1	2	7	8	5	6
球队 4	3	2	1	8	5	6	7
球队 5	6	7	8	1	4	3	2
球队 6	5	8	7	2	1	4	3
球队 7	8	5	6	3	2	1	4
球队 8	7	6	5	4	3	2	1

(c) 8 支球队赛程

图 2-4 循环赛日程安排示例

该问题的解决，直接的方法是回溯法，即逐步生成排列，每排列一个数，就检查是否满足条件，若不满足，则撤消，尝试下个选择。这种方法很耗时，最多可能试探 n!次。下面给出一种更快捷的解法，是基于迭代的分治法。

为了方便，我们将赛程排列做成一张 n 行(n-1)列的表，表的每列代表一天，每行代表一支球队，i 行 j 列上的数字就代表球队 i 在第 j 天的比赛对手的编号。显然，表中有一半的信息是重复的，即某列中，若有一行表示球队 k 与 s 比赛，则必有另一行，它表

示球队 s 与 k 比赛。这里我们为了迭代才不省略重复信息。

首先，考虑只有两支球队时的赛程，这很容易排定，见图 2-0(a)。然后考虑只有 4 支球队的情况，这需要 3（即 4-1）天，即表有 4 行 3 列。该表在两支球队的赛程表的基础上生成。具体方法是：

■ 左上角(2 行 1 列)：将只有两支球队时的表做为左上角；

■ 左下角（2 行 1 列）：各元素通过将左上角对应元素加 2 得到；

■ 右上角（2 行 2 列）：设置为 1-2 号球队与 3-4 号球队的比赛日程，例如，先（即第 2 天）安排 1-2 依次与 3-4 比赛，然后几天的比赛，都是在上一天的赛程的基础上循环轮转得到，例如，第 2 天为 3 和 4，则第三天为 4 和 3。

■ 右下角（2 行 2 列）：可参照右上角生成（因为此时表的上下两部份信息重复）：上部 i 行 j 列的值，填入 j 行 i 列。

然后，可用类似的方法依次生成 8，16，32，…, 2^k 支球队的赛程。见图 2-0。

总结这种解法，我们是把求解 2^n 支球队的问题，划分成了依次求球队数为 2^1、2^2、…、2^n 的情况的赛程。2^k 支球队的赛程，是在 2^{k-1} 支球队的赛程的基础上求得的。所以属于迭代方法。但每次迭代中（在求 2^k 支球队的赛程时），是将问题分为 4 部份：左上角（$1\sim2^{k-1}$ 行中的第 $1\sim2^{k-1}-1$ 列）、左下角（左上角的正下方全部）、右上角（$1\sim2^{k-1}$ 行中的第 $2^{k-1}\sim2^k-1$ 列）、右下角（右上角的正下方全部）。其中，左上角为 2^{k-1} 的情况的解，左下角根据左上角得到，右上角为另组赛队的循环轮转，右下角则根据右上角得到。所以，每次迭代属于平面划分。

具体的源程序如下：

```
const int CNST_MaxPlayer=8;
void GameTimeTable(int k, int a[][CNST_MaxPlayer-1] )
{//求球队数为 2 的 k 次方时的赛程表，结果存于 a[][CNST_MaxPlayer-1]
 int n,n0, i,j,kk;

 //生成只有两支球队的日程表
 a[1][1]=2;
 a[2][1]=1;

 kk=1;
 n=2;
 while (kk<k)
 {
  kk++;
  n0=n;
  n=n*2;

  //填左下角
```

```
    for (i=n0+1; i<=n; i++)
        for (j=1; j<=n0-1; j++)
            a[i][j] = a[i-n0][j] + n0;

    //填右上角的第 1 列
    a[1][n0] = n0+1;
    for (i=2; i<=n0; i++)
        a[i][n0] = a[i-1][n0] + 1;

    for (j=n0+1; j<n; j++)      //填右上角的其他各列
    { for (i=1; i<n0; i++)
            a[i][j] = a[i+1][j-1];
        a[n0][j] =a[1][j-1];
    }

    for (j=n0; j<n; j++) //填右下角各列
        for (i=1; i<=n0; i++)
            a[a[i][j]][j] = i;

  } //while(kk)

}
```

本章小结

本章首先介绍了算法的基本概念，然后重点介绍了三种基本的算法（程序）设计方法和策略。

算法是用于求解访问的规则序列，具有有穷性、确定性、可行性，并具有输入量和输出结果。衡量一个算法的好坏，一般从正确性、可读性、健壮性、时间和空间复杂度等几个方面进行。

时间复杂度用来衡量算法的执行的时间消耗，只与算法本身及算法的处理规模有关，而与具体执行算法的机器无关。一般可用语句频度（即语句被重复执行的次数）之和作为算法的时间复杂度的度量。

穷举法顾名思义是通过列举所有可能解来求解的。每列举出一个可能解，就检查它是否满足解的条件，若满足则接受其为解，否则放弃，然后尝试新的列举。注意，在穷举法中，每次列举的都是一个形式完整的可能解（不是部分可能解）。另外，要实现无遗

漏无重复的列举，需要按某种规则有条不紊地进行，有时还需记录下已列举的路径。

递推法与迭代法都是基于递增求解的方法。求一个解分成若干步骤进行，每个步骤得到的都是部分解（解的一部分）或近似解，最终得到的是完整解。每个部分解都是在前面的部分解的基础上求得的----这种方式称为递推/迭代。递推/迭代往往可以按公式进行，但也有许多情况没有公式，只是按某种规则进行。递推法与迭代法的主要不同在于迭代法产生的中间结果（部分解）一般是“近似”解，而递推法的部分解是结构上不完整的解，即结构上的部分解。

分治法的基本思想是“分而治之”：将问题分成若干个较易解决的子问题，加以“各个击破”，然后再将各子问题的结果综合起来，得到整个问题的解。当子问题的解决和整个问题的解决具有相似性时，分治法的实现常常使用递归技术。

递归法是一种十分有效的解决问题的方法。递归法的关键是将问题分为两大部分，一部分可以立即解决（不需要递归），而另一部分与原问题“相似”，只是规模不同，所以可以按与解决原问题“类似”的方法解决----递归调用。这里的关键是，1)划分出“相似”部分；2)划分出的两大部分必须配合：在递归调用过程中，第二部分总有一次会归结到第一部分的情况，从而结束递归。

逐步求精可以看作是一种处理/解决问题的方式（严格地讲，不是一种算法设计方法），或一种思维方法。它的基本精神是：按由大到小、由外到内、由表及里、自顶向下、由粗到精来的思想/方式解决问题。这种方式符合一般人的思维特点。用这种方式解决问题，可以使问题从大化小、将难化易，变得有条不紊。

习　题

1．用递推法生成杨辉三角形。杨辉是我国南宋数学家，他在 1261 年写的《详解九章算法》一书中，把二项式系数排为三角形，人称杨辉三角形。例如，下面是一个 6 行的杨辉三角形。

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5  10  10   5   1
… …
```

2．计算由曲线 $y=x^2+1$、直线 $x=a$ 和 $x=b$ 及 x 轴所围成的区域的面积。这里，$a<b$， $a>0, b>0$。

3．用非递归方法实现顺序统计算法。

4．用穷举法生成 n 阶魔方。所谓 n 阶魔方，是指这样一种 $n\times n$ 的方阵，如果将 n^2 个自然数 $1\sim n^2$ 分别填在它的 n^2 个不同的格子中，它的每行之和、每列之和、每主对角线（正主对角线和反主对角线）之和都分别等于某个常数。例如，下面是一个 3 阶方阵。

4	9	2
3	5	7
8	1	6

我国是最早研究魔方的国家。1275 年，南宋数学家杨辉对魔方作了研究，他在《续古摘奇算法》一书中，描述了 3~10 阶魔方，其中对 3 阶魔方的生成这样描述："九子斜排，上下对易，左右相更，四维挺出，戴九履一，左三右七，二四为肩，六八为足"，其意思是，将 1~9 这 9 个数排成三条平行斜线，第一条上为(1,4,7),第二条为(2,5,8),第三条为(3,6,9)，如下所示：

```
      1
    4   2
  7   5   3
    8   6
      9
```

然后，上下对调后收缩一行，左右对调后收缩一列。这个方法可以推广到任意奇数阶魔方。我们现在的任务是，不按这种方法，而使用穷举法生成。

5．编写生成"选排列（注：是全排列吗？---答：应该为选排列）"的程序（例如，(1,2,3,4)四个数中取 3 个的各种不同取法（排列）有：123，132，124，142，134，143，213，231，214，241，234，243，…..，共 24 种）.

6．编写生成组合的程序（例如，(1,2,3,4)中取 3 个的不同取法（组合）有：123，234，341，412）

7．编写程序，在由 n 个数构成的序列中，找出最长的单调递增子序列。要求时间复杂度不超过 $O(n^2)$。