CHAPTER 5

第5章 递　　归

在算法设计中，经常需要用递归方法求解，特别是本书后面的树、查找和排序等几章中，大量地用到递归算法。递归是计算机科学中一个重要的方法，很多程序设计语言如 C/C++都支持递归程序设计。本章介绍递归的定义和递归算法设计方法等，为后面的学习打下基础。

5.1 什么是递归

5.1.1 递归的定义

在定义一个过程或函数时出现调用本过程或本函数的成分，称为递归。若调用自身，称为直接递归。若过程或函数 p 调用过程或函数 q，而 q 又调用 p，称为间接递归。

递归不仅是数学中的一个重要概念，也是计算技术中重要的概念之一。在计算技术中，与递归有关的概念有：递归关系、递归数列、递归过程、递归算法、递归程序、递归方法。

(1) 递归关系指的是：一个数列的若干连续项之间的关系。

(2) 递归数列指的是：由递归关系所确定的数列。

(3) 递归过程指的是：直接或间接调用自身的过程。

(4) 递归算法指的是：包含递归过程的算法。

(5) 递归程序指的是：直接或间接调用自身的程序。

(6) 递归方法指的是：是一种在有限步骤内，根据特定的法则或公式对一个或多个前面所列的元素进行运算，以确定一系列元素(如数或函数)的方法。

如果一个递归过程或递归函数中递归调用语句是最后一条执行语句，则称这种递归调用为尾递归。

【例 5.1】 以下是求 n!(n 为正整数)的递归函数。

```
int fun(int n)
{   if (n== 1)                              //语句 1
        return(1);                          //语句 2
    else                                    //语句 3
        return(fun(n - 1) * n);             //语句 4
}
```

在该函数 fun(n)求解过程中,直接调用 fun(n－1)(语句 4)即自身,所以它是一个直接递归函数。又由于递归调用是最后一条语句,所以它又属于尾递归。

递归算法通常把一个大的复杂问题层层转化为一个或多个与原问题相似的规模较小的问题来求解,只需少量的代码就可以描述出解题过程所需要的多次重复计算,大大减少了算法的代码量。

一般来说,能够用递归解决的问题应该满足以下三个条件:

(1) 需要解决的问题可以转化为一个或多个子问题来求解,而这些子问题的求解方法与原问题完全相同,只是在数量规模上会有所不同;

(2) 递归调用的次数必须是有限的;

(3) 必须有结束递归的条件来终止递归。

5.1.2 何时使用递归

在以下三种情况下,常常要用到递归的方法。

1. 定义是递归的

有许多数学公式、数列等的定义是递归的。例如,n! 和 Fibonacci 数列等。求解这些问题可以将其递归定义直接转化为对应的递归算法。例如,求 n! 可以将其转化为例 5.1 的递归算法。求 Fibonacci 数列的递归算法如下:

```
int Fib(int n)
{   if (n== 1 ‖ n== 2)
        return(1);
    else
        return(Fib(n - 1) + Fib(n - 2));
}
```

2. 数据结构是递归的

有些数据结构是递归的。例如,第 2 章中介绍过的单链表就是一种递归数据结构,其节点类型定义如下:

```
typedef struct LNode
{    ElemType data;
     struct LNode * next;
} LinkList;
```

该定义中,结构体 LNode 的定义中用到了它自身,即指针域 next 是一种指向自身类型的指针,所以它是一种递归数据结构。

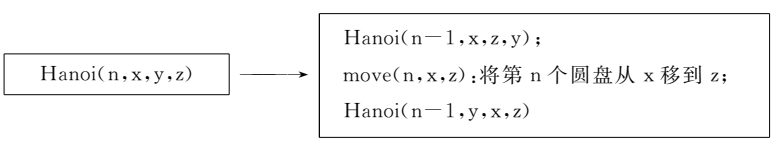
对于递归数据结构,采用递归的方法编写算法既方便又有效。例如,求一个不带头节点的单链表 L 的所有 data 域(假设为 int 型)之和的递归算法如下:

```
int Sum(LinkList *L)
{   if (L== NULL)
        return 0;
    else
        return(L->data + Sum(L->next));
}
```

3. 问题的求解方法是递归的

有些问题的解法是递归的，典型的有 Hanoi 问题，该问题的描述是：设有 3 个分别命名为 X，Y 和 Z 的塔座，在塔座 X 上有 n 个直径各不相同，从小到大依次编号为 1，2，…，n 的盘片，现要求将 X 塔座上的 n 个盘片移到塔座 Z 上并仍按同样顺序叠放，盘片移动时必须遵守以下规则：每次只能移动一个盘片；盘片可以插在 X，Y 和 Z 中任一塔座；任何时候都不能将一个较大的盘片放在较小的盘片上。设计递归求解算法。

设 Hanoi(n，X，Y，Z)表示将 n 个盘片从 X 通过 Y 移动到 Z 上，递归分解的过程是：



由此得到 Hanoi()递归算法如下：

```
void Hanoi(int n,char X,char Y,char Z)
{   if (n== 1) printf("\t 将第 %d 个盘片从 %c 移动到 %c\n",n,X,Z);
    else
    {   Hanoi(n－1,X,Z,Y);
        printf("\t 将第 %d 个盘片从 %c 移动到 %c\n",n,X,Z);
        Hanoi(n－1,Y,X,Z);
    }
}
```

5.1.3　递归模型

递归模型是递归算法的抽象，它反映一个递归问题的递归结构，例如，例 5.1 的递归算法对应的递归模型如下：

```
f(n) = 1                  n = 1
f(n) = n * f(n－1)        n > 1
```

其中，第一个式子给出了递归的终止条件，第二个式子给出了 f(n)的值与 f(n－1)的值之间的关系，第一个式子称为递归出口，第二个式子称为递归体。

一般地，一个递归模型是由递归出口和递归体两部分组成，前者确定递归何时结束，即指出明确的递归结束条件，后者确定递归求解时的递推关系。递归出口的一般格式如下：

$$f(s_1)=m_1 \tag{5.1}$$

这里的 s_1 与 m_1 均为常量。有些递归问题可能有几个递归出口。递归体的一般格式如下：

$$f(s_{n+1})=g(f(s_i),f(s_{i+1}),\cdots,f(s_n),c_j,c_{j+1},\cdots,c_m) \tag{5.2}$$

其中,n,i,j,m 均为正整数。这里的 s_{n+1}是一个递归“大问题”,s_i,s_{i+1},…,s_n 为递归“小问题”,c_j,c_{j+1},…,c_m 是若干个可以直接(用非递归方法)解决的问题,g 是一个非递归函数,可以直接求值。

实际上,递归思路是把一个不能或不好直接求解的“大问题”转化成一个或几个“小问题”来解决,再把这些“小问题”进一步分解成更小的“小问题”来解决,如此分解,直至每个“小问题”都可以直接解决(此时分解到递归出口)。但递归分解不是随意地分解,递归分解要保证“大问题”与“小问题”相似,即求解过程与环境都相似。

为了讨论方便,简化上述递归模型为:

$$f(s_1) = m_1 \tag{5.3}$$

$$f(s_n) = g(f(s_{n-1}),c_{n-1}) \tag{5.4}$$

求 $f(s_n)$的分解过程如下:

$$\begin{array}{c} f(s_n) \\ \downarrow \\ f(s_{n-1}) \\ \downarrow \\ \cdots \\ \downarrow \\ f(s_2) \\ \downarrow \\ f(s_1) \end{array}$$

一旦遇到递归出口,分解过程结束,开始求值过程。所以分解过程是“量变”过程,即原来的“大问题”在慢慢变小,但尚未解决,遇到递归出口后,便发生了“质变”,即原递归问题转化成了直接问题。求值过程如下:

$$\begin{array}{l} f(s_1)=m_1 \\ \quad\downarrow \\ f(s_2)=g(f(s_1),c_1) \\ \quad\downarrow \\ f(s_3)=g(f(s_2),c_2) \\ \quad\downarrow \\ \quad\cdots \\ \quad\downarrow \\ f(s_n)=g(f(s_{n-1}),c_{n-1}) \end{array}$$

这样 $f(s_n)$便计算出来了。因此,递归的执行过程由分解和求值两部分构成。

5.1.4 递归与数学归纳法

从递归体可以看出,如果已知 s_i,s_{i+1},…,s_n,就可以确定 s_{n+1}。从数学归纳法的角度来看,这相当于数学归纳法归纳步骤的内容。但仅有这个关系,还不能确定这个数列,若要使它完全确定,还应给出这个数列的初始值 s_1,这相当于数学归纳法基础的内容。

例如,采用数学归纳法证明下式:

$$1+2+\cdots+n=\frac{n(n+1)}{2}$$

当 n=1 时，左式=1，右式=$\frac{1\times 2}{2}$=1，左右两式相等，等式成立。

假设当 n=k−1 时等式成立，有 1+2+…+(k−1)=$\frac{k(k-1)}{2}$

当 n=k 时，左式=1+2+…+k=1+2+…+(k−1)+k=$\frac{k(k-1)}{2}$+k=$\frac{k(k+1)}{2}$

等式成立。即证。

数学归纳法是一种论证方法，而递归是算法和程序设计的一种实现技术，数学归纳法是递归的基础。

5.2　递归调用的实现原理

从前面的例子看出，递归函数直接(或间接)调用自身。但如果仅有这些操作，那么将会由于无休止地调用而引起死循环。因此，一个正确的递归程序虽然每次调用的是相同的子程序，但它的参量、输入数据等均有变化，并且在正常的情况下，随着调用的不断深入，必定会在调用到某一层函数时，不再执行递归调用而终止函数的执行(遇到递归出口，如例 5.1 中的语句 1)。

递归调用是函数嵌套调用的一种特殊情况，即它是调用自身代码。因此，也可以把每一次递归调用理解成调用自身代码的一个复制件。由于每次调用时，它的参量和局部变量均不相同，因而也就保证了各个复制件执行时的独立性。

但这些调用在内部实现时，并不是每次调用真的去复制一个复制件存放到内存中，而是采用代码共享的方式，也就是说它们都是调用同一个函数的代码，而系统为每一次调用开辟一组存储单元，用来存放本次调用的返回地址以及被中断的函数的参量值。这些单元以栈的形式存放，每调用一次进栈一次，当返回时执行出栈操作，把当前栈顶保留的值送回相应的参量中进行恢复，并按栈顶中的返回地址，从断点继续执行。下面通过计算 fun(5)的值，介绍递归调用过程实现的内部机理。

表 5.1 中是按例 5.1 算法求解 fun(5)的递归调用过程中，程序执行及栈的变化情况。fun(5)的执行是从表 5.1 的序号 1 开始执行语句 1、3、4，当遇到其中的 fun(5−1)时，必须中断当前执行的程序，转去执行 fun(4)，此时要把当前断点的地址 d1 以及参量 5 进栈，进入序号 2 的 fun(4)的执行，……当执行到 fun(1)时，因为 n=1，转入递归出口，程序执行完 1、2、5 语句后，得到 fun(1)=1，返回时，从当前栈顶获得返回地址 d4，并将参量恢复为 2，退去栈顶元素，从断点 d4 处继续执行 fun(3)的语句 4、5、…，如此继续执行，直到最后求得 fun(5)=120。从以上过程可以得出：

(1) 每递归调用一次，就需进栈一次，进栈次数称为递归深度，当 n 越大，递归深度越深，开辟的栈空间也越大。

(2) 每当遇到递归出口或完成本次执行时，需从栈顶元素中得到返回地址，并恢复参量值，当全部执行完毕时，栈应为空。

归纳起来，递归调用的实现是分两步进行的，第一步是分解过程，即用递归体将“大问题”分解成“小问题”，直到遇到递归出口为止；第二步是求值过程，即已知“小问题”，计算“大问题”。前面的 fun(5)求解过程如图 5.1 所示。对于复杂的递归调用则是这两步的循

环反复,直到求出最终值。

算法执行中最长的递归调用的链的长度称为该算法的递归调用深度。例如,求 n! 对应的递归算法在求 fun(5)时递归调用深度是 4(如图 5.1 所示)。

表 5.1　fun(5)的执行过程

序号	调用	执行语句	中断地址	进出栈	栈内情况 返址　n	继续执行	说明
1	fun(5)	1,3,4	d1;fun(4)	进栈	d1　5	fun(4)	
2	fun(4)	1,3,4	d2;fun(3)	进栈	d2　4 d1　5	fun(3)	
3	fun(3)	1,3,4	d3;fun(2)	进栈	d3　3 d2　4 d1　5	fun(2)	
4	fun(2)	1,3,4	d4;fun(1)	进栈	d4　2 d3　3 d2　4 d1　5	fun(1)	
5	fun(1)	1,2	返回 d4	出栈	d3　3 d2　4 d1　5	fun(2)	求得 fun(1)=1
6	fun(2)	4	返回 d3	出栈	d2　4 d1　5	fun(3)	求得 fun(2)=2
7	fun(3)	4	返回 d2	出栈	d1　5	fun(4)	求得 fun(3)=6
8	fun(4)	4	返回 d1	出栈	栈空	fun(5)	求得 fun(4)=24
9	fun(5)	4				结束	求得 fun(5)=120

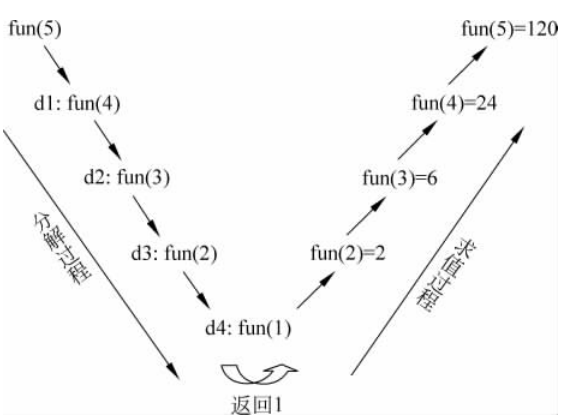


图 5.1　fun(5)求值过程

5.3　递归算法的设计

5.3.1　递归算法设计的步骤

递归算法求解过程是：先将整个问题划分为若干个子问题，通过分别求解子问题，最后获得整个问题的解。这种自上而下将问题分解、求解，再自下而上引用、合并，求出最后解答的过程称为递归求解过程。这是一种分而治之的算法设计方法。

递归算法设计先要给出递归模型，再转换成对应的 C/C++函数。

对于式(5.3)和式(5.4)简化的递归模型而言，要求解 $f(s_n)$，不是直接求其解，而是转化为计算 $f(s_{n-1})$和一个常量 c_{n-1}，求解 $f(s_{n-1})$的方法与环境和求解 $f(s_n)$的方法与环境是相似的，但$f(s_n)$是一个“大问题”，而 $f(s_{n-1})$是一个“较小问题”，尽管 $f(s_{n-1})$还未解决，但向解决目标靠近了一步，这就是一个“量变”，如此到达递归出口时，便发生了“质变”，即递归问题解决了。因此，递归设计就是要给出合理的“较小问题”，然后确定“大问题”的解与“较小问题”之间的关系，即确定递归体；最后朝此方向分解，必然有一个简单基本问题解，以此作为递归出口。由此得出递归设计的步骤如下：

(1) 对原问题 $f(s_n)$进行分析，假设出合理的“较小问题”$f(s_{n-1})$(与数学归纳法中假设 i＝n－1 时等式成立相似)；

(2) 假设 $f(s_{n-1})$是可解的，在此基础上确定 $f(s_n)$的解，即给出 $f(s_n)$与 $f(s_{n-1})$之间的关系(与数学归纳法中求证 i＝n 时等式成立的过程相似)；

(3) 确定一个特定情况(如 f(1)或 f(0))的解，由此作为递归出口(与数学归纳法中求证 i＝1 或 i＝0 时等式成立相似)。

例如，采用递归算法求实数数组 A[0..n－1]中的最小值。

假设 f(A,i)函数求数组元素 A[0]～A[i](共 i＋1 个元素)中的最小值。当 i＝0 时，有 f(A,i)＝A[0]；假设 f(A,i－1)已求出，则 f(A,i)＝MIN(f(A,i－1),A[i])，其中 MIN()为求两个值中较小值的函数。因此得到如下递归模型：

$$f(A,i)=\begin{cases} A[0] & \text{当 } i=0 \text{ 时} \\ MIN(f(A,i-1),A[i]) & \text{其他情况} \end{cases}$$

由此得到如下递归求解算法：

```
double f(float A[],int i)
{   double m;
    if (i== 0) return A[0];
    else
    {   m = f(A,i - 1);
        if (m > A[i]) return(A[i]);
        else return(m);
    }
}
```

下面通过两个例子进一步说明递归算法设计过程。

【例 5.2】 利用串的基本运算写出对串求逆的递归算法。

解：经分析，求逆串的递归模型如下：

$$f(s)=\begin{cases} s & \text{若 } s=\Phi \\ Concat(f(SubStr(s,2,StrLength(s)-1)),SubStr(s,1,1)) & \text{其他情况} \end{cases}$$

其递归思路是：对于 s="$s_1 s_2 \cdots s_n$"的串，假设"$s_2 s_3 \cdots s_n$"已求出其逆串即 f(SubStr(s,2,StrLength(s)-1))，再将 s_1(为 SubStr(s,1,1))连接到最后即得到 s 的逆串。对应的算法如下：

```
SqString invert(SqString &s)
{   SqString s1,s2;
    if (StrLength(s)> 0)
    {   s1 = invert(SubStr(s,2,StrLength(s) - 1));
        s2 = Concat(s1,SubStr(s,1,1));
    }
    else
        StrCopy(s2,s);
    return s2;
}
```

【例 5.3】 求顺序表$\{a_1,a_2,\cdots,a_n\}$中最大元素。

解：将线性表分解成$\{a_1,a_2,\cdots,a_m\}$和$\{a_{m+1},\cdots,a_n\}$两个子表，分别求得子表中的最大元素 a_i 和 a_j，比较 a_i 和 a_j 求较大者，就可以求得整个线性表的最大元素。而求解子表中的最大元素方法与求解总表相同，即再分别将它们分成两个更小的子表，如此不断分解，直到表中只有一个元素为止(当只有一个元素时，该元素便是该表的最大元素)。对应的算法如下：

```
ElemType Max(SqList L,int i,int j)          //求顺序表 L 中最大元素
{   int mid;
    ElemType max,max1,max2;
    if (i== j)
        max = L.data[i];                    //递归出口
    else
    {   mid = (i + j)/2;
        max1 = Max(L,i,mid);                //递归调用 1
        max2 = Max(L,mid + 1,j);            //递归调用 2
        max = (max1 > max2)?max1:max2;
    }
    return(max);
}
```

5.3.2 递归数据结构的递归算法设计

采用递归方式定义的数据结构称为递归数据结构。在递归数据结构定义中包含的递归运算称为基本递归运算。

例如，正整数的定义为：1 是正整数，若 n 是正整数(n≥1)，则 n+1 也是正整数。从中看出，正整数是一种递归数据结构。显然，若 n 是正整数(n>1)，则 m=n-1 也是正整数，也就是说，对于大于 1 的正整数 n，n-1 是一种递归运算。

所以求 n! 的算法中，递归体 f(n)＝n * f(n－1)是可行的，因为对于大于 1 的 n，n 和 n－1 都是正整数。

对于递归数据结构

RD = (D, Op)

其中，D＝{d_i}(1≤i≤n，共 n 个元素)为构成该数据结构的所有元素的集合，Op 是递归运算，Op＝{op_j}(1≤j≤m，共 m 个运算)，不妨设 op_j 为一元运算符，∀ d_i∈D，应有 $op_j(d_i)$∈D，也就是说，递归运算符具有封闭性。

对于上述正整数的定义，D 是正整数的集合(对于固定位数的计算机，所能表示的正整数是有限的)，Op＝{op_1，op_2}由基本递归运算符构成，op_1 和 op_2 的定义如下：

$op_1(n) = n - 1$

其中，n 为大于 1 的正整数。

$op_2(n) = n + 1$

其中，n 为大于等于 1 的正整数。

对于不带头节点的单链表，其节点类型为 LinkList，每个节点的 next 域为 LinkList 类型的指针。这样的单链表通过首节点指针来标识。采用递归数据结构的定义如下：

SL = (D, Op)

其中，D 是由部分或全部节点构成的单链表的集合(含空单链表)，Op＝{op_1}：

$op_1(L) = L->next$　　L 为含一个或一个以上节点的单链表

显然这个基本递归运算符是一元运算符，且具有封闭性。

实际上，递归算法设计步骤中第 1 步和第 2 步是用于确定递归模型中的递归体。在假设原问题 f(s)合理的“较小问题”f(s')时，需要考虑递归数据结构的递归运算。例如，在设计不带头节点的单链表的递归算法时，通常设 s 为以 L 为首节点指针的整个单链表，s'为除首节点外余下节点构成的单链表(由 L－＞next 标识，而该运算为递归运算)。所以在设计递归算法时，如果处理的数据是递归数据结构，要对该数据结构及其递归运算进行分析，找出正确的递归体。

【例 5.4】 假设有一个不带头节点的单链表 L，设计一个算法释放其中所有节点。

解：设 f(L)的功能是释放 a_1 到 a_n 的所有节点，则 f(L－＞next)的功能是释放 a_2 到 a_n 的所有节点，如图 5.2 所示。假设 f(L－＞next)是可实现的，则 f(L)的功能是先调用 f(L－＞next)，然后释放 L 所指节点。对应的递归模型如下：

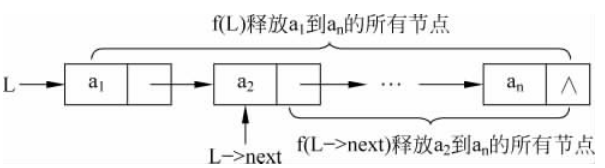


图 5.2　一个不带头节点的单链表

f(L)≡不做任何事件　　当L=NULL时
f(L)≡f(L->next);释放*L节点　　其他情况

其中,“≡”表示功能等价关系。对应的算法如下:

```
void release(LinkList *L)
{   if (L!=NULL)
    {   release(L->next);
        free(L);
    }
}
```

说明:在对单链表设计递归算法时,通常采用不带头节点的单链表。以图5.2为例,L->next表示的单链表一定是不带头节点的,也就是说“小问题”的单链表是不带头节点的单链表,所以“大问题”即整个单链表也应设计成不带头节点的单链表。

5.3.3 递归求解方法的递归算法设计

当求解问题的方法是递归(如Hanoi问题)的或者可以转换成递归方法求解时(如皇后问题),可以设计成递归算法。

例如,求f(n)=1+2+…+n(n≥1),这个问题可以转化为用递归方法求解,假设“小问题”f(n-1)=1+2+…+(n-1)是可求的,则f(n)=f(n-1)+n。

对于采用递归方法求解的问题,需要对问题本身进行分析,确定大、小问题解之间的关系,构造合理的递归体。

【例5.5】 采用递归算法求解迷宫问题,并输出从入口到出口的所有迷宫路径。

解:迷宫问题在第3章介绍过,设mgpath(int xi,int yi,int xe,int ye,PathType path)是求从(xi,yi)到(xe,ye)的迷宫路径,用path变量保存迷宫路径,其中PathType类型定义如下:

```
typedef struct
{   int i;                          //当前方块的行号
    int j;                          //当前方块的列号
} Box;
typedef struct
{   Box data[MaxSize];
    int length;                     //路径长度
} PathType;                         //定义路径类型
```

当从(xi,yi)方块找到一个可走相邻方块(i,j)后,mgpath(i,j,xe,ye,path)表示求从(i,j)到(xe,ye)的一条迷宫路径。显然,mgpath(xi,yi,xe,ye,path)是“大问题”,而mgpath(i,j,xe,ye,path)是“小问题”。求解迷宫问题的递归模型如下:

mgpath(xi,yi,xe,ye,path)≡将(xi,yi)添加到path中;输出path中的迷宫路径;
　　若(xi,yi)=(xe,ye)

mgpath(xi,yi,xe,ye,path)≡将(xi,yi)添加到path中;
　　找出(xi,yi)四周的一个相邻方块(i,j);
　　mg[xi][yi]=-1;

mgpath(i,j,xe,ye,path);
path 回退一步并置 mg[xi][yi]=0;
若(xi,yi)不为出口且可走

上述递归模型中,当完成“小问题”mgpath(i,j,xe,ye,path)后将 path 回退并置 mg[x_i][y_i]为 0,其目的是恢复前面求迷宫路径中的环境,以便找出所有的迷宫路径。对应的递归算法如下:

```
void mgpath(int xi,int yi,int xe,int ye,PathType path)
//求解路径为:(xi,yi)->(xe,ye)
{   int di,k,i,j;
    if (xi==xe && yi==ye)                    //找到了出口,输出路径
    {   path.data[path.length].i = xi;
        path.data[path.length].j = yi;
        path.length++;
        printf("迷宫路径%d 如下:\n",++count);
        for (k=0;k<path.length;k++)
        {   printf("\t(%d,%d)",path.data[k].i,path.data[k].j);
            if ((k+1)%5==0)                  //每输出每 5 个方块后换一行
               printf("\n");
        }
        printf("\n");
    }
    else                                     //(xi,yi)不是出口
    {   if (mg[xi][yi]==0)                   //(xi,yi)是一个可走方块
        {   di=0;
            while (di<4)                     //找(xi,yi)的一个相邻方块(i,j)
            {   path.data[path.length].i = xi;
               path.data[path.length].j = yi;
               path.length++;                //路径长度增 1
               switch(di)
               {
               case 0:i=xi-1; j=yi; break;
               case 1:i=xi; j=yi+1; break;
               case 2:i=xi+1; j=yi; break;
               case 3:i=xi; j=yi-1; break;
               }
               mg[xi][yi]=-1;                //避免重复找路径
               mgpath(i,j,xe,ye,path);
               mg[xi][yi]=0;                 //恢复(xi,yi)为可走
               path.length--;                //回退一个方块
               di++;
            }
        }
    }
}
```

本算法可以输出所有的迷宫路径,可以通过比较找出最短路径(可能存在多条最短路径)。

本章小结

本章基本学习要点如下:

(1) 理解递归的定义和递归模型。
(2) 重点掌握递归的执行过程。
(3) 掌握递归设计的一般方法。
(4) 灵活运用递归算法解决一些较复杂应用问题。

练习题 5

5.1 已知 A[n]为整数数组,编写一个递归算法求其中 n 个元素的平均值。

5.2 有一个不带表头节点的单链表,其节点类型如下:

```
typedef int ElemType;
typedef struct node
{   ElemType data;
    struct node *next;
} Node;
```

设计如下递归算法:
(1) 求以 h 为头指针的单链表的节点个数;
(2) 正向显示以 h 为头指针的单链表的所有节点值;
(3) 反向显示以 h 为头指针的单链表的所有节点值;
(4) 删除以 h 为头指针的单链表中值为 x 的第一个节点;
(5) 删除以 h 为头指针的单链表中值为 x 的所有节点;
(6) 输出以 h 为头指针的单链表中最大节点值;
(7) 输出以 h 为头指针的单链表中最小节点值。

上机实验题 5

实验题 5.1 编写一个程序 exp5-1.cpp,求解皇后问题:在 n×n 的方格棋盘上,放置 n 个皇后,要求每个皇后不同行、不同列、不同左右对角线。

要求:(1) 皇后的个数 n 由用户输入,其值不能超过 20,输出所有的解。(2) 采用递归方法求解。

实验题 5.2 编写一个程序 exp5-2.cpp,求解背包问题:设有不同价值、不同重量的物品 n 件,求从这 n 件物品中选取一部分物品的方案,使选中物品的总重量不超过指定的限制重量,但选中物品的总价值最大。