第5章 递归

关联的知识点

- 理解递归的定义和递归模型。
- 重点掌握递归的执行过程。
- 掌握递归设计的一般方法。掌握消除递归的基本方法。
- 灵活运用递归算法解决一些较复杂应用问题。

5.1 什么是递归

关联的知识点

- 递归的概念
- 什么情况使用递归
- 什么是递归模型

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，称之为递归。若调用自身,称之为直接递归。若过程或函数p调用过程或函数q，而q又调用p，称之为间接递归。

递归相关的概念

递归关系：一个数列的若干连续项之间的关系。

递归数列：由递归关系所确定的数列。

递归过程：直接或间接调用自身的过程。

递归算法：包含递归过程的算法。

递归程序：直接或间接调用自身的程序。

递归方法：是指一种在有限步骤内，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**。

例如，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）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。

✓有人送了我金、银、铜、铁、木五个宝箱，我想打开金箱子，却没有打开这个箱子的钥匙。

✓在金箱子上面写着一句话：“打开我的钥匙装在银箱子里。”

✓于是我来到银箱子前，发现还是没有打开银箱子的钥匙。

✓银箱子上也写着一句话：“打开我的钥匙装在铜箱子里。”

✓于是我再来到铜箱子前，发现还是没有打开铜箱子的钥匙。

✓铜箱子上也写着一句话：“打开我的钥匙装在铁箱子里。”

✓于是我又来到了铁箱子前，发现还是没有打开铁箱子的钥匙。

✓铁箱子上也写着一句话：“打开我的钥匙装在木箱子里。”

✓我来到木箱子前，打开了木箱，

✓并从木箱里拿出铁箱子的钥匙，打开了铁箱，

✓从铁箱里拿了出铜箱的钥匙，打开了铜箱，

✓再从铜箱里拿出银箱的钥匙打开了银箱，

✓最后从银箱里取出金箱的钥匙，打开了我想打开的金箱子。

✓晕吧……很啰嗦地讲了这么长一个故事。

```
void FindKey（ 箱子 ）{
    if（ 木箱子）return ;
    else FindKey（ 下面的箱子 ）}
```

当多个函数构成嵌套调用时，遵循

后调用先返回

栈

5.1.2 何时使用递归

- 以下三种情况常常用到递归方法
  - 递归定义的数学函数
  - 具有递归特性的数据结构
  - 可递归求解的问题

能够用递归解决的问题就应该满足一下三个条件：

- 需要解决的问题可以转化为一个或多个子问题求解，而这些子问题的求解方法与原问题完全相同，只是在数量规模上有所不同。
- 递归调用的次数必须是有限的。
- 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,求n!和Fibonacci数列等。这些问题的求解过程可以将其递归定义直接转化为对应的递归算法。

递归定义的数学函数:

- 阶乘函数:

$$Fact(n)=\begin{cases}1 & \text{若n}=0\\ n\cdot Fact(n-1) & \text{若n}>0\end{cases}$$

- 2阶Fibonaci数列:

$$Fib(n)=\begin{cases}1 & \text{若n}=1\text{或}2\\ Fib(n-1)+Fib(n-2) & \text{其它}\end{cases}$$

2. 数据结构是递归的

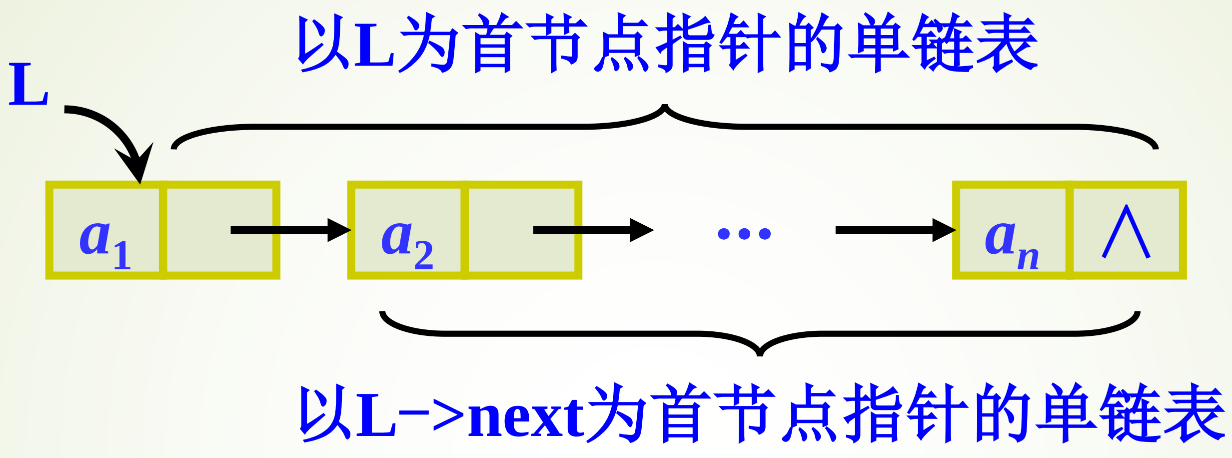
有些数据结构是递归的。例如,第2章中介绍过的单链表就是一种递归数据结构，其结点类型定义如下：

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

为什么可以这样递归定义类型

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。

单链表示意图



体现出数据结构的递归性。

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问题求解。

汉诺塔

在印度圣庙里，一块黄铜板上插着三根宝石针。
主神梵天在创造世界时，在其中一根针上穿好了由大到小的64片金片，这就是汉诺塔。
僧侣不停移动这些金片，一次只移动一片，小片必在大片上面。
当所有的金片都移到另外一个针上时，世界将会灭亡。

该问题描述是：设有3个分别命名为A,B和C的塔座，在塔座A上有n个直径各不相同，从小到大依次编号为1,2,…,n的圆盘，现要求将A塔座上的n个圆盘移到塔座C上并仍按同样顺序叠放。

设计递归求解算法。

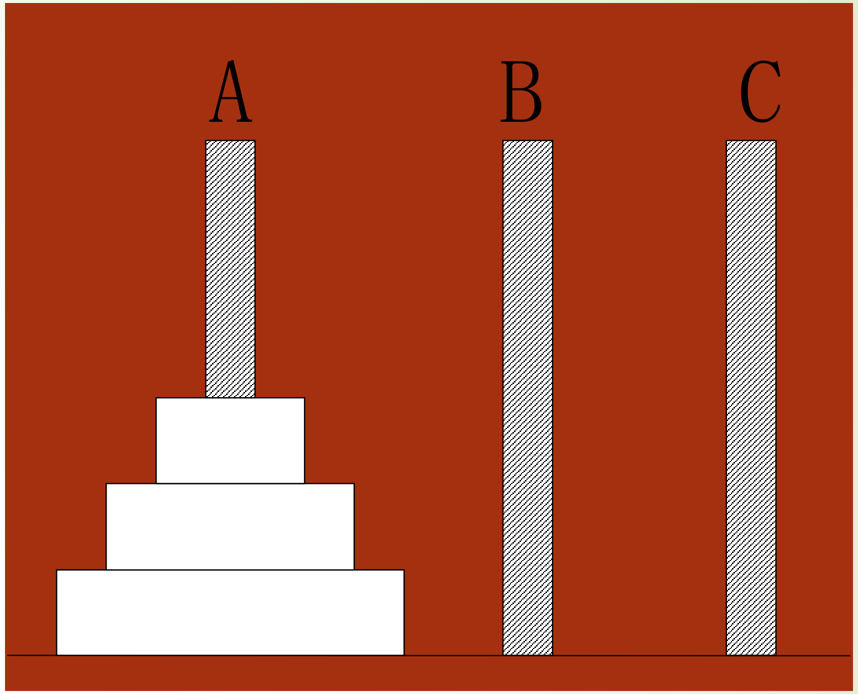
Hanoi塔问题

规则：

（1）每次只能移动一个圆盘

（2）圆盘可以插在A, B和C中的任一塔座上

（3）任何时刻不可将较大圆盘压在较小圆盘之上



设Hanoi(n,A,B,C)表示将n个盘片从A通过B移动到C上。

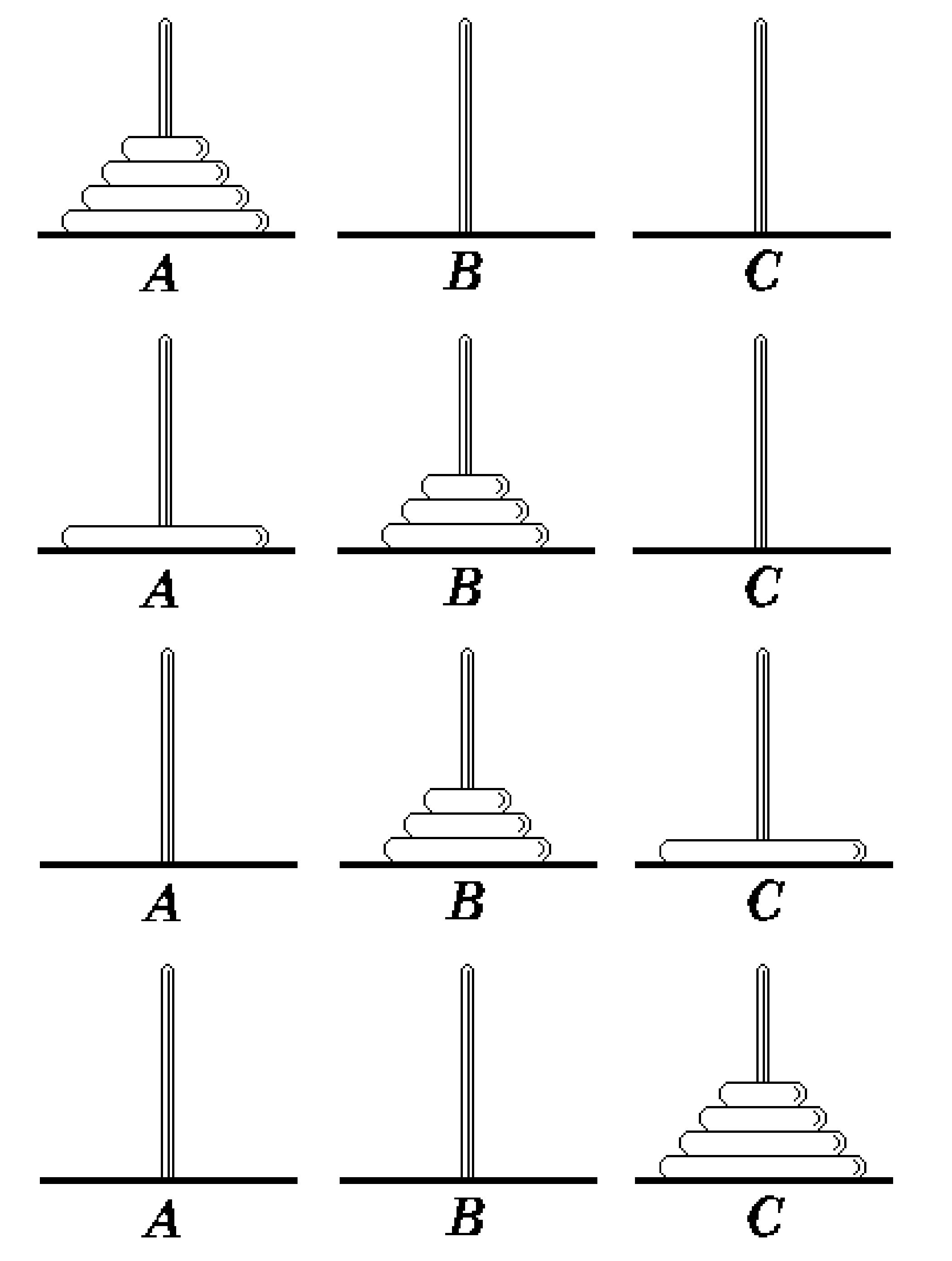
Hanoi塔问题

n = 1，则直接从 A 移到 C。否则

(1)用 C 柱做过渡，将 A 的(n-1)个移到 B

(2)将 A 最后一个直接移到 C

(3)用 A 做过渡，将 B 的 (n-1) 个移到 C



```
Hanoi(n,A,B,C)
```

```
Hanoi(n-1,A,C,B);
move(n,A,C):将第n个圆盘从A移到C;
Hanoi(n-1,B,A,C)
```

“大问题”转化为若干个“小问题”求解

```
int c=0;
void move(int n,char x, char z)
{printf（++c，“,”，n，“,”，x，“,”，z）；}
void Hanoi(int n,char A,char B,char C)
{ if(n==1) move(A,1,C);
  else
  {  Hanoi(n-1,A,C,B);
     move(n, A,C);
     Hanoi(n-1,B,A,C);
   }
 }
```

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。例如前面的递归算法对应的递归模型如下：

```
fun(1)=1                              (1)
fun(n)=n*fun(n-1)        n>1          (2)
```

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fun(n)的值与fun(n−1)的值之间的关系，我们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，前者确定递归到何时结束，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：

$$f(s_1)=m_1 \qquad (5.1)$$

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：

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

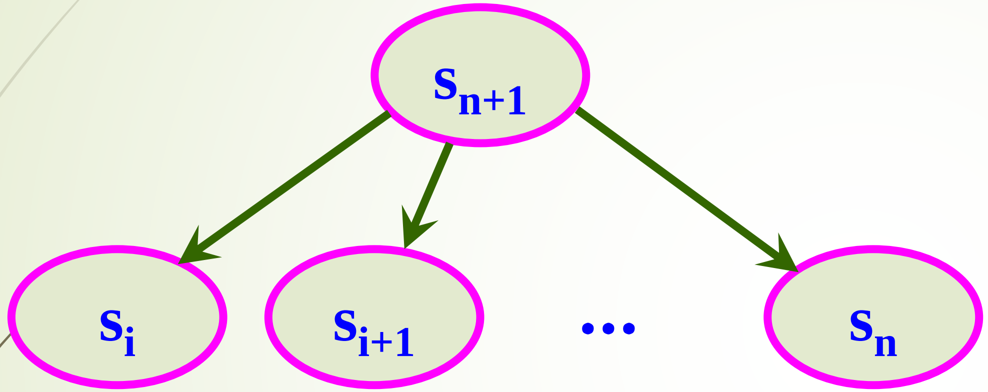
其中，n、i、j和m均为正整数。这里的s_{n+1}是一个递归“大问题”，s_i、s_{i+1}、…、s_n为递归“小问题”，c_j、c_{j+1}、…、c_m是若干个可以直接（用非递归方法）解决的问题，g是一个非递归函数，可以直接求值。

递归模型的动画演示

通用递归模型：

$f(s_1)=m_1$

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



大问题求解

↓

若干个相似子问题求解

```
fun(1)=1
fun(n)=n*fun(n-1)
```

↓ 通用形式

```
fun(1)=1
fun(n)=g(n,fun(n-1))
```

← g为乘法函数

递归思路

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的分解，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：

$$f(s_1)=m_1$$

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

求$f(s_n)$的分解过程如下：

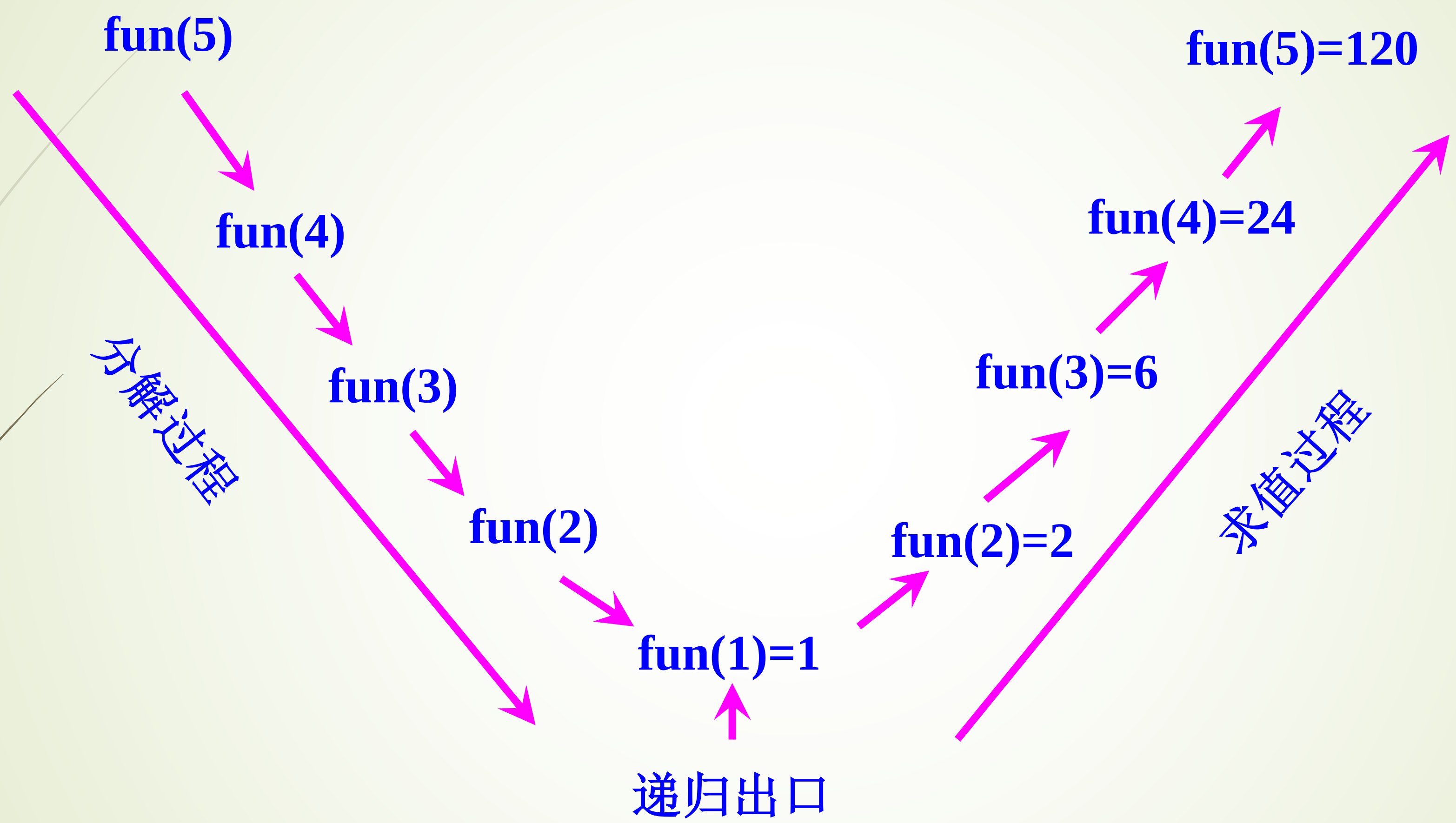
$$f(s_n)$$
$$\downarrow$$
$$f(s_{n-1})$$
$$\downarrow$$
$$\cdots$$
$$\downarrow$$
$$f(s_2)$$
$$\downarrow$$
$$f(s_1)$$

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：

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

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。

求解fun(5)的过程如下：



递归函数求解的动画演示

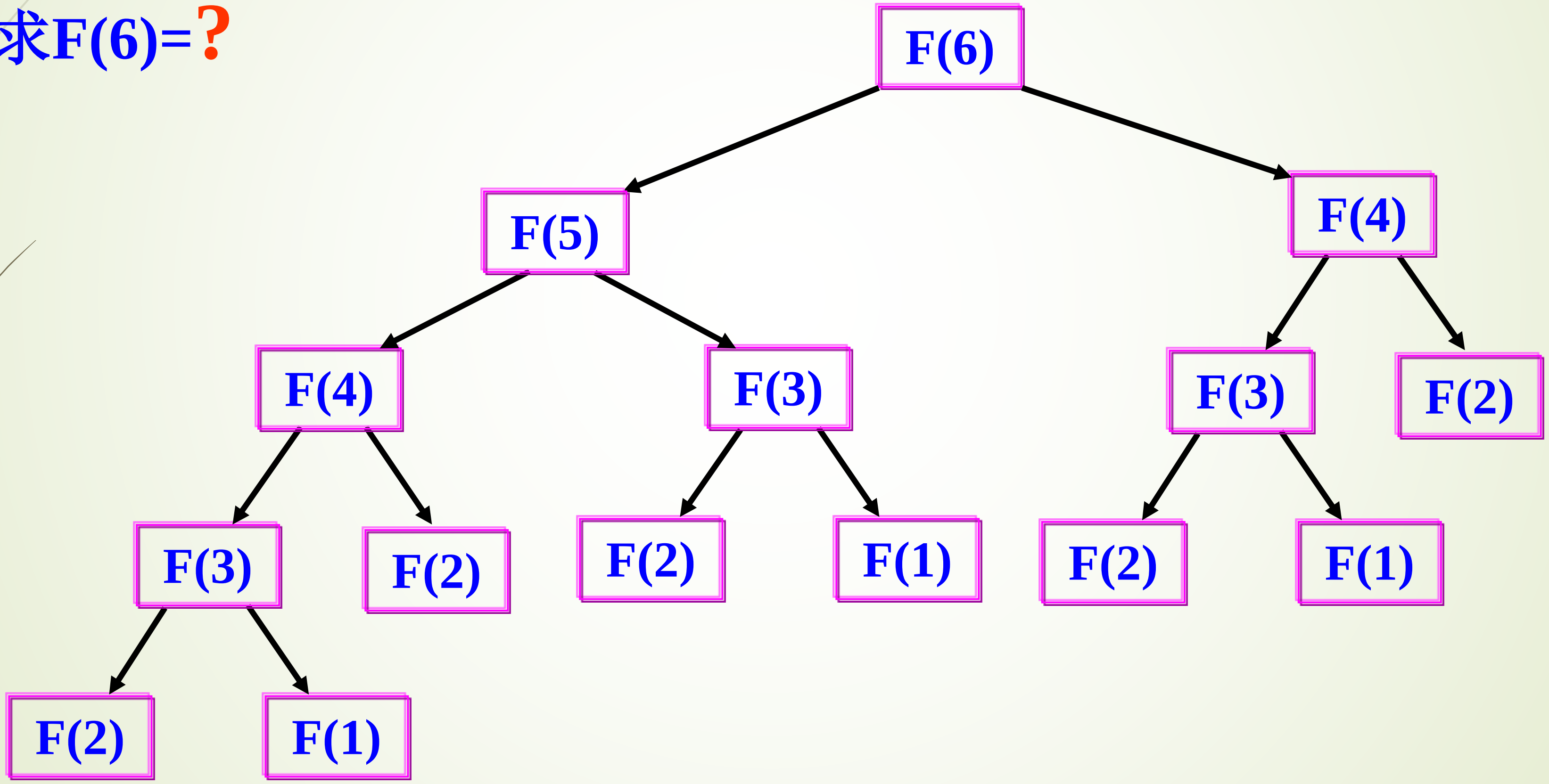
$F(1)=1$

$F(2)=1$

$F(n)=F(n-1)+F(n-2)\quad n>2$

求$F(6)=$?

求得$F(6)=8$



求解递归树

该问题描述是：设有3个分别命名为A,B和C的塔座，在塔座A上有n个直径各不相同，从小到大依次编号为1,2,…,n的圆盘，现要求将A塔座上的n个圆盘移到塔座C上并仍按同样顺序叠放。

设计递归求解算法，并将其转换为非递归算法。

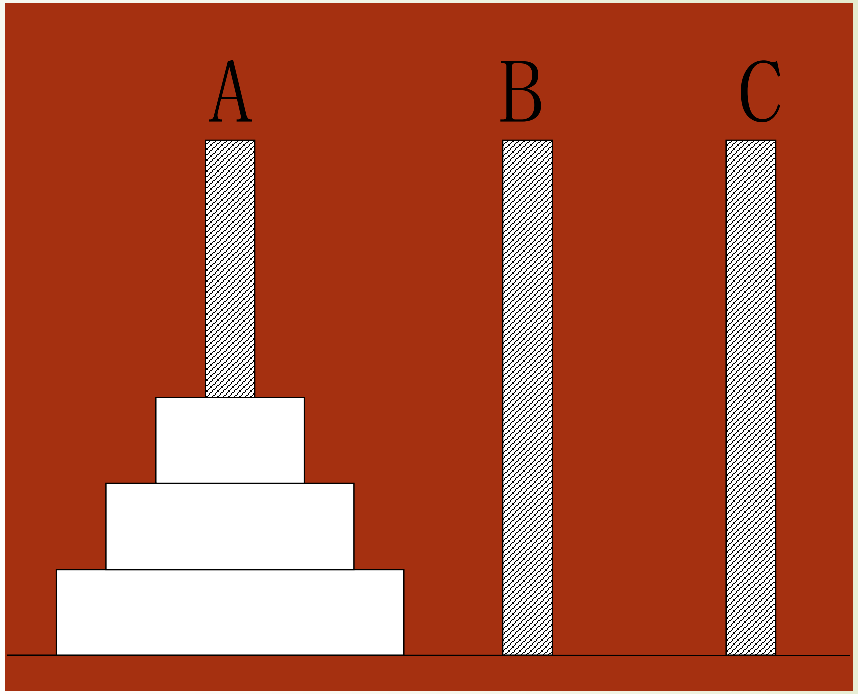
Hanoi塔问题

规则：

（1） 每次只能移动一个圆盘

（2） 圆盘可以插在A, B和C中的任一塔座上

（3） 任何时刻不可将较大圆盘压在较小圆盘之上

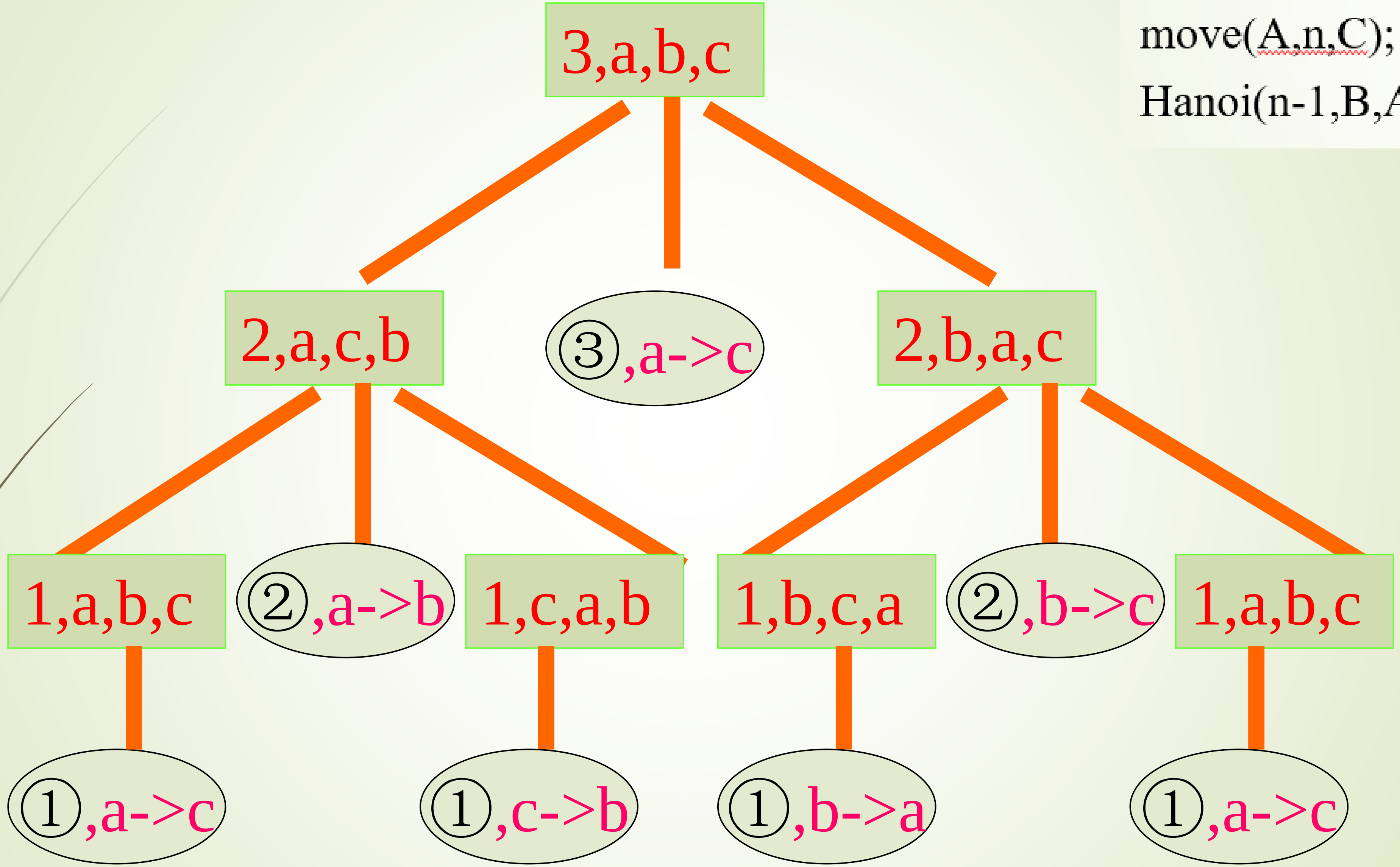


设Hanoi(n,A,B,C)表示将n个盘片从A通过B移动到C上。

```
int c=0;
void move(char x,int n,char z)
{printf（++c，“,”，n，“,”，x，“,”，z）；}
void Hanoi(int n,char A,char B,char C)
{ if(n==1) move(A,1,C);
  else
  {  Hanoi(n-1,A,C,B);
     move(A,n,C);
     Hanoi(n-1,B,A,C);
   }
 }
void main(){Hanoi(3,'a','b','c');}
```

递归调用树

```
Hanoi(n-1,A,C,B);
move(A,n,C);
Hanoi(n-1,B,A,C);
```



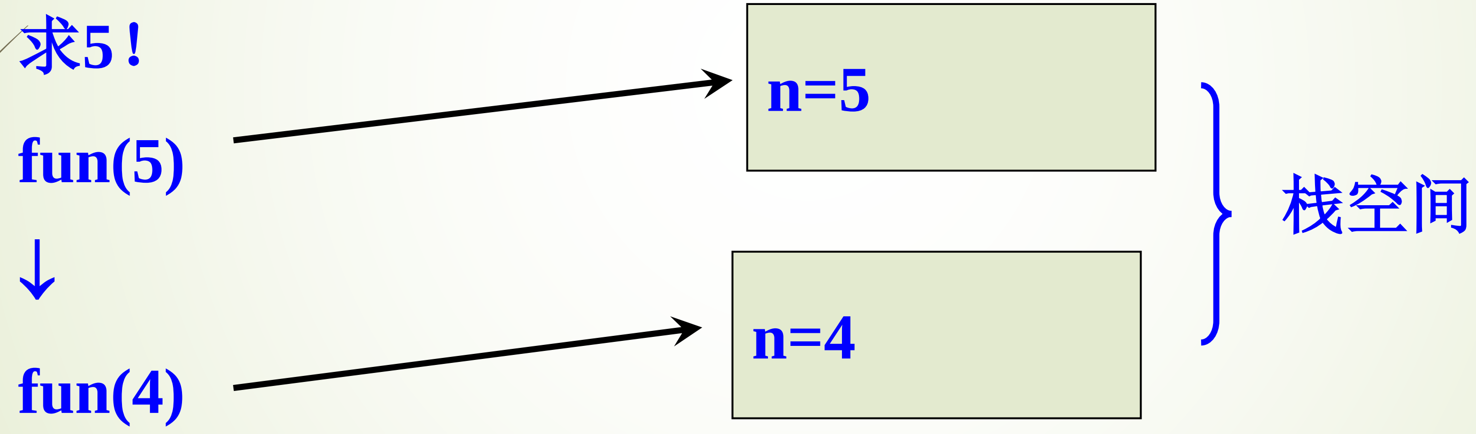
5.2 递归调用的实现原理

关联的知识点

递归调用的实现原理

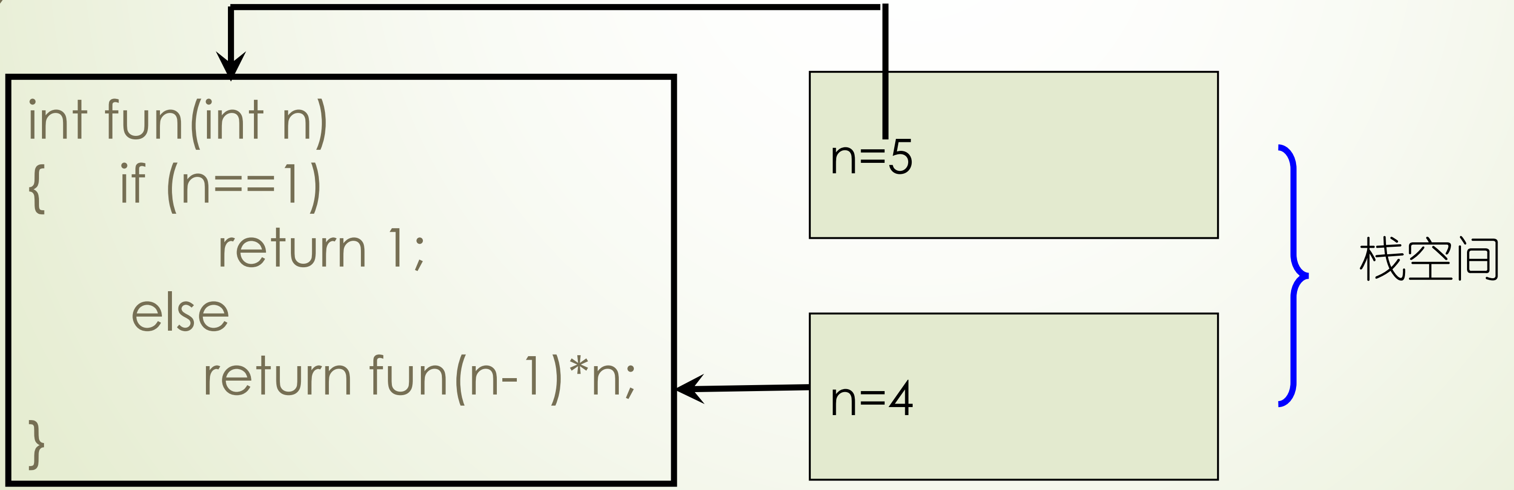
递归调用是函数嵌套调用的一种特殊情况，即它是调用自身代码。因此，也可以把每一次递归调用理解成调用自身代码的一个复制件。

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。



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。

这些单元以内部栈的形式存放，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。

求5!栈变化过程的动画演示

```
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
}
```

调用fun(5)

分解过程:进栈

d1	5	fun(4)*5
返回地址	n	函数值等

d2	4	fun(3)*4
d1	5	fun(4)*5
返回地址	n	函数值等

d3	3	fun(2)*3
d2	4	fun(3)*4
d1	5	fun(4)*5
返回地址	n	函数值等

d4	2	fun(1)*2
d3	3	fun(2)*3
d2	4	fun(3)*4
d1	5	fun(4)*5
返回地址	n	函数值等

1	
2	fun(1)*2
3	fun(2)*3
4	fun(3)*4
5	fun(4)*5
n	函数值等

求值过程:退栈

```
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
}
```

返回地址	n	函数值等
	1	fun(1)=1
d4	2	fun(1)*2
d3	3	fun(2)*3
d2	4	fun(3)*4
d1	5	fun(4)*5

返回地址	n	函数值等
	2	2
d3	3	fun(2)*3
d2	4	fun(3)*4
d1	5	fun(4)*5

返回地址	n	函数值等
	3	6
d2	4	fun(3)*4
d1	5	fun(4)*5

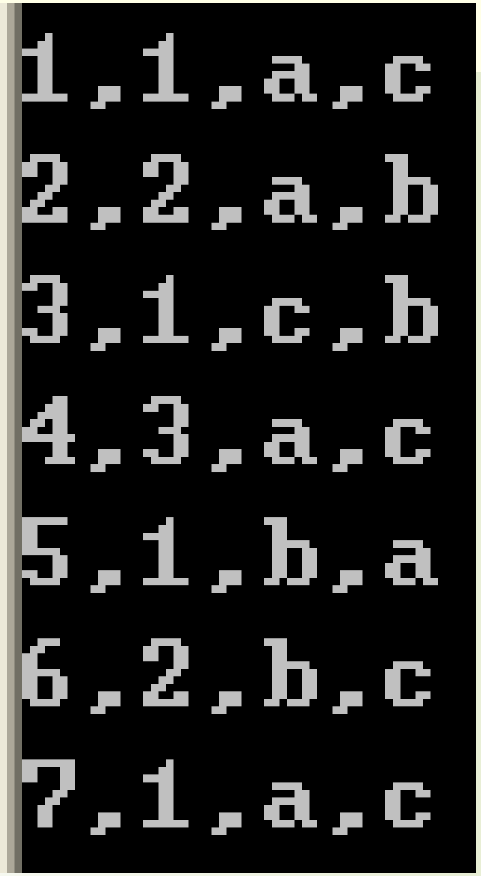
返回地址	n	函数值等
	4	24
d1	5	fun(4)*5

n	函数值等
5	120

返回函数值120

跟踪程序，给出下列程序的运行结果，以深刻地理解递归的调用和返回过程

```
int c=0;
void move(char x,int n,char z)
{printf（++c，“,”，n，“,”，x，“,”，z）；}
void Hanoi(int n,char A,char B,char C)
{ if(n==1) move(A,1,C);
  else
  {  Hanoi(n-1,A,C,B);
     move(A,n,C);
     Hanoi(n-1,B,A,C);
   }
 }
void main(){Hanoi(3,'a','b','c');}
```



5.3 递归算法的设计

关联的知识点

递归算法设计的一般步骤

基于递归数据结构的递归算法设计

基于递归求解方法的递归算法设计

5.3.1 递归算法设计的步骤

递归的求解的过程均有这样的特征：

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。

而这些子问题具有与原问题相同的求解方法，于是可以再将它们划分成若干个子问题，分别求解，如此反复进行，直到不能再划分成子问题，或已经可以求解为止。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++语言函数。

求递归模型的步骤如下：

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

（2）假设f(s')是可解的，在此基础上确定f(s)的解，即给出f(s)与f(s')之间的关系（与数学归纳法中求证$n=k$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$n=1$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]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()为求两个值较小值函数。因此得到如下递归模型：

f(A,i)=A[0]	当i=0时
f(A,i)= MIN(f(A,i-1),A[i])	其他情况

由此得到如下递归求解算法：

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

【P128例5.2】利用串的基本运算写出对串求逆的递归算法。

其递归思路是：对于s="$s_1s_2……s_n$"的串，假设"$s_2s_3……s_n$"已求出其逆串，再将s1链接到最后即得到s的逆串。

s： $s_1\ s_2\ ……\ s_n$

s 的逆：$\underbrace{s_n\ s_{n-1}……s_2}_{\text{f(SubStr(s,2,Strlength(s)-1)}}\ \ \underbrace{s_1}_{\text{SubStr(s,1,1)}}$

$$f(s) = \begin{cases} s & \text{当}s=\varnothing\text{时} \\ \text{Concat(f(SubStr(s,2,Strlength(s)-1)),SubStr(s,1,1))} & \text{其他情况} \end{cases}$$

【P128例5.3】求顺序表{a_1， a_2， ……,a_n}中最大元素。

$$\underbrace{a_1, a_2, \ldots\ldots, a_m}_{a_i} \quad \underbrace{a_{m+1}, a_{m+2}, \ldots\ldots, a_n}_{a_j} \qquad Max(a_i, a_j)$$

……直到表中只有一个元素为止，此时该元素就是该表的最大元素。

由此得到如下递归求解算法：

```
ElemType Max(SqList L,int i,int j)
{   int mid;
    ElemType max, max1, max2;
    if (i==j)
       max=L.data[i];
    else
    {   mid=(i+j)/2;
        max1=Max[L,i,mid];
        max2=Max[L,mid+1,j];
        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，如n是正整数（n>=1），则n+1也是正整数。

从中可以看出，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*（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，$D=\{d_i\}(1\leq i \leq n$，共n个元素)为构成该数据结构的所有元素的集合，$Op=\{op_j\}(1\leq j \leq m$，共m个元素)，不妨设$op_j$为一元运算符，$\forall\ d_i \in D$，应有$op(d_i) \in D$，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（对于固定数位的计算机，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1的正整数

例如，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={op1}：

op1(L)=L->next　　　L为含一个或一个以上节点的单链表

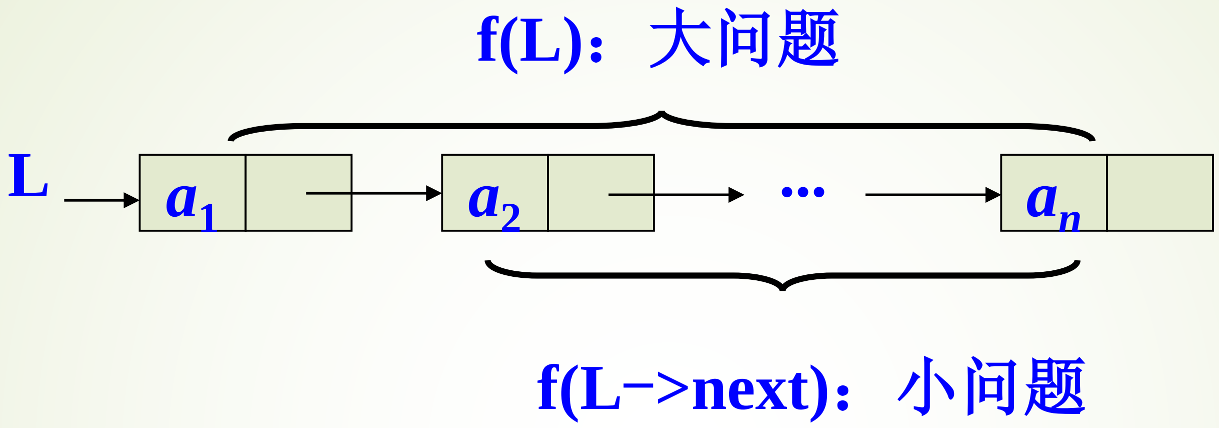
显然这个基本递归运算符是一元运算符，且具有封闭性。

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。

➔例如，设计不带头结点的单链表的相关递归算法



（1）求单链表中数据结点个数

递归模型如下：

f(L)=0　　　　　当L=NULL

f(L)=f(L->next)+1　其他情况

递归算法如下：

```
int count(Node *L)
{  if (L==NULL)
     return 0;
   else
     return count(L->next)+1;
}
```

（2）正向显示以L为首节点指针的单链表的所有节点值

递归模型如下：

f(L)↔不做任何事件　　　　当L=NULL

f(L)↔输出L->data;f(L->next) 其他情况

递归算法如下：

```
void traverse(Node *L)
{  if (L==NULL) return;
   printf("%d ",L->data);
   traverse(L->next);
}
```

（3）反向显示以L为首节点指针的单链表的所有节点值

递归模型如下：

f(L)↔不做任何事件　　　　当L=NULL

f(L)↔f(L->next);输出L->data;其他情况

递归算法如下：

```
void traverseR(Node *L)
{  if (L==NULL) return;
   traverseR(L->next);
   printf("%d ",L->data);
}
```

（4）输出以L为首节点指针的单链表中最大节点值

递归模型如下：

f(L)=L->data　　　　　　当L中只有一个结点

f(L)=MAX(f(L->next),L->data)　　其他情况

递归算法如下：

```
ElemType maxv(Node *L)
{   ElemType m;
    if (L->next==NULL)
     return L->data;
    m=maxv(L->next);
    if (m>L->data) return m;
      else return L->data;
}
```

（5）输出以L为首节点指针的单链表中最小节点值。

递归模型如下：

f(L)=L->data　　　　　　当L中只有一个结点
f(L)=MIN(f(L->next),L->data)其他情况

递归算法如下：

```
ElemType minv(Node *L)
{    ElemType m;
     if (L->next==NULL)
      return L->data;
     m=minv(L->next);
     if (m>L->data) return L->data;
       else return m;
}
```

（6）释放L为首节点指针的单链表的所有节点

递归模型如下：

f(L)↔不做任何事件　　　　当L=NULL

f(L)↔f(L->next);释放L所指结点; 其他情况

递归算法如下：

```
void Destroy(Node *L)
{  if (L==NULL) return;
   Destroy(L->next);
   free(L);
}
```

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$f(n)=1+2+\cdots+n$（$n\geq 1$），这个问题可以转化为递归方法求解，假设“小问题”是$f(n-1)=1+2+\cdots+(n-1)$是可求的，则$f(n)=f(n-1)+n$。

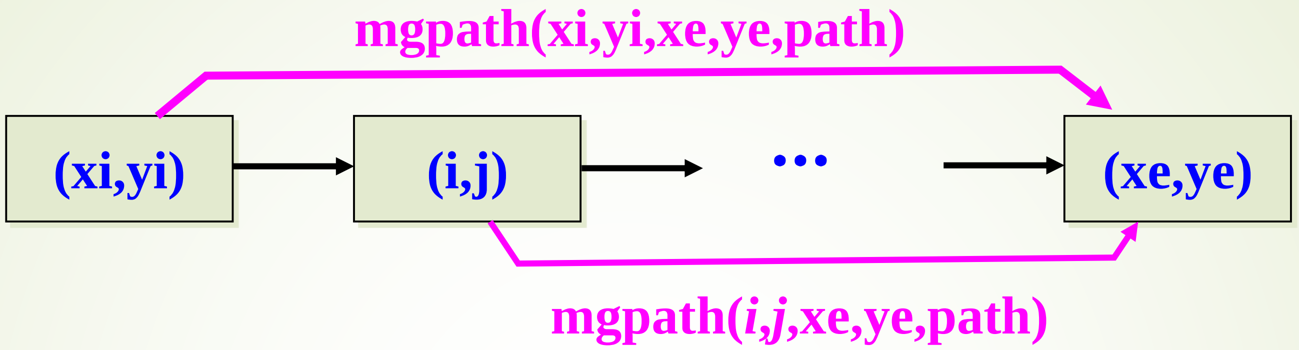
对于采用递归方法求解的问题，需要对问题本身进行分析，确定大、小问题解之间的关系，构造合理的递归体。

【例5.5：p130】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)方块出发找到一个可走相邻方块(*i*,*j*)后，用算法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)是求从(xi,yi)到(xe,ye)的一条迷宫路径，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[xi][yi]为0，其目的是恢复前面求迷宫路径中的环境，以便找出多有的迷宫路径。对应的递归算法如下：

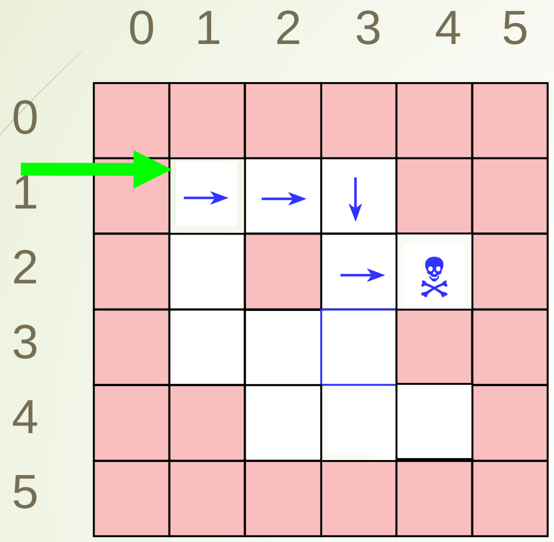
```
void mgpath(int xi,int yi,int xe,int ye,PathType path)
//求解路径为:(xi,yi)->(xe,ye)
{   int di,k,i,j;
    if (xi==xe && yi==ye)    //找到了出口,输出路径
    {  path.data[path.length].i = xi;
       path.data[path.length].j = yi;
       path.length++;
       printf("迷宫路径%d如下:\n",++count);
       for (k=0;k<path.length;k++)
       {  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++;
      }// end of while
    }//end of  if (mg[xi][yi]==0)
  }// end of else  (xi,yi)不是出口
}
```

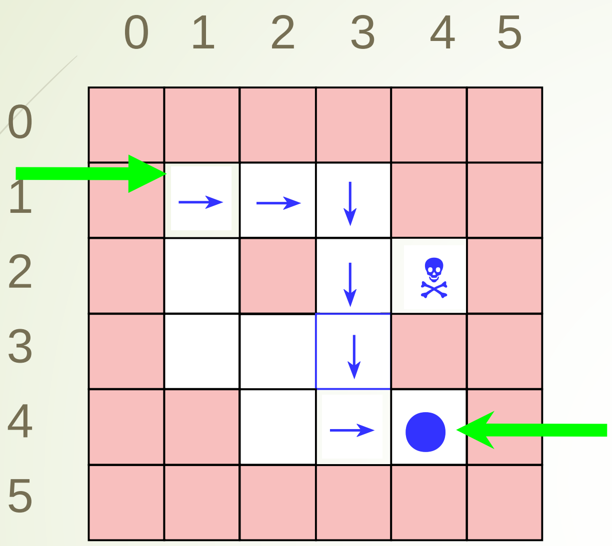
本算法可以输出所有的迷宫路径，可以通过比较找出最短路径（可能存在多条最短路径）。

用递归方法求解迷宫路径的动画演示



1，1	1，2	1，3	2，3	2，4	

用递归方法求解迷宫路径的动画演示



x_i, y_i为4，3

1，1	1，2	1，3	2，3	3，3	4，3	4，4	

思考题：

迷宫问题的递归求解与用栈和队列求解有什么异同。

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。

实验五　求解皇后问题

- 【问题描述】请编写一个程序求解皇后问题：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。
- 【基本要求】

  由用户来输入皇后的个数，不低于4，不能超过20；

  要求输出所有的解。
- 【提示】

  采用递归方法求解。

print（int n）：输出一个解。
place(int k,int j):测试（k,j）位置能否摆放皇后。
queen(int k,int n):用于在1~k行放置皇后。

采用整数数组q[n]求解结果，因为每行只能放一个皇后，q[i](1≤ i ≤n)的值表示第i个皇后所在的列号，即该皇后放在（i，q[i]）的位置上。

大问题 ← 设queen(k,n)是在1~k-1行上已经放好了k-1个皇后，用于在k~n行放置n-k+1个皇后

小问题 ← 则queen(k+1,n)表示在1~k行上已经放好了k个皇后，用于在k+1~n行放置n-k个皇后

显然queen（k+1，n）比queen（k，n）少放置一个皇后

递归模型

queen（k, n) ↔ n个皇后放置完毕，输出解　　　　若k>n

queen（k，n）↔对于第k行的每个合适的位置i，其他情况

在其上放置一个皇后；

queen(k+1,n);

递归过程

```
queen(int k,int n)
{   if (k>n)
      输出一个解;
    else
      for(j=1;j<=n;j++)
        if (第k行的第j列合适)
        {  在（k，j）位置处放一个皇后即q[k]=j
           queen（k+1，n）;
        }
 }
```

—本章完—