第 5 章 递归

本章思政

在算法设计中经常需要用递归方法求解，特别是后面的树和二叉树、图、查找及排序等章节中大量地用到了递归算法。递归是计算机科学中的一个重要工具，很多程序设计语言（例如 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**(recursion)。若调用自身,称为**直接递归**(direct recursion)。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**(indirect recursion)。在算法设计中,任何间接递归算法都可以转换为直接递归算法来实现,所以后面主要讨论直接递归。

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

扫一扫

视频讲解

(1) 递归数列指的是由递归关系所确定的数列。

(2) 递归过程指的是直接或间接调用自身的过程。

(3) 递归算法指的是包含递归过程的算法。

(4) 递归程序指的是直接或间接调用自身的程序。

(5) 递归方法指的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的元素进行运算,以确定一系列元素(例如数或函数)的方法。

【例 5.1】 根据求阶乘的定义,给出求 $n!$(n 为正整数)的递归函数。

解 求 $n!$ 的定义是 $1!=1$,$n!=n*(n-1)!$,对应的递归函数 fact(n)如下:

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

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,递归策略只需少量的代码就可以描述出解题过程中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3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。

递归算法的优点是结构简单、清晰,易于阅读,方便其正确性证明;缺点是算法执行中占用的内存空间较多,执行效率低,不容易优化。

5.1.2 何时使用递归

在以下 3 种情况下经常要用到递归方法。

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,求 n!和 Fibonacci 数列等。对于这些问题的求解可以将其递归定义直接转化为对应的递归算法。例如,求 n!可以转化为例 5.1 的递归算法。求 Fibonacci 数列的递归算法如下:

扫一扫

视频讲解

```
int Fib1(int n)                    //求 Fibonacci 数列的第 n 项
{   if (n==1 || n==2)
        return(1);
    else
        return(Fib1(n-1)+Fib1(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 章中介绍的单链表就是一种递归数据结构,其结点类型声明如下:

```
typedef struct LNode
{    ElemType data;              //存放结点数据
     struct LNode *next;         //指向下一个同类型结点的指针
} LinkNode;                      //单链表的结点类型
```

其中,结构体 LNode 的声明用到了它自身,即指针域 next 是一种指向相同类型结点的指针,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。例如求一个不带头结点的单链表 L 的所有 data 域(假设为 int 类型)之和的递归算法如下:

```
int Sum(LinkNode *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 问题的求解,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 个直径各不相同的盘片,从小到大依次编号为 1、2、…、n。现要求将 X 塔座上的这 n 个盘片移到塔座 Z 上并仍按同样的顺序叠放,移动盘片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 中的任一塔座上;任何时候都不能将一个较大的盘片放在较小的盘片上。图 5.1 所示为 $n=4$ 时的 Hanoi 问题。设计求解该问题的算法。

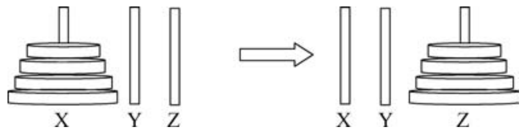


图 5.1　Hanoi 问题($n=4$)

Hanoi 问题特别适合采用递归方法来求解。设 Hanoi(n,x,y,z)表示将 n 个盘片从 x 塔座借助 y 塔座移到 z 塔座上,递归分解的过程如下:

Hanoi(n, x, y, z) ⟹ Hanoi(n−1, x, z, y);
move(n, x, z): 将第 n 个盘片从 x 移到 z;
Hanoi(n−1, y, x, z)

其含义是首先将 x 塔座上的 $n-1$ 个盘片借助 z 塔座移到 y 塔座上;此时 x 塔座上只有一个盘片,将其直接移到 z 塔座上;再将 y 塔座上的 $n-1$ 个盘片借助 x 塔座移到 z 塔座上。由此得到 Hanoi1 递归算法如下:

```
void Hanoi1(int n, char X, char Y, char Z)
{   if (n==1)                                      //只有一个盘片的情况
        printf("\t 将第%d 个盘片从%c 移动到%c\n", n, X, Z);
    else                                           //有两个或多个盘片的情况
    {   Hanoi1(n-1, X, Z, Y);
        printf("\t 将第%d 个盘片从%c 移动到%c\n", n, X, Z);
        Hanoi1(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:

$$f(n)=1 \qquad n=1$$
$$f(n)=n*f(n-1) \qquad n>1$$

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,把第一个式子称为递归出口,把第二个式子称为递归体。

一般情况下,一个递归模型由递归出口和递归体两部分组成。**递归出口**(recursive exit)确定递归到何时结束,即指出明确的递归结束条件。**递归体**(recursive body)确定递归求解时的递推关系。

扫一扫

视频讲解

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:

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

其中，n、i、j、m 均为正整数。这里的 s_n 是一个递归“大问题”，s_i、s_{i+1}、…、s_{n-1} 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，如图 5.2 所示，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，直到每个“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。

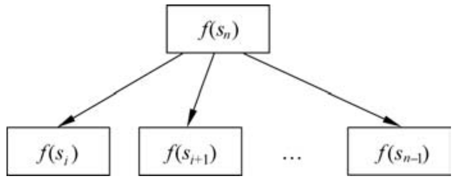


图 5.2　把大问题 $f(s_n)$ 转化成几个小问题来解决

为了讨论方便，设简化的递归模型（即将一个“大问题”分解为一个“小问题”）如下：

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

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

在求 $f(s_n)$ 时的分解过程如下：

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

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”，即原递归问题便转化成直接问题。上面的求值过程如下：

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

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，分解部分就是用递归体将“大问题”分解成“小问题”，直到递归出口为止，然后进行求值过程，即已知

“小问题”计算“大问题”。前面的 fun(5)的求解过程如图 5.3 所示。

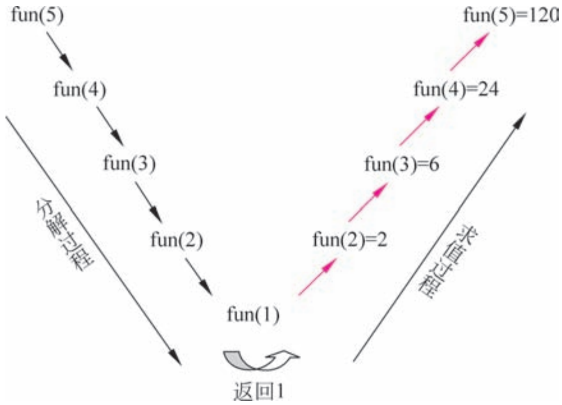


图 5.3 fun(5)的求值过程

在递归算法的执行中，最长的递归调用的链长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 5。

对于复杂的递归算法，其执行过程可能需要循环反复地分解和求值才能获得最终解。例如，对于前面求 Fibonacci 数列的 Fib1 算法，求 Fib1(6)的过程构成的递归树如图 5.4 所示，向下的实箭头表示分解，向上的虚箭头表示求值，每个方框旁边的数字是该方框的求值结果，最后求得 Fib1(6)为 8。该递归树的高度为 5，所以递归调用深度也是 5。

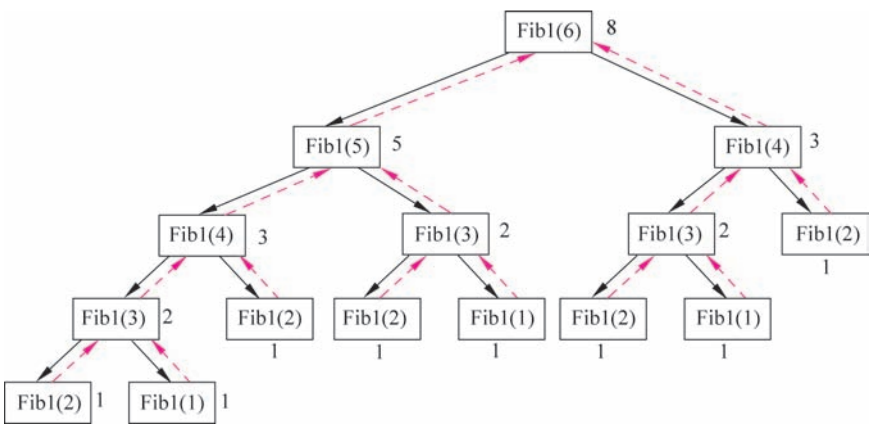


图 5.4 求 Fib(6)对应的递归树

扫一扫

视频讲解

5.1.4 递归与数学归纳法

从递归体可以看到，如果已知 s_i、s_{i+1}、…、s_{n-1}，就可以确定 s_n。从数学归纳法的角度来看，这相当于数学归纳法的归纳步骤的内容。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。

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

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

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，数学归纳法是递归求解问题的理论基础。可以说递归的思想来自数学归纳法。

5.2 栈和递归

5.2.1 函数调用栈

扫一扫

视频讲解

函数调用操作包括从一块代码到另一块代码之间的双向数据传递和执行控制转移。数据传递通过函数参数和返回值实现，另外还需要在进入函数时为函数的局部变量分配存储空间，并且在退出函数时收回这部分空间。

大多数 CPU 上的程序实现使用栈来支持函数调用操作。单个函数调用操作所使用的函数调用栈被称为**栈帧**(stack frame)结构。每次函数调用时都会相应地创建一帧，保存返回地址、函数实参和局部变量值等，并将该帧压入调用栈。若在该函数返回之前又发生了新的调用，则同样要将与新函数对应的一帧进栈，成为栈顶。函数一旦执行完毕，对应的帧便出栈，控制权交还给该函数的上层调用函数，并按照该帧中保存的返回地址确定程序中继续执行的位置。

例如，若有以下程序：

```
int main()
{   int m,n;
    …
    f(m,n);                        //后面第一个语句的地址为 d1
    …
    return 1;
}
void f(int s,int t)
{       int i;
        …
        g(i);                      //后面第一个语句的地址为 d2
        …
}
void g(int d)
{       int x,y;
        …
}
```

在执行上述程序时，假设 main 函数的返回地址为 d_0。当执行 main 函数时，将栈帧①进栈。在 main 函数中调用 f 函数时，将栈帧②进栈。在 f 函数中调用 g 函数时，将栈帧③进栈，如图 5.5 所示。当 g 函数执行完毕，将栈帧③退栈，控制权交回到 f 函数，转向其中的 d_2 地址继续执行，其余执行过程类似。

	返回地址	实参	局部变量	
栈顶	d_2	i	x, y	栈帧③
	d_1	m, n	i	栈帧②
	d_0		m, n	栈帧①

图 5.5　函数调用栈

5.2.2　递归调用的实现

递归是函数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数和局部变量可能不相同，所以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正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，也就是它们都是调用同一个函数的代码，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数值。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数中进行恢复，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。

扫一扫

视频讲解

表 5.1 给出了求解 fun(5)的递归调用过程中程序的执行及栈的变化情况，设调用 fun(5)的返回地址为 d_0。

在调用 fun(5)时，先把返回地址 d_0 以及参数 5 进栈，然后执行语句 1、3、4，当遇到其中的 fun(5-1)(即 fun(4))时，必须中断当前执行的程序，转去调用 fun(4)，记录下其返回地址为 d_1；在调用 fun(4)时，先把返回地址 d_1 以及参数 4 进栈，然后执行语句 1、3、4，当遇到其中的 fun(3)时转去调用 fun(3)，记录下其返回地址为 d_2；…。直到调用 fun(1)，把返回地址 d_4 以及参数 1 进栈。

表 5.1　fun(5)的执行过程

序号	调用/执行	返回地址	进/出栈	栈内情况		执行语句	说明
				返回地址	实参		
1	调用 fun(5)	d_0	进栈	d_0	5	1,3,4	
2	调用 fun(4)	d_1	进栈	d_1 d_0	4 5	1,3,4	
3	调用 fun(3)	d_2	进栈	d_2 d_1 d_0	3 4 5	1,3,4	

续表

序号	调用/执行	返回地址	进/出栈	栈内情况		执行语句	说明
				返回地址	实参		
4	调用 fun(2)	d_3	进栈	d_3 d_2 d_1 d_0	2 3 4 5	1,3,4	
5	调用 fun(1)	d_4	进栈	d_4 d_3 d_2 d_1 d_0	1 2 3 4 5	1,3,4	
6	执行 fun(1)	返回 d_4	出栈	d_3 d_2 d_1 d_0	2 3 4 5	1,2	求得 fun(1)=1
7	执行 fun(2)	返回 d_3	出栈	d_2 d_1 d_0	3 4 5	4	求得 fun(2)=2
8	执行 fun(3)	返回 d_2	出栈	d_1 d_0	4 5	4	求得 fun(3)=6
9	执行 fun(4)	返回 d_1	出栈	d_0	5	4	求得 fun(4)=24
10	执行 fun(5)	返回 d_0		栈空		4	求得 fun(5)=120

然后执行 fun(1),执行语句 1、2,返回 1 并出栈一次;执行 fun(2),执行语句 4,返回 2 并出栈一次;…。直到执行 fun(5),此时栈空,返回 120 并转向 d_0。

例如,已知程序如下:

```
int S(int n)
{    return (n<=0) ? 0 : S(n-1)+n; }
int main()
{    printf("%d\n",S(1));
     return 1;
}
```

程序执行时使用一个栈来保存调用过程的信息,这些信息用 main()、S(0)和 S(1)表示,那么自栈底到栈顶保存的信息的顺序是怎样的?

首先从 main()开始执行程序,将 main()信息进栈,遇到调用 S(1),将 S(1)信息进栈,在执行递归函数 S(1)时又遇到调用 S(0),再将 S(0)信息进栈。所以,自栈底到栈顶保存的

信息的顺序是 main()→S(1)→S(0)。

5.2.3 递归算法的时空性能分析

1. 递归算法的时间复杂度分析

扫一扫 视频讲解

递归算法分析不能简单地采用非递归算法分析的方法，递归算法分析属于变长时空分析，非递归算法分析属于定长时空分析。

在递归算法分析中，首先写出对应的递推式，然后求解递推式得出算法的执行时间或者空间。例如，对于前面求 Hanoi 问题的递归算法，分析其时间复杂度的过程如下。

设 Hanoi1(n,x,y,z)的执行时间为 $T(n)$，则两个问题规模为 $n-1$ 的子问题的执行时间均为 $T(n-1)$，总时间是累加关系。对应的递推式如下：

$$
\begin{aligned}
&T(n)=1 && \text{当 } n=1 \text{ 时}\\
&T(n)=2T(n-1)+1 && \text{当 } n>1 \text{ 时}
\end{aligned}
$$

则：

$$
\begin{aligned}
T(n)&=2T(n-1)+1=2(2T(n-2)+1)+1\\
&=2^2T(n-2)+2+1=2^2(2T(n-3)+1)+2+1\\
&=2^3T(n-3)+2^2+2+1\\
&=\cdots\\
&=2^{n-1}T(1)+2^{n-2}+\cdots+2^2+2+1\\
&=2^n-1=O(2^n)
\end{aligned}
$$

【例 5.2】 有如下递归算法：

```
void fun(int a[],int n,int k)                //数组 a 共有 n 个元素,执行时间为 T1(n,k)
{    int i;
     if (k==n-1)
     {    for (i=0;i<n;i++)
               printf("%d\n",a[i]);          //该语句的执行次数为 n
     }
     else
     {    for (i=k;i<n;i++)
               a[i]=a[i]+i*i;                //该语句的执行次数为 n-k
          fun(a,n,k+1);                      //执行时间为 T1(n,k+1)
     }
}
```

调用上述算法的语句为 fun(a,n,0)，求其时间复杂度。

解 设 fun(a,n,k)的执行时间为 $T_1(n,k)$，fun(a,n,0)的执行时间为 $T(n)$。显然有 $T(n)=T_1(n,0)$。由 fun()算法得到如下执行时间的递推式：

$$
T_1(n,k)=\begin{cases} n & \text{当 } k=n-1 \text{ 时}\\ (n-k)+T_1(n,k+1) & \text{其他情况}\end{cases}
$$

则：

$$\begin{aligned} T(n) &= T_1(n,0) = n + T_1(n,1) \\ &= n + (n-1) + T_1(n,2) \\ &\ \vdots \\ &= n + (n-1) + \cdots + 2 + T_1(n,n-1) \\ &= \frac{(n+2)(n-1)}{2} + n = \frac{n^2}{2} + \frac{3n}{2} - 1 \\ &= O(n^2) \end{aligned}$$

所以调用 fun(a,n,0)的时间复杂度为 $O(n^2)$。

2. 递归算法的空间复杂度分析

递归算法执行中使用了系统栈空间，因此需要根据递归调用的深度来分析递归算法的空间复杂度，其过程与递归算法的时间复杂度分析类似。例如，对于前面求 Hanoi 问题的递归算法，分析其空间复杂度的过程如下。

设 Hanoi1(n,x,y,z)的临时空间为 $S(n)$，则两个问题规模为 $n-1$ 的子问题的临时空间均为 $S(n-1)$，总空间是最大值关系，因为前一个子问题执行完毕其空间被释放，释放的空间被后一个子问题重复使用。对应的递推式如下：

$$\begin{array}{ll} S(n)=1 & \text{当 } n=1 \text{ 时} \\ S(n)=S(n-1)+1 & \text{当 } n>1 \text{ 时} \end{array}$$

则：

$$\begin{aligned} S(n) &= S(n-1)+1 = (S(n-2)+1)+1 \\ &= \cdots \\ &= S(1)+1+\cdots+1 \\ &= \underbrace{1+1+\cdots+1}_{n\text{个}1} = O(n) \end{aligned}$$

【例 5.3】 对于例 5.2 的递归算法，分析调用语句 fun(a,n,0)的空间复杂度。

解 设 fun(a,n,k)占用的临时空间为 $S_1(n,k)$，fun(a,n,0)占用的临时空间为 $S(n)$。显然有 $S(n)=S_1(n,0)$。由 fun()算法得到如下占用临时空间的递推式：

$$S_1(n,k)=\begin{cases} 1 & \text{当 } k=n-1 \text{ 时(此时仅定义了一个临时变量 } i\text{)} \\ 1+S_1(n,k+1) & \text{其他情况} \end{cases}$$

则：

$$\begin{aligned} S(n) &= S_1(n,0) = 1 + S_1(n,1) \\ &= 1+1+S_1(n,2) \\ &\ \vdots \\ &= 1+1+\cdots+1+S_1(n,n-1) \\ &= \underbrace{1+1+\cdots+1}_{n\text{个}1} = O(n) \end{aligned}$$

所以调用 fun(a,n,0)的空间复杂度为 $O(n)$。

5.2.4 递归到非递归的转换*

通常递归算法的执行效率比较差，当递归调用层次较深时容易出现“栈溢出”，可以将递

归算法转换为等效的非递归算法，主要有两种方法，即直接转换法和间接转换法。

1. 直接转换法

所谓直接转换法就是用迭代方式或者循环语言替代多次重复的递归调用。直接转换法通常用来消除尾递归和单向递归。

扫一扫

视频讲解

如果一个递归函数中的递归调用语句是最后一条执行语句，且把当前运算结果放在参数里传给下层函数，称这种递归调用为**尾递归**(tail recursion)。例如，例 5.1 的递归函数转换为尾递归函数如下：

```
int fact1(int n,int ans)                    //求 n!的尾递归算法
{   if(n==1)
        return ans;
    else
        return fact1(n-1,n * ans);
}
```

利用上述算法求 5! 的过程是，fact1(5,1)→fact1(4,5)→fact1(3,20)→fact1(2,60) →fact1(1,120)，最后返回值 120 表示 5!=120。有些编译器针对尾递归的特点通过优化将返回地址不保存在系统栈中，从而节省栈空间开销。单向递归是指递归的求值过程总是朝着一个方向进行的，例如，前面求 Fibonacci 数列的递归算法 Fib1 就属于单向递归，因为求值过程是 1,1,2,3,5,8,…，即单向生长的。可以采用迭代方式将 Fib1 转换为如下非递归算法：

```
int Fib2(int n)                          //求 Fibonacci 数列的第 n 项
{   int a=1,b=1,i,s;
    if (n==1 || n==2)
        return 1;
    else
    {   for (i=3;i<=n;i++)
        {   s=a+b;
            a=b;
            b=s;
        }
        return s;
    }
}
```

2. 间接转换法

其他相对复杂的递归算法不能直接求值，在执行中需要回溯。可以在理解递归调用实现过程的基础上用栈来模拟递归执行过程，即使用栈保存中间结果，从而将其转换为等效的非递归算法，这称为间接转换法。

例如，在将前面求解 Hanoi 问题的递归算法 Hanoi1 转换为等价的非递归算法时，需要使用一个栈暂时存放还不能直接移动盘片的任务/子任务。

首先将任务 Hanoi(n,x,y,z)进栈，栈不空时循环：出栈一个任务 Hanoi(n,x,y,z)，如果它是可直接移动的，就移动盘片；否则该任务转化为 Hanoi($n-1$,x,z,y)、move(n,x,z)、 Hanoi($n-1$,y,x,z),按相反顺序(即将 3 个任务 Hanoi($n-1$,y,x,z)、move(n,x,z)和 Hanoi($n-1$,x,z,y))依次进栈,其中 move(n,x,z)是可直接移动的任务。为此设计一个顺序栈的类型如下:

```
typedef struct
{   int n;                          //盘片的个数
    char x,y,z;                     //3 个塔座
    bool flag;                      //可直接移动盘片时为 true,否则为 false
} ElemType;                         //顺序栈中元素的类型
typedef struct
{   ElemType data[MaxSize];         //存放元素
    int top;                        //栈顶指针
} StackType;                        //顺序栈的类型
```

栈中的每个元素对应一个求解任务,flag 标识该任务是否可以直接移动盘片。采用第 3 章的原理设计好顺序栈的基本运算算法(除了将 SqStack 改为 StackType 以外,其他代码都是相同的)。对应的求解 Hanoi 问题的非递归算法如下:

```
void Hanoi2(int n, char x, char y, char z)
{   StackType *st;                          //定义顺序栈指针
    ElemType e,e1,e2,e3;
    if (n<=0) return;                       //参数错误时直接返回
    InitStack(st);                          //初始化栈
    e. n=n; e.x=x; e.y=y; e.z=z; e.flag=false;
    Push(st,e);                             //元素 e 进栈
    while (!StackEmpty(st))                 //栈不空时循环
    {   Pop(st,e);                          //出栈元素 e
        if (e.flag==false)                  //当不能直接移动盘片时
        {   e1.n=e.n-1; e1.x=e.y; e1.y=e.x; e1.z=e.z;
            if (e1.n==1)                    //只有一个盘片时可直接移动
                e1.flag=true;
            else                            //有一个以上盘片时不能直接移动
                e1.flag=false;
            Push(st,e1);                    //处理 Hanoi(n-1,y,x,z)步骤
            e2.n=e.n; e2.x=e.x; e2.y=e.y; e2.z=e.z; e2.flag=true;
            Push(st,e2);                    //处理 move(n,x,z)步骤
            e3.n=e.n-1; e3.x=e.x; e3.y=e.z; e3.z=e.y;
            if (e3.n==1)                    //只有一个盘片时可直接移动
                e3.flag=true;
            else
                e3.flag=false;              //有一个以上盘片时不能直接移动
            Push(st,e3);                    //处理 Hanoi(n-1,x,z,y)步骤
        }
        else                                //当可以直接移动时
            printf("\t 将第%d 个盘片从%c 移动到%c\n",e.n,e.x,e.z);
    }
    DestroyStack(st);                       //销毁栈
}
```

5.3 递归算法的设计

5.3.1 递归算法的设计步骤

递归算法设计的基本步骤是先确定求解问题的递归模型，再转换成对应的 C/C++语言函数。由于递归模型反映递归问题的“本质”，所以前一步是关键，也是讨论的重点。

递归算法的求解过程是先将整个问题划分为若干个子问题，然后分别求解子问题，最后获得整个问题的解。这是一种分而治之的思路，通常由整个问题划分的若干子问题的求解是独立的，所以求解过程对应一棵递归树。如果在设计算法时就考虑递归树中的每一个分解/求值部分会使问题复杂化，不妨只考虑递归树中第 1 层和第 2 层之间的关系，即“大问题”和“小问题”的关系，其他关系与之相似。

扫一扫

视频讲解

由此得出获取求解问题递归模型(简化递归模型)的步骤如下：

(1) 对原问题 $f(s_n)$进行分析，假设出合理的小问题 $f(s_{n-1})$。

(2) 假设小问题 $f(s_{n-1})$是可解的，在此基础上确定大问题 $f(s_n)$的解，即给出 $f(s_n)$与 $f(s_{n-1})$之间的关系，也就是确定递归体(与数学归纳法中假设 $i=n-1$ 时等式成立，再求证 $i=n$ 时等式成立的过程相似)。

(3) 确定一个特定情况(例如 $f(1)$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 5.4】 采用递归算法求实数数组 $A[0..n-1]$中的最小值。

解 假设用 $f(A,i)$求数组元素 $A[0..i]$(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，显然有 $f(A,i)=\mathrm{MIN}(f(A,i-1),A[i])$，其中 MIN()为求两个值中较小值的函数。因此得到以下递归模型：

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

由此得到以下递归求解算法：

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

例如，若一个实数数组为 double $a[]=\{9.2,5.5,3.8,7.1,6.5\}$，调用 Min($a$,4)返回最小元素 3.8。

【例 5.5】 求含 $n(n>1)$ 个元素的顺序表 L 中的最大元素。

解法 1：顺序表 L 采用数组 data[0..$n-1$]存放全部元素，采用与例 5.4 类似的思路。对应的递归算法如下：

```
ElemType Max1(SqList L,int i)                       //解法 1:求顺序表 L 中的最大元素
{   ElemType max;
    if (i==0) return L.data[0];
    else
    {   max=Max1(L,i-1);
        if (max<L.data[i]) return L.data[i];
        else return max;
    }
}
```

解法 2：假设顺序表 L 中 data 数组的元素为 a_0、a_1、…、a_{n-1}，将其分解成(a_0，a_1，…，a_m)和(a_{m+1}，…，a_{n-1})左、右两个子表，分别求得它们的最大元素为 max1 和 max2，则整个顺序表 L 的最大元素就是 max1 和 max2 中的较大者，而左、右子表求最大元素是两个相似的子问题。对应的递归算法如下：

```
ElemType Max2(SqList L,int i,int j)                 //解法 2:求顺序表 L 中的最大元素
{   int mid;
    ElemType max,max1,max2;
    if (i==j)                                       //顺序表中只有一个元素,即递归出口
        max=L.data[i];                              //该元素就是最大元素
    else                                            //顺序表中有多个元素
    {   mid=(i+j)/2;                                //求中间位置
        max1=Max2(L,i,mid);                         //递归调用求左子表的最大元素 max1
        max2=Max2(L,mid+1,j);                       //递归调用求右子表的最大元素 max2
        max=(max1>max2)?max1:max2;                  //求整个表的最大元素 max
    }
    return(max);
}
```

5.3.2 基于递归数据结构的递归算法设计

具有递归特性的数据结构称为递归数据结构。递归数据结构通常是采用递归方式定义的。在一个递归数据结构中总会包含一个或者多个递归运算。

例如，正整数的定义为 1 是正整数，若 n 是正整数($n\geqslant1$)，则 $n+1$ 也是正整数。从中可以看出，正整数就是一种递归数据结构。显然，若 n 是正整数($n>1$)，$m=n-1$ 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。

扫一扫

视频讲解

所以在求 $n!$ 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。

一般情况下，对于递归数据结构

```
RD=(D,Op)
```

其中，$D=\{d_i\}(1\leqslant i\leqslant n$，共 n 个元素)为构成该数据结构的所有元素的集合，Op 是递归运算的集合，$\mathrm{Op}=\{\mathrm{op}_j\}(1\leqslant j\leqslant m$，共 m 个运算)。对于 $d_i\in D$，不妨设 op_j 为一元运算符，则有 $\mathrm{op}_j(d_i)\in D$，也就是说，递归运算具有封闭性。

在上述正整数的定义中，D 是正整数的集合，$\mathrm{Op}=\{\mathrm{op}_1,\mathrm{op}_2\}$由两个基本递归运算符构成，$\mathrm{op}_1$ 的定义为 $\mathrm{op}_1(n)=n-1(n>1)$；$\mathrm{op}_2$ 的定义为 $\mathrm{op}_2(n)=n+1(n\geqslant1)$。

对于不带头结点的单链表，其结点类型为 LinkNode，每个结点的 next 域为 LinkNode 类型的指针，这样的单链表通过首结点指针来标识。采用递归数据结构的定义如下：

```
SL=(D,Op)
```

其中，D 是由部分或全部结点构成的单链表的集合(含空单链表)，$\mathrm{Op}=\{\mathrm{op}_1\}$，$\mathrm{op}_1$ 的定义如下：

```
op1(L)=L->next          //L为含一个或一个以上结点的单链表
```

显然这个递归运算符是一元运算符，且具有封闭性。也就是说，若 L 为不带头结点的非空单链表，则 L->next 也是一个不带头结点的单链表。

实际上，递归算法设计步骤中的第 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，在设计不带头结点的单链表的递归算法时，通常设 s 为以 L 为首结点指针的整个单链表，s'为除首结点以外余下结点构成的单链表(由 L->next 标识，而该运算为递归运算)。

【例 5.6】 假设有一个不带头结点的单链表 L，设计一个算法释放其中的所有结点。

解 设 $f(L)$的功能是释放 $a_1\sim a_n$ 的所有结点，则 $f(L$->next)的功能是释放 $a_2\sim a_n$ 的所有结点，如图 5.6 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 所指的结点。

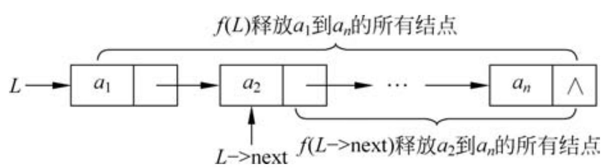


图 5.6　一个不带头结点的单链表

对应的递归模型如下：

$f(L)\equiv$	不做任何事情	当 L=NULL 时
$f(L)\equiv$	$f(L$->next)；释放 L 所指的结点	其他情况

其中，"≡"表示功能等价关系。对应的算法如下：

```
void release(LinkNode *&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结点的单链表。以图 5.6 为例，L—>next 表示的单链表一定是不带头结点的，也就是说“小问题”的单链表是不带头结点的单链表，这样“大问题”（即整个单链表）也应设计成不带头结点的单链表。

所以在设计递归算法时，如果处理的数据是递归数据结构，需要对该数据结构及其递归运算进行分析，从而设计出正确的递归体。再假设一种特殊情况，得到递归出口。

5.3.3 基于递归求解方法的递归算法设计

扫一扫

视频讲解

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

例如，求 $f(n)=1+2+\cdots+n(n\geqslant 1)$，这个问题可以转化为用递归方法求解，假设“小问题”是 $f(n-1)=1+2+\cdots+(n-1)$，它是可求的，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 5.7】 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 章中介绍过，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 变量保存一条迷宫路径，其中 PathType 类型的声明如下。

```
typedef struct
{   int i;                          //方块的行号
    int j;                          //方块的列号
} Box;                              //方块的类型
typedef struct
{   Box data[MaxSize];              //存放一条路径上的所有方块
    int length;                     //迷宫路径的长度
} PathType;                         //迷宫路径的类型
```

当从(xi,yi)方块找到一个相邻的可走方块(i,j)后，mgpath(i,j,xe,ye,path)表示求从(i,j)到出口(xe,ye)的迷宫路径。显然，mgpath(xi,yi,xe,ye,path)是“大问题”，而 mgpath(i,j,xe,ye,path)是“小问题”（即大问题＝试探一步＋小问题）。求解上述迷宫问题的递归模型如下：

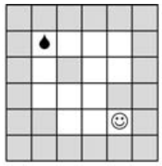
```
mgpath(xi, yi, xe, ye, path) ≡ 将(xi, yi)添加到 path 中;输出 path 中的迷宫路径;
                                                        若(xi, yi)=(xe, ye)为出口
mgpath(xi, yi, xe, ye, path) ≡ 对于(xi, yi)四周的每一个相邻方块(i,j): 若(xi, yi)不是出口且可走
                               ① 将(xi, yi)添加到 path 中;
                               ② mg[xi][yi]=-1;
                               ③ mgpath(i, j, xe, ye, path);
```

④ path 回退一步并置 mg[xi][yi]=0;
mgpath(xi,yi,xe,ye,path) ≡ 不做任何事情;　　　　若(xi,yi)不是出口且不可走

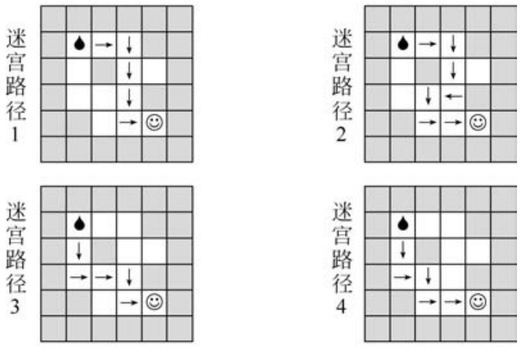
在上述递归模型中,当完成“小问题”mgpath(i,j,xe,ye,path)后将 path 回退并置 mg[xi][yi]为 0(对应④),其目的是恢复前面求迷宫路径而改变的环境,以便找出所有的迷宫路径。对应的递归算法如下:

```
void mgpath(int xi,int yi,int xe,int ye,PathType path)
//求解迷宫路径为(xi,yi)->(xe,ye)
{   int di,k,i,j;
    if (xi==xe && yi==ye)                        //找到了出口,输出一个迷宫路径
    {   path.data[path.length].i=xi;             //将(xi,yi)添加到 path 中
        path.data[path.length].j=yi;
        path.length++;
        printf("迷宫路径%d 如下:\n",++count);    //输出 path 中的迷宫路径
        for (k=0;k<path.length;k++)
            printf("\t(%d,%d)",path.data[k].i,path.data[k].j);
        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)                         //处理(xi,yi)四周的每一个相邻方块(i,j)
            {   path.data[path.length].i=xi;     //①将(xi,yi)添加到 path 中
                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;                   //②mg[xi][yi]=-1
                mgpath(i,j,xe,ye,path);          //③mgpath(i,j,xe,ye,path)
                mg[xi][yi]=0;                    //④恢复(xi,yi)为可走
                path.length--;                   //回退一个方块
                di++;                            //继续处理(xi,yi)下一个相邻方块
            }
        }
    }
}
```

本算法输出所有的迷宫路径,对于如图 5.7(a)所示的迷宫,指定入口为(1,1)、出口为(4,4),求出的迷宫路径有 4 条,如图 5.7(b)所示,可以通过比较路径长度求出最短迷宫路径(可能存在多条最短迷宫路径)。



(a) 一个迷宫



(b) 求出的所有迷宫路径

图 5.7　一个迷宫及其所有的迷宫路径

本 章 小 结

本章的基本学习要点如下：

(1) 理解递归的定义和递归模型。

(2) 理解递归算法的执行过程。

(3) 掌握递归算法设计的一般方法。

(4) 灵活地运用递归算法解决一些较复杂的应用问题。

练习题 5

1. 有以下递归函数：

```
void fun(int n)
{   if (n==1)
        printf("a:%d\n",n);
    else
    {   printf("b:%d\n",n);
        fun(n-1);
        printf("c:%d\n",n);
    }
}
```

分析调用 fun(5)的输出结果。

2. 有以下递归算法用于对数组 $a[i..j]$中的元素进行归并排序：

```
void mergesort(int a[], int i, int j)
{   int m;
    if (i!=j)
    {   m=(i+j)/2;
        mergesort(a, i, m);
        mergesort(a, m+1, j);
        merge(a, i, j, m);
    }
}
```

求执行 mergesort(a,0,$n-1$)的时间复杂度。其中，merge(a,i,j,m)用于两个有序子序列 $a[i..m]$和 $a[m+1..j]$的合并，是非递归函数，它的时间复杂度为 O(合并的元素个数)。

3. 已知 $A[0..n-1]$为实数数组，设计一个递归算法求这 n 个元素的平均值。

4. 设计一个算法求正整数 n 的位数。

5. 上楼可以一步上一阶，也可以一步上两阶，设计一个递归算法，计算共有多少种不同的走法。

6. 设计一个递归算法，利用顺序串的基本运算求串 s 的逆串。

7. 设有一个不带表头结点的单链表 L，设计一个递归算法 count(L)求以 L 为首结点指针的单链表的结点个数。

8. 设有一个不带表头结点的单链表 L，设计两个递归算法，traverse(L)正向输出单链表 L 中的所有结点值，traverseR(L)反向输出单链表 L 中的所有结点值。

9. 设有一个不带表头结点的单链表 L，设计两个递归算法，del(L,x)删除单链表 L 中第一个值为 x 的结点，delall(L,x)删除单链表 L 中所有值为 x 的结点。

10. 设有一个不带表头结点的单链表 L，设计两个递归算法，maxnode(L)返回单链表 L 中的最大结点值，minnode(L)返回单链表 L 中的最小结点值。

11. 设计一个模式匹配算法，其中模式串 t 中含一个或多个通配符'*'，每个'*'可以和任意子串匹配。对于目标串 s，求其中匹配模式串 t 的一个子串的位置('*'不能出现在 t 的开头)。

上机实验题 5

验证性实验

实验题 1：采用递归和非递归方法求解 Hanoi 问题

目的：领会基本递归算法的设计和递归到非递归的转换方法。

内容：编写程序 exp5-1.cpp，采用递归和非递归方法求解 Hanoi 问题，输出 3 个盘片的移动过程。

实验题 2：求路径和路径条数问题

目的：领会基本递归算法的设计和递归的执行过程。

内容：编写程序 exp5-2.cpp 求路径和路径条数。有一个 $m \times n$ 的网格，如图 5.8 所示为一个 2×5 的网格。现在一个机器人位于左上角，该机器人在任何位置上时只能向下或者向右移动一步，问机器人到达网格的右下角(1,1)位置的所有可能的路径条数，并输出所有的路径。以 $m=2$，$n=2$ 为例说明输出所有路径的过程。

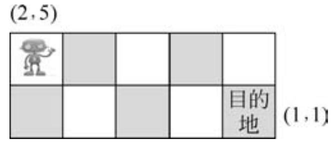


图 5.8　一个 2×5 的网格

设计性实验

实验题 3：恢复 IP 地址

目的：掌握基本递归算法的设计。

内容：编写程序 exp5-3.cpp 恢复 IP 地址。给定一个仅包含数字的字符串，恢复它的所有可能的有效 IP 地址。例如，给定字符串为"25525511135"，返回"255.255.11.135"和"255.255.111.35"(顺序可以任意)。

实验题 4：高效求解 x^n

目的：掌握基本递归算法的设计。

内容：编写程序 exp5-4.cpp 高效求解 x^n，要求最多使用 $O(\log_2 n)$次递归调用。

实验题 5：用递归方法逆置带头结点的单链表

目的：掌握单链表递归算法的设计方法。

内容：编写一个程序 exp5-5.cpp，用递归方法逆置一个带头结点的单链表。

实验题 6：用递归方法求单链表中的倒数第 k 个结点

目的：掌握单链表递归算法的设计方法。

内容：编写一个程序 exp5-6.cpp，用递归方法求单链表中的倒数第 k 个结点。

综合性实验

实验题 7：用递归方法求解 n 皇后问题

目的：深入掌握递归算法的设计方法。

内容：编写一个程序 exp5-7.cpp，用递归方法求解 n 皇后问题，对 n 皇后问题的描述参见第 3 章中的实验题 8。

实验题 8：用递归方法求解 0/1 背包问题

目的：深入掌握递归算法的设计方法。

内容：编写一个程序 exp5-8.cpp，用递归方法求解 0/1 背包问题。0/1 背包问题是设有 n 件物品，每件物品有相应的重量和价值，求从这 n 件物品中选取全部或部分物品的方案，使选中物品的总重量不超过指定的限制重量 W，但选中物品的价值之和为最大。注意，每种物品要么被选中，要么不被选中。

LeetCode 在线编程题 5

1. LeetCode509——斐波那契数★
2. LeetCode50——Pow(x,n)★★

3. LeetCode206——翻转链表★
4. LeetCode234——回文链表★
5. LeetCode24——两两交换链表中的结点★★
6. LeetCode59——螺旋矩阵Ⅱ★★
7. LeetCode51——n 皇后★★★