第5章　递归

5.1　什么是递归

5.2　递归和栈

5.3　递归算法的设计

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，出现直接或者间接调用自己的成分，称之为**递归**。

- 若直接调用自己，称之为**直接递归**。
- 若间接调用自己，称之为**间接递归**。

直接递归函数示例：求n!（n为正整数）

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

间接递归示例：

```
void  f1(...)
{
    ...
    f2( ...);
    ...

}
```

```
void  f2(...)
{
    ...
    f1( ...);
    ...

}
```

总可以转换为直接递归函数

如果一个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**。

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

直接递归函数、尾递归

- **尾递归算法：可以用循环语句转换为等价的非递归算法**
- **其他递归算法：可以通过栈来转换为等价的非递归算法**

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：求*n*!和Fibonacci数列等 --》 直接使用递归算法。

- 著名的兔子问题
  - 假设有一对兔子，长两个月它们就算成年了。以后每个月都会生出1对兔子，生下来的兔子也都是长两个月就算成年，然后每个月也都会生出1对兔子了。这里假设兔子不会死，每次都是只生1对兔子。
  - 第一个月，只有1对小兔子；
  - 第二个月，小兔子还没长成年，还是只有1对兔子；
  - 第三个月，兔子长成年了，同时生了1对小兔子，因此有两对兔子；
  - 第四个月，成年兔子又生了1对兔子，加上自己及上月生的小兔子，共有3对兔子；
  - 第五个月，成年兔子又生了1对兔子，第三月生的小兔子现在已经成年且生了1对小兔，加上上月生的小兔子，共5对兔子；
  - 这样过了一年之后，会有多少对兔子了呢？

2、数据结构是递归的

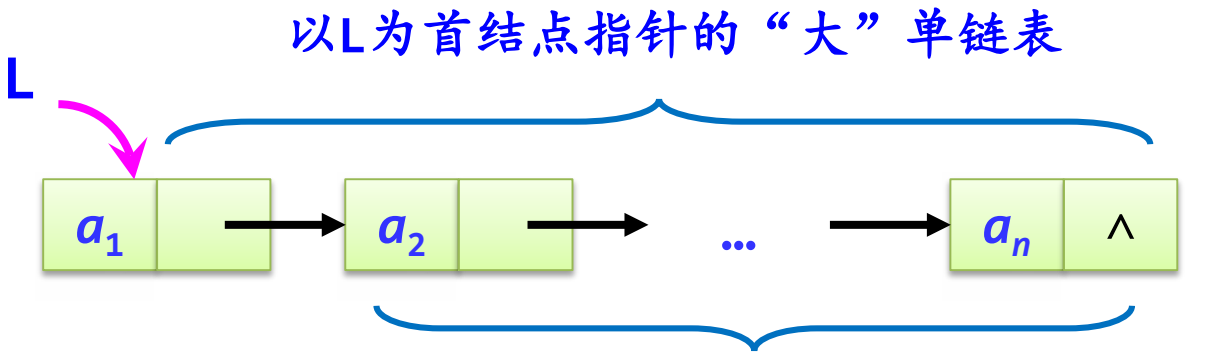
有些数据结构是递归的。例如，第2章中介绍过的单链表就是一种递归数据结构，其结点类型定义如下：

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

指向同类型结点的指针

递归数据结构

不带头结点单链表示意图



体现出这种单链表的递归性。

思考：如果带有头结点又会怎样呢？？？

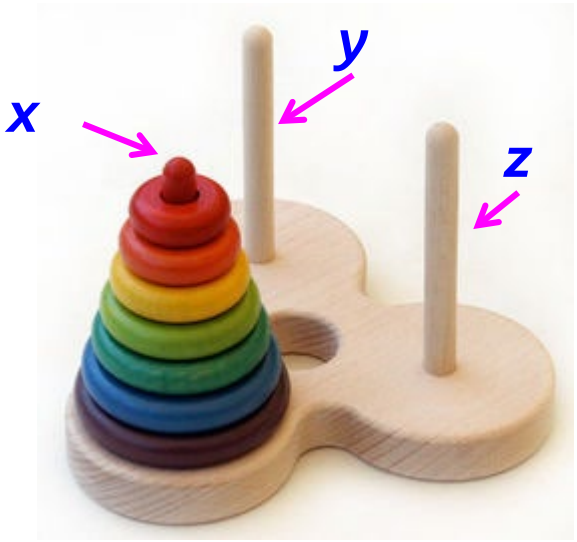
3、问题的求解方法是递归的

Hanoi问题：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～n的盘片。要求将X塔座上的n个盘片移到塔座Z上。

移动规则：

- 每次只能移动一个盘片；
- 盘片可以插在X、Y和Z中任一塔座上；
- 任何时候都不能将一个较大的盘片放在较小的盘片上方。

设Hanoi(*n*，*x*，*y*，*z*)表示将*n*个盘片从*x*通过*y*移动到*z*上。



Hanoi(n, x, y, z)

```
Hanoi(n-1, x, z, y);
move(n, x, z):将第n个圆盘从x移到z;
Hanoi(n-1, y, x, z)
```

“大问题”转化为若干个“小问题”求解

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。

例如求$n!$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)
```

递归出口

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。

- 递归出口确定递归到何时结束。
- 递归体确定递归求解时的递推关系。

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；

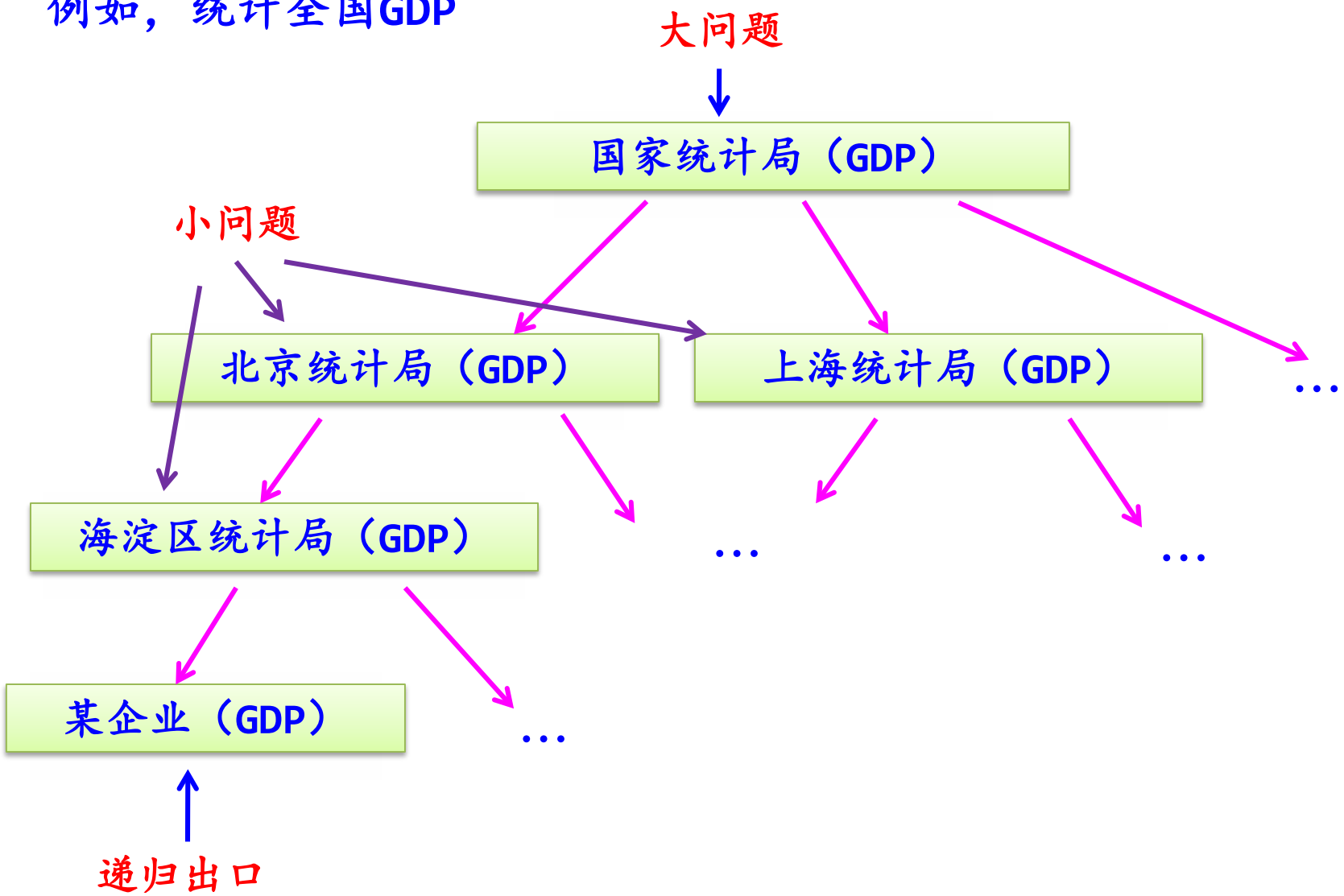
再把这些“小问题”进一步分解成更小的“小问题”来解决。

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。

例如，统计全国GDP



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：

$f(s_n)$

↓

$f(s_{n-1})$

↓

…

↓

$f(s_2)$

↓

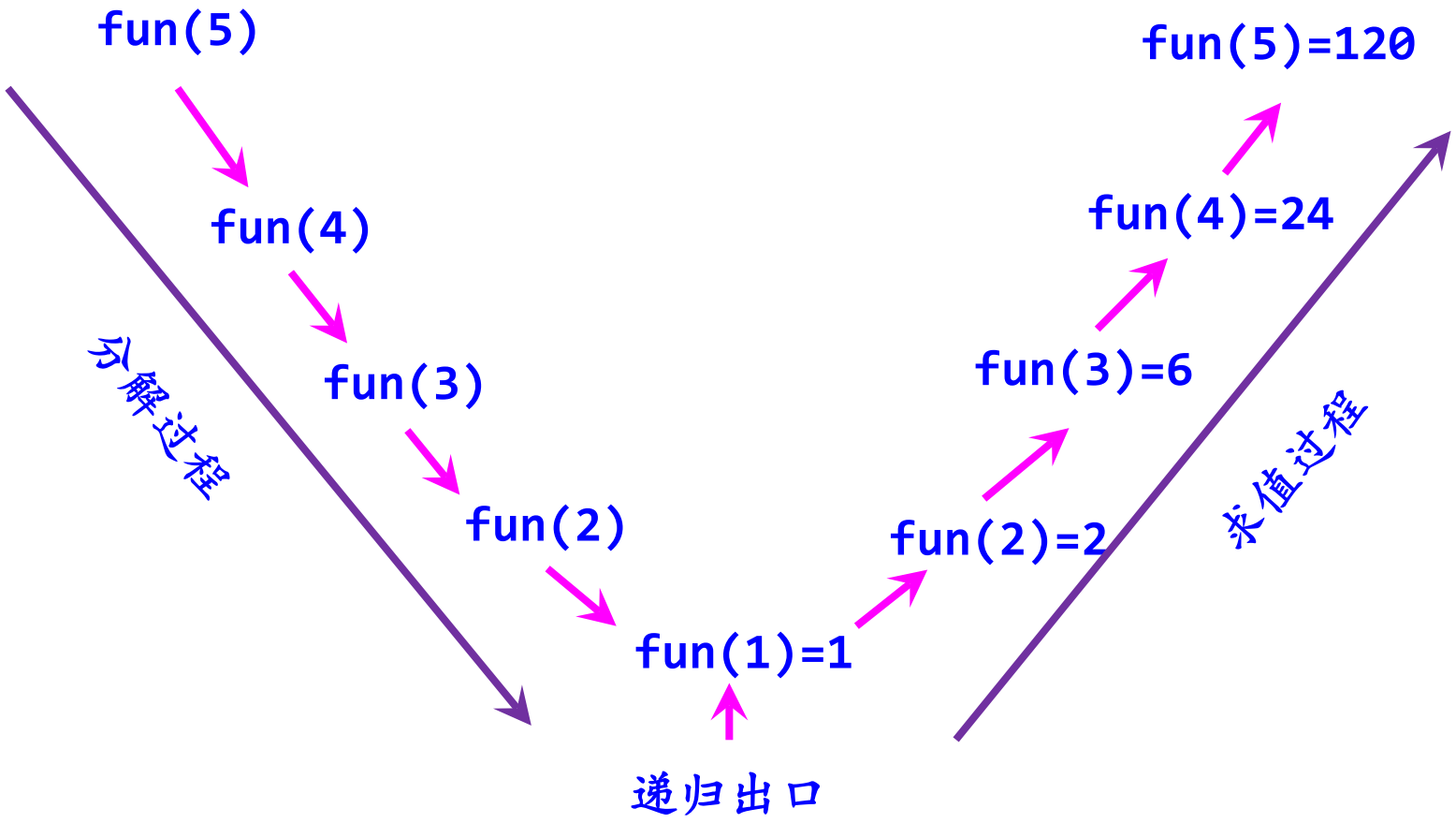
$f(s_1)$

遇到递归出口可以直接求解问题。求值过程：

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

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)即5!的过程如下：

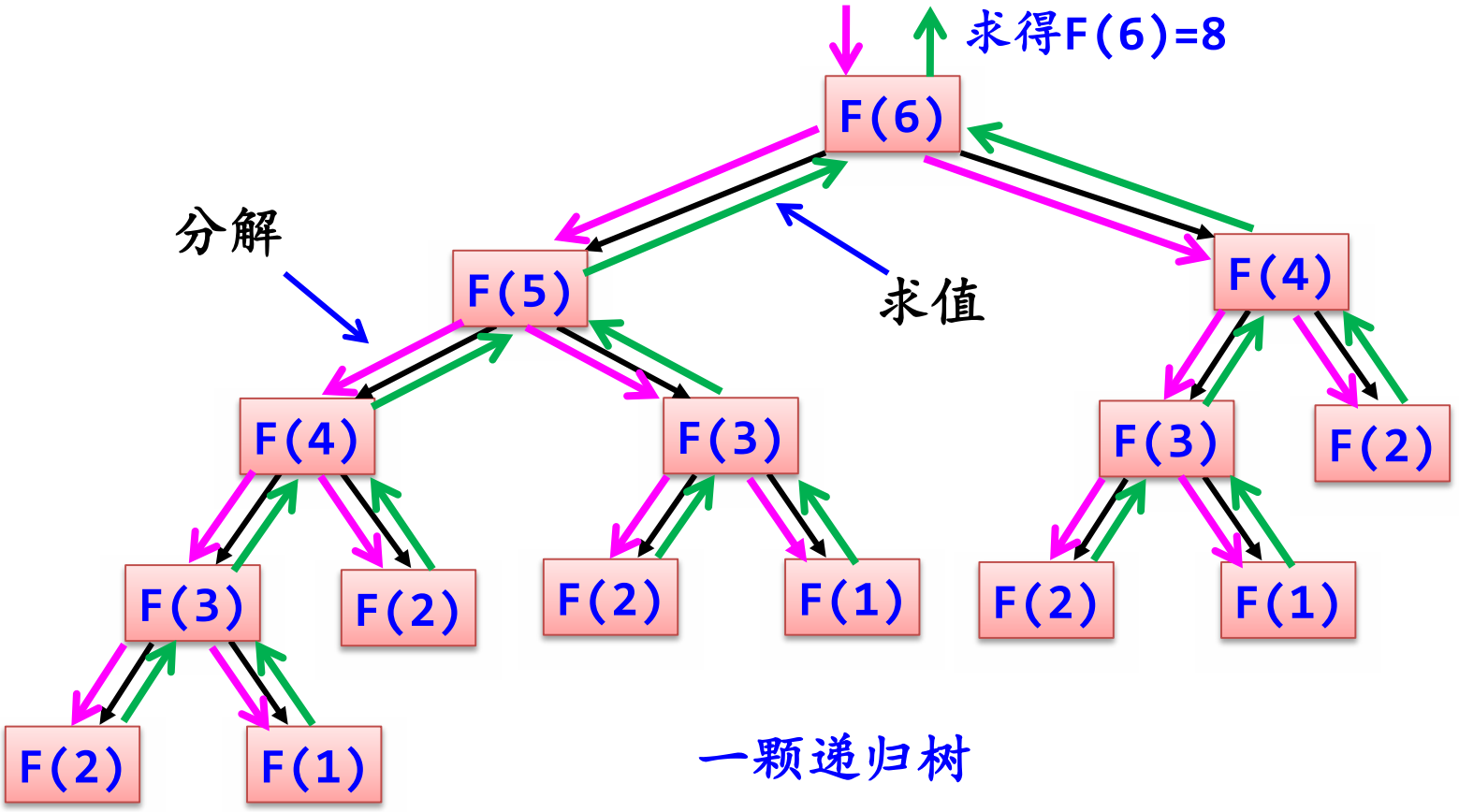


对于复杂的递归问题，在求解时需要进行**多次分解和求值**。

例如：

F(1)=1，F(2)=1

F(*n*)=F(*n*-1)+F(*n*-2)　*n*>2　　求F(6) = ?



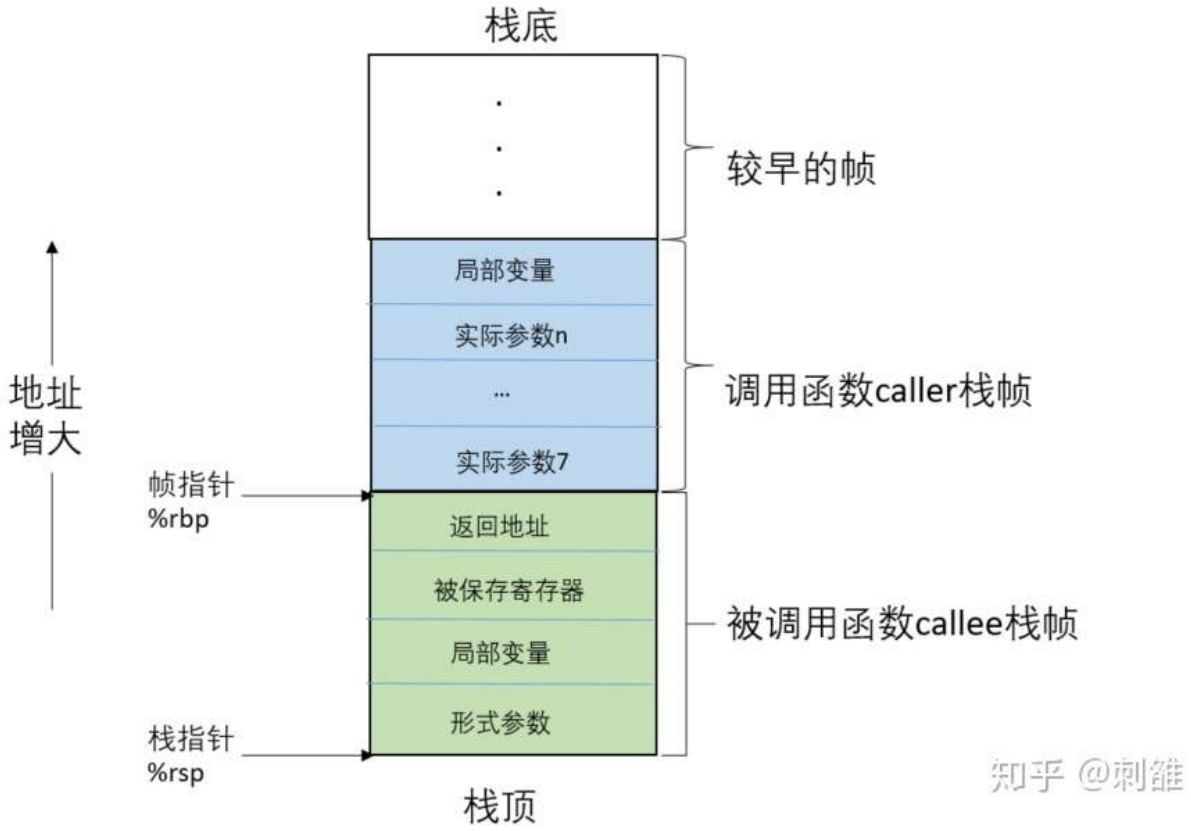
一颗递归树

5.2 递归和栈

5.2.1 函数调用栈

- 函数执行是通过系统栈实现的。系统栈分为若干个栈帧（stack frame）。
- 当一个函数在运行时，需要为它在堆栈中创建一个栈帧用来记录运行时产生的相关信息，因此每个函数在执行前都会创建一个栈帧，在它返回时会销毁该栈帧。
- 一次函数调用相关的数据保存在栈帧中。体现先进后出的特点！

- 计算机的一块内存区域作为栈，高地址向低地址扩展
- 每个函数可以在栈上申请一块内存区域作为函数的存储空间

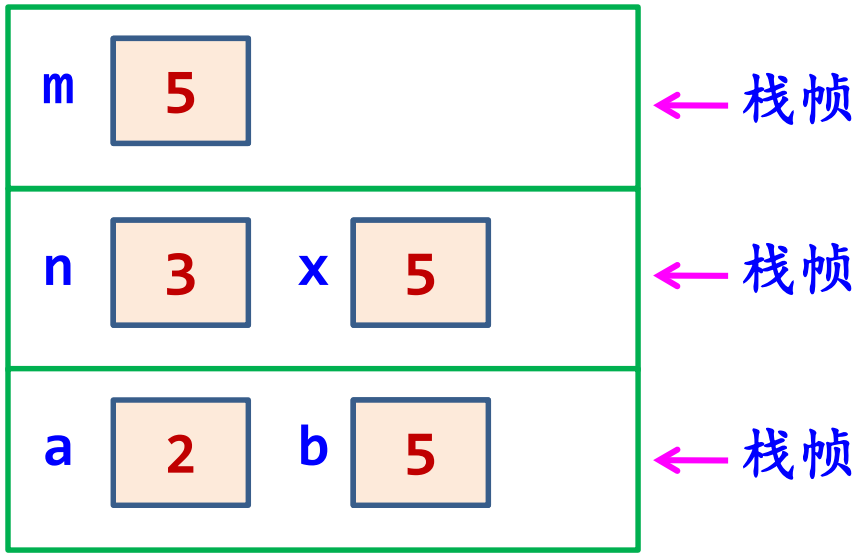


《计算机体系结构》

例如：

```
int fun1(int n)
{   int x;
    n++;
    x=fun2(n);
    return x;
}
int fun2(int m)
{   m+=2;
    return m;
}
void main()
{   int a=2, b;
    b=fun1(a);
    printf("b=%d\n", b);
}
```

执行过程：



屏幕输出b=5

程序执行完毕

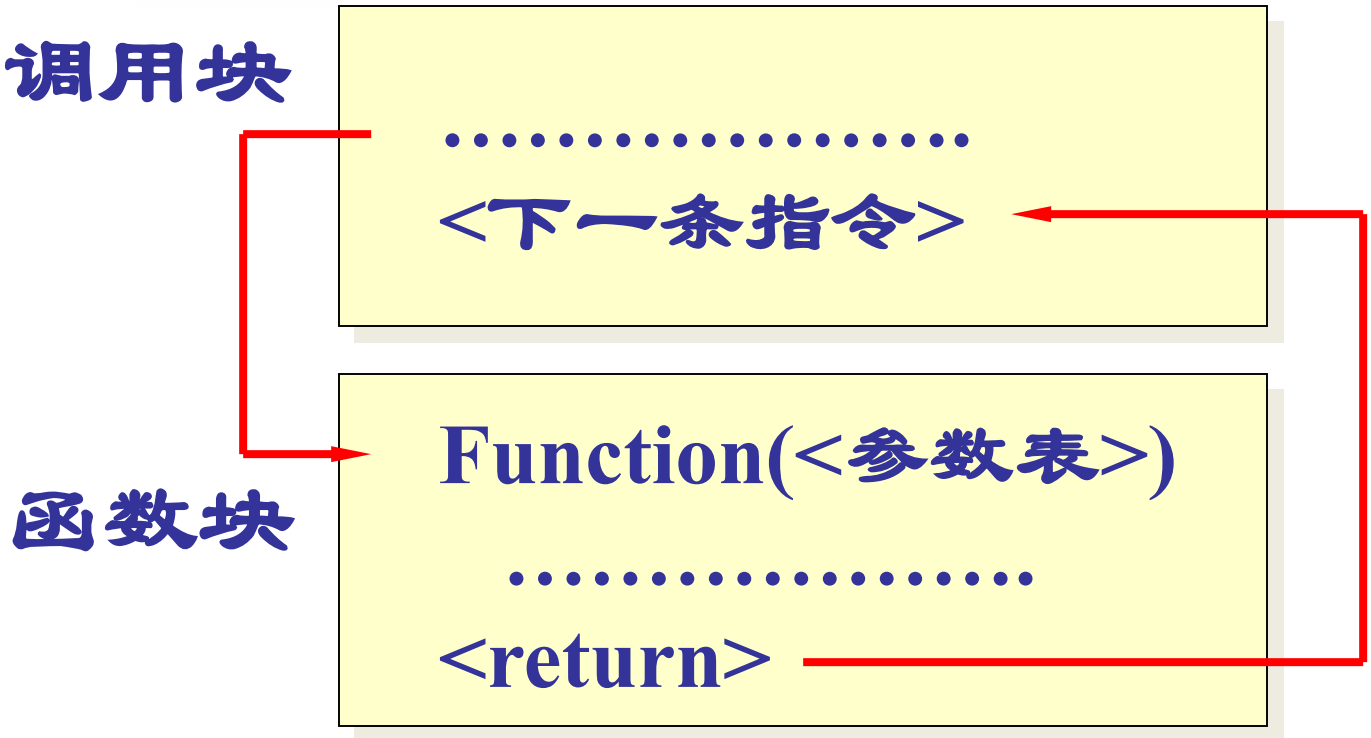
5.2.2 递归函数的实现

- 递归是函数调用的一种特殊情况，即它是调用自身代码。
- 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，因而也就保证了各个复制件执行时的独立性。

递归工作栈

- 每一次递归调用时，需要为过程中使用的参数、局部变量等另外分配存储空间。
- 每层递归调用需分配的空间形成递归工作记录，按后进先出的栈组织。

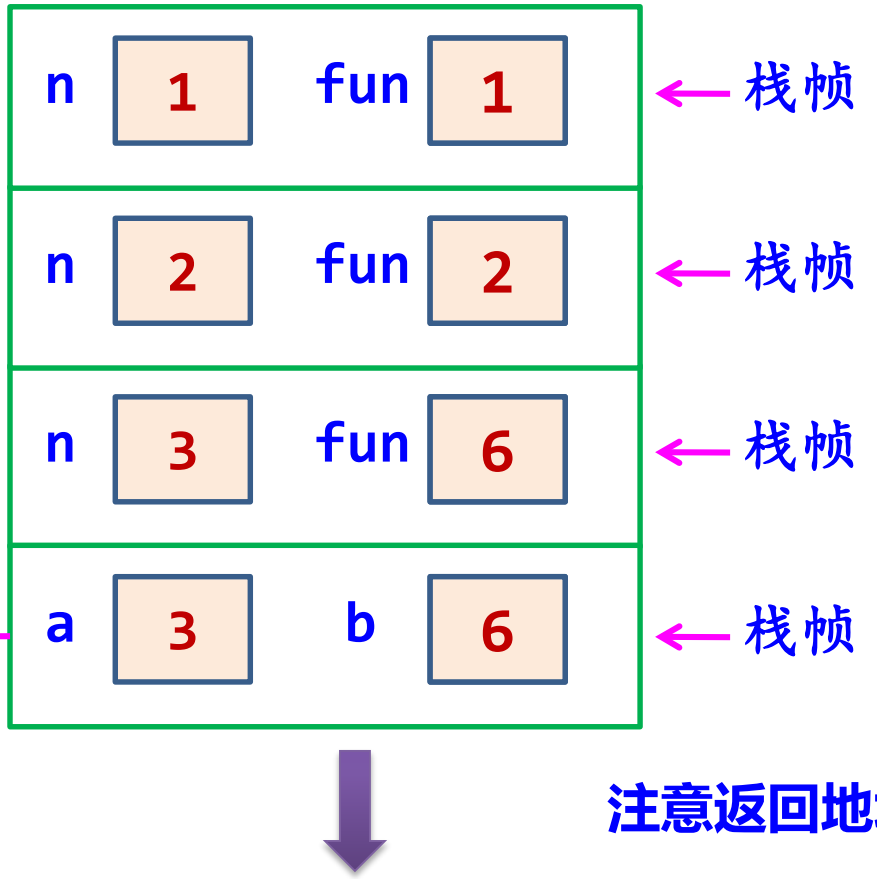
函数递归时的活动记录



执行过程：

例如：

```
int fun(int n)
{  if (n==1)
      return 1;
   else;
      return fun(n-1)*n;
}
void main()
{  int a=3, b;
   b=fun(a);
   printf("b=%d\n", b);
}
```



注意返回地址

屏幕输出b=6

程序执行完毕

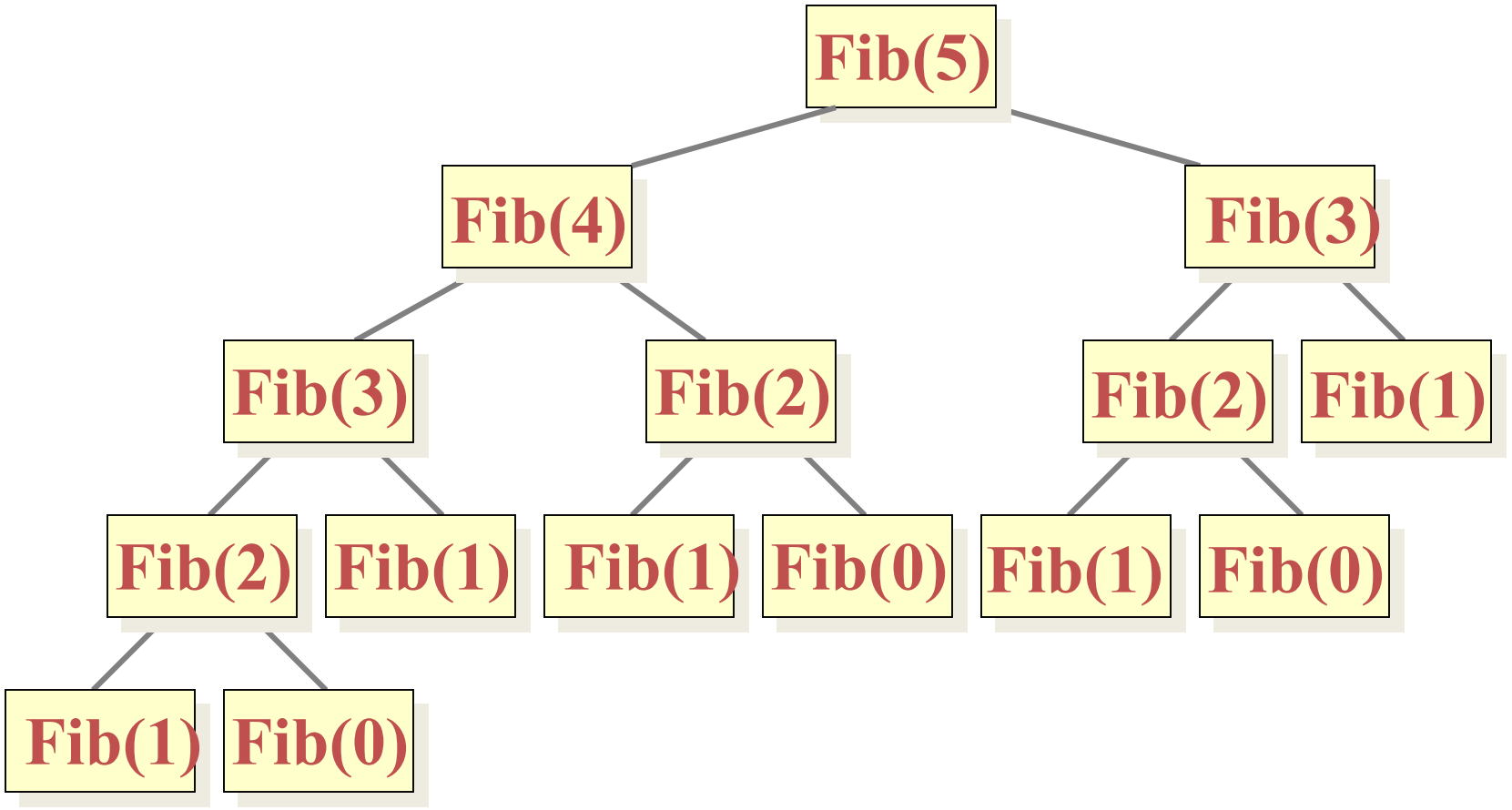
计算斐波那契数列的函数Fib(n)的定义

$$Fib(n)=\begin{cases} n, & n=0 \\ Fib(n-1)+Fib(n-2), & n>1 \end{cases}$$

如 $F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5$

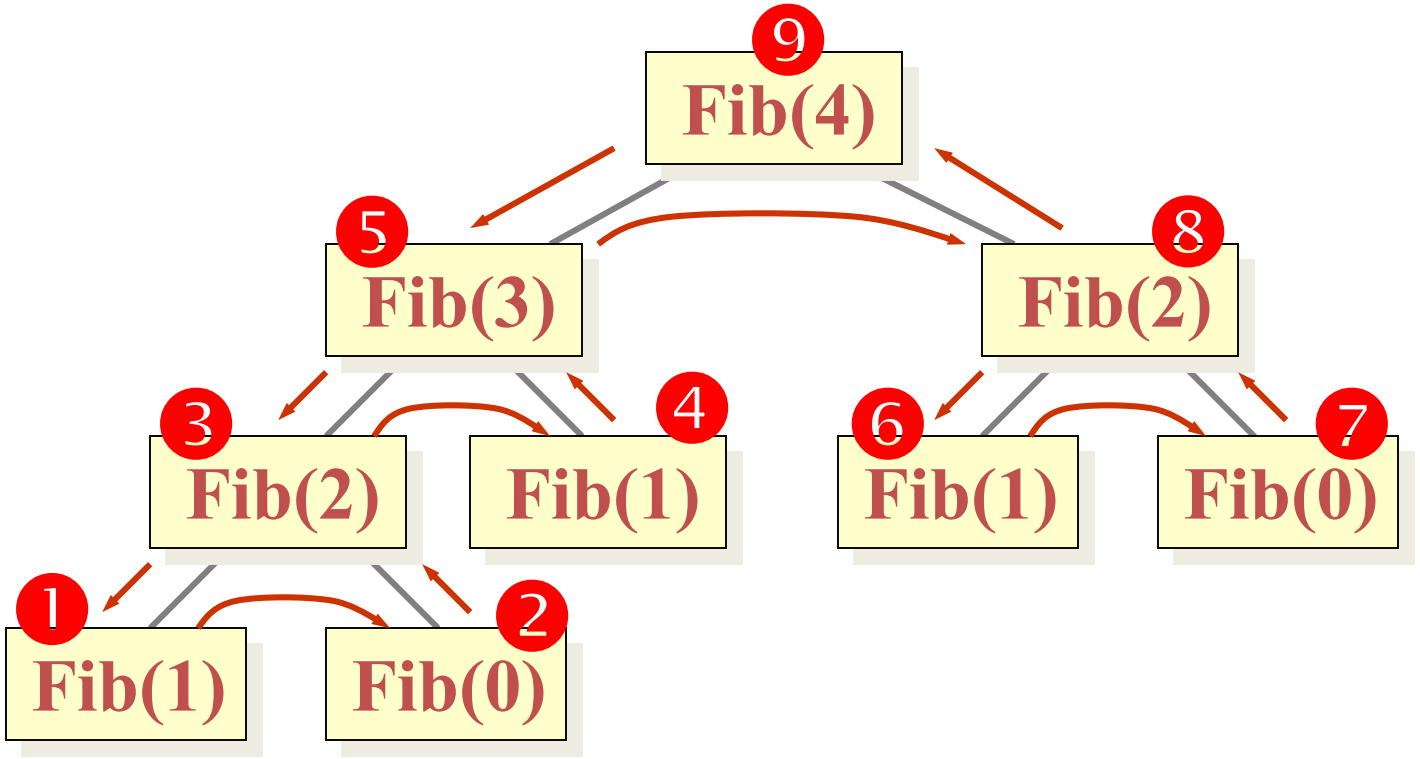
求解斐波那契数列的递归算法

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

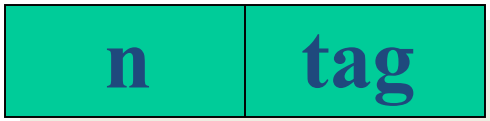


斐波那契数列的递归调用树

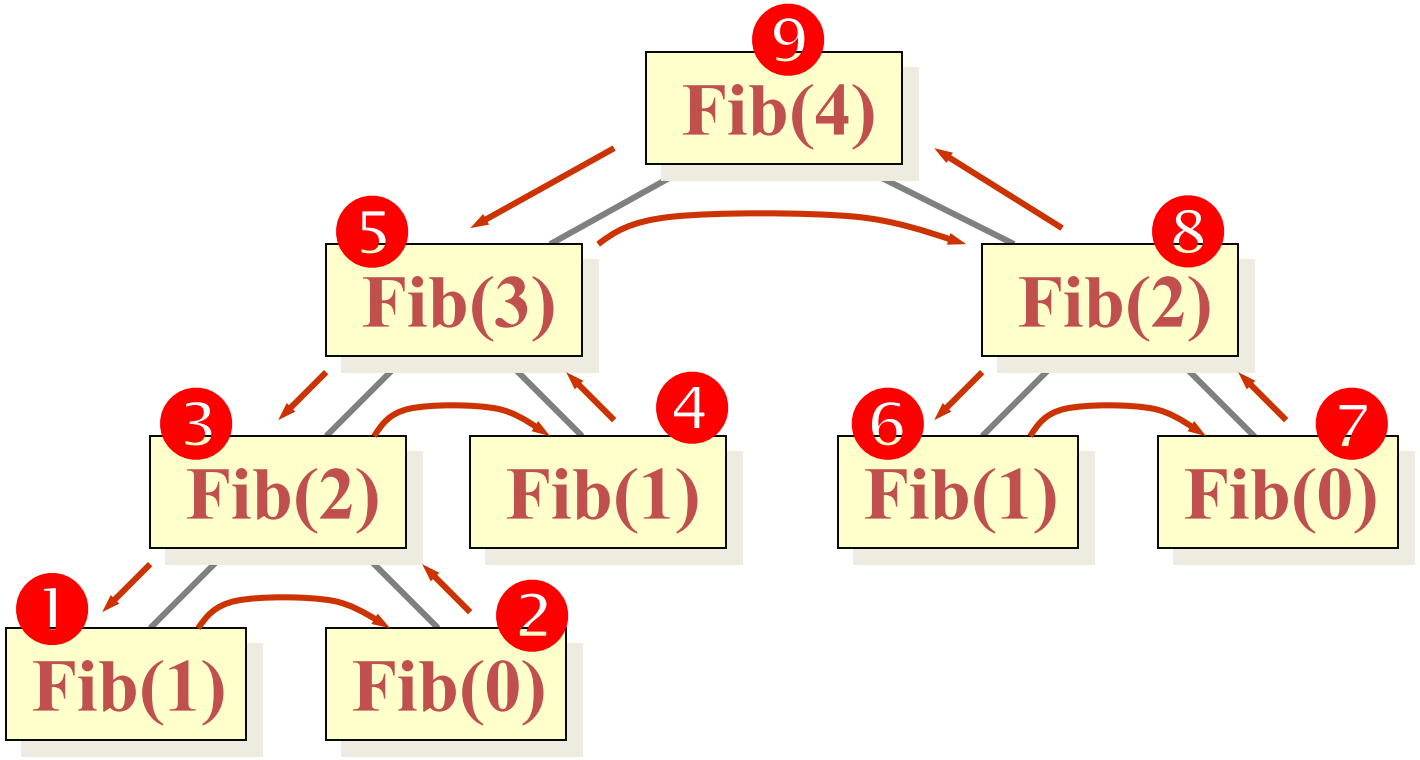
调用次数 NumCall(k) = 2*Fib(k+1) - 1。

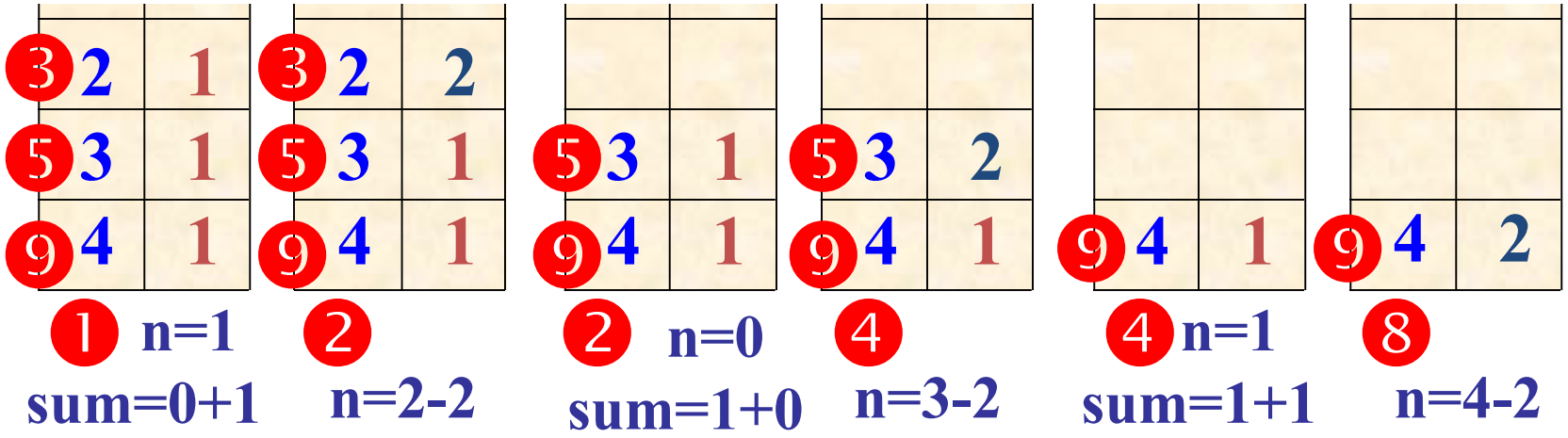


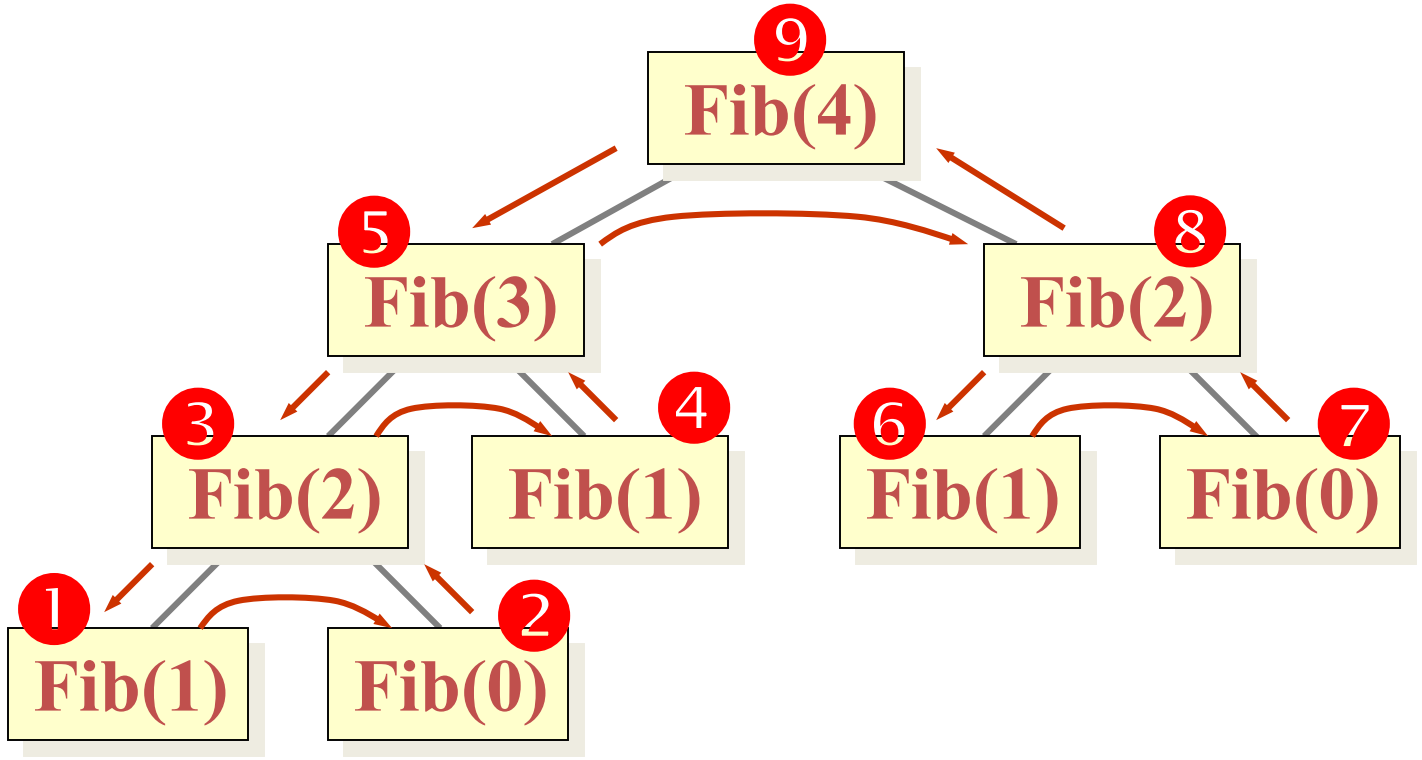
栈结点

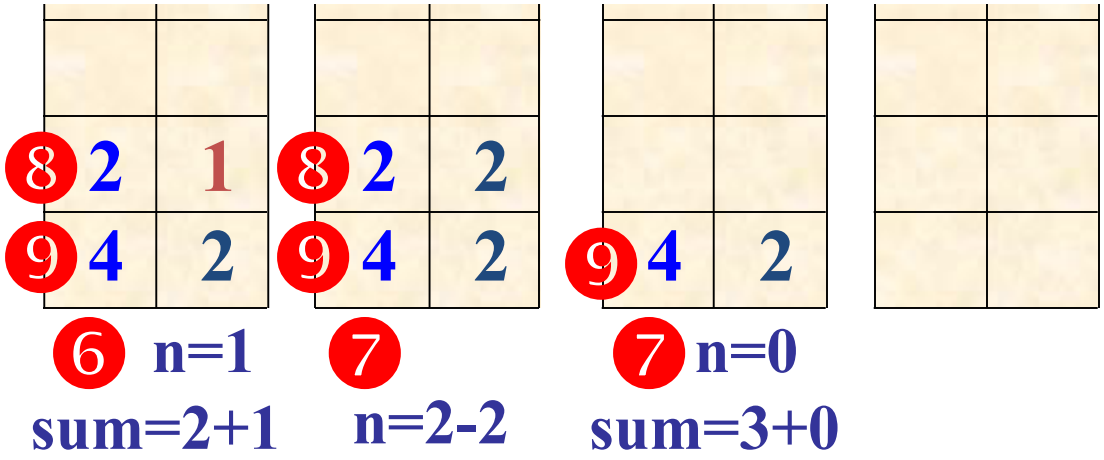


tag = 1, 向左递归; tag = 2, 向右递归


9
Fib(4)
5
Fib(3)
8
Fib(2)
3
Fib(2)
4
Fib(1)
6
Fib(1)
7
Fib(0)
1
Fib(1)
2
Fib(0)


3 2 1
5 3 1
9 4 1
1 n=1
sum=0+1
3 2 2
5 3 1
9 4 1
2 n=2-2
5 3 1
9 4 1
2 n=0
sum=1+0
5 3 2
9 4 1
4 n=3-2
9 4 1
4 n=1
sum=1+1
9 4 2
8 n=4-2

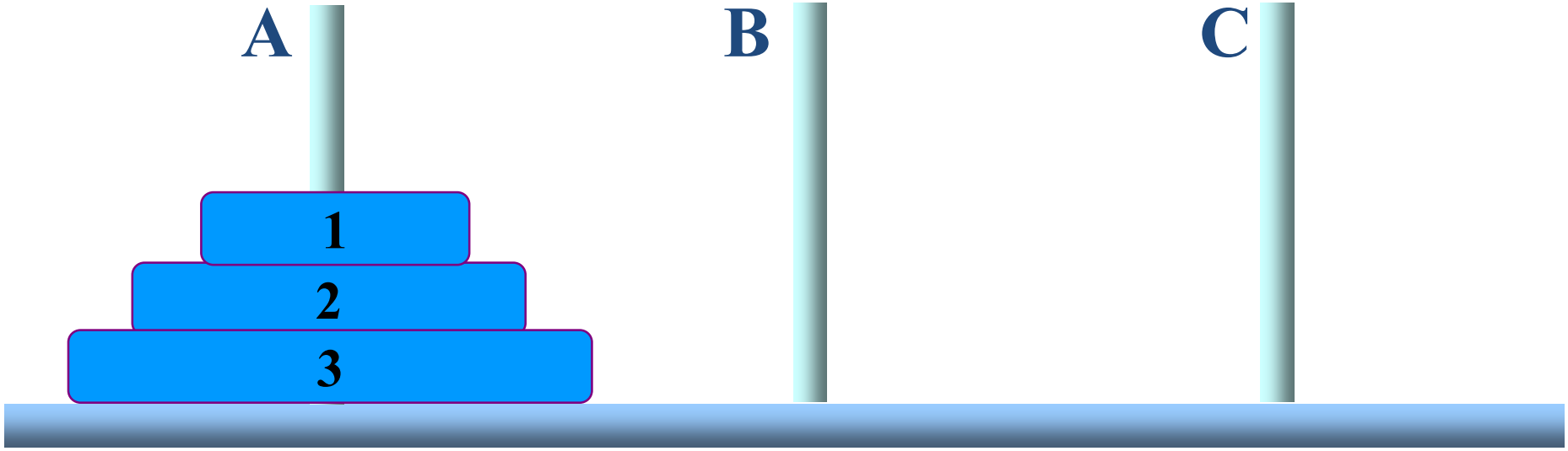

9
Fib(4)
5
Fib(3)
8
Fib(2)
3
Fib(2)
4
Fib(1)
6
Fib(1)
7
Fib(0)
1
Fib(1)
2
Fib(0)


8 2 1
9 4 2
8 2 2
9 4 2
9 4 2
6 n=1
sum=2+1
7
n=2-2
7 n=0
sum=3+0

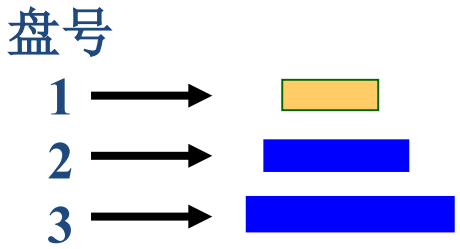
看 A 柱上有3只盘子hanoi(3, A, B, C)的情况：

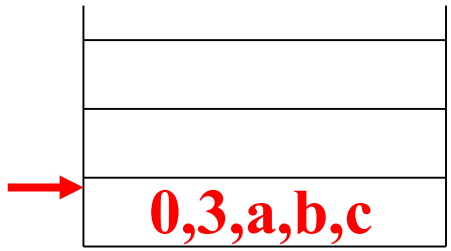
hanoi (3, A, B, C)

- hanoi (2, A, C, B)
  - move (A, 2, B)
  - hanoi (1, C, A, B)
- move (A, 3, C)
- hanoi (2, B, A, C)
  - hanoi (1, B, C, A)
  - move (B, 2, C)
  - hanoi (1, A, B, C)



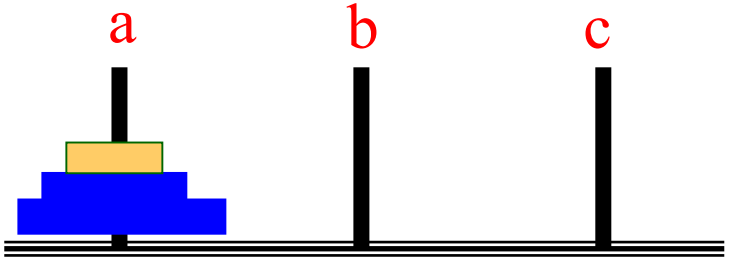
```
0 void Hanoi(int n, char x, char y, char z)
1 {
2   if(n = = 1)
3     move(x, 1, z);
4   else{
5         Hanoi(n-1, x, z, y);
6           move(x, n, z);
7           Hanoi(n-1, y, x, z);
8    }
9 }
```





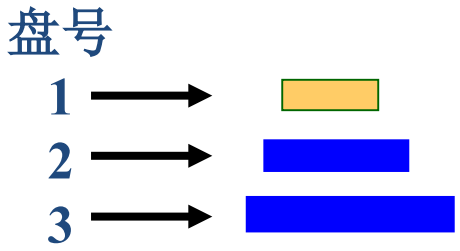
1　　1,2,4,5

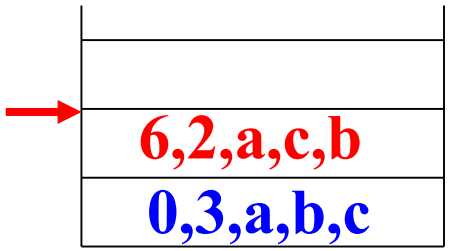
递归层次　运行语句序号　递归工作栈状态（返址，盘号，x,y,z）

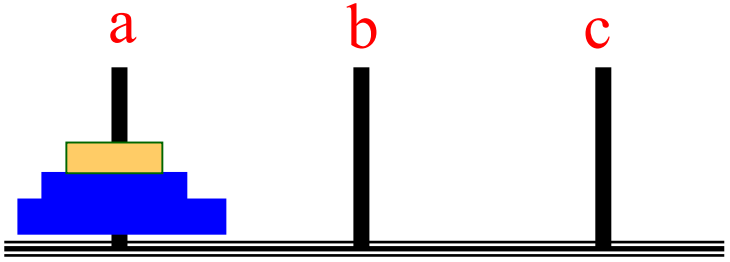


塔与圆盘的状态

```
0 void Hanoi(int n, char x, char y, char z)
1 {
2   if(n = = 1)
3     move(x, 1, z);
4   else{
5         Hanoi(n-1, x, z, y);
6           move(x, n, z);
7           Hanoi(n-1, y, x, z);
8           }
9 }
```



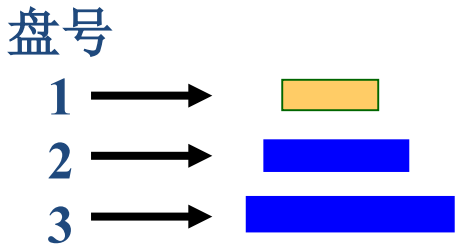


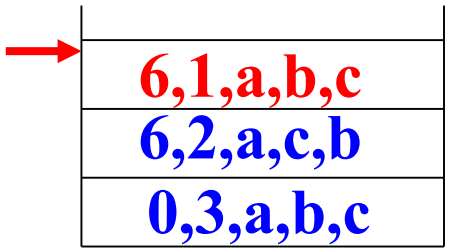


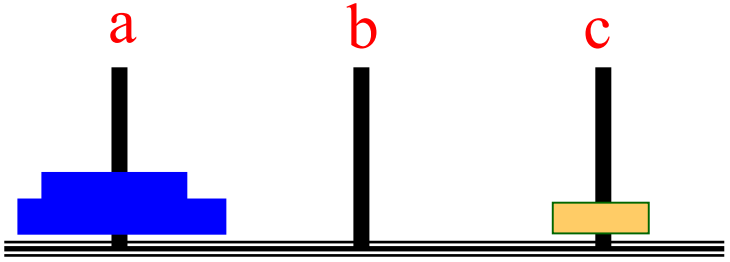
递归层次	运行语句序号	递归工作栈状态（返址，盘号，x,y,z）
2	1,2,4,5	

塔与圆盘的状态

```
0 void Hanoi(int n, char x, char y, char z)
1 {
2   if(n = = 1)
3     move(x, 1, z);
4   else{
5         Hanoi(n-1, x, z, y);
6           move(x, n, z);
7           Hanoi(n-1, y, x, z);
8           }
9 }
```



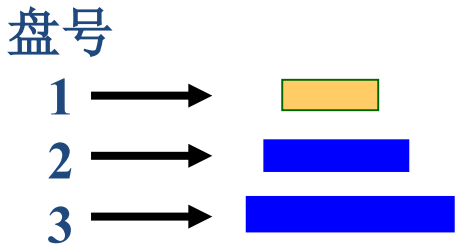


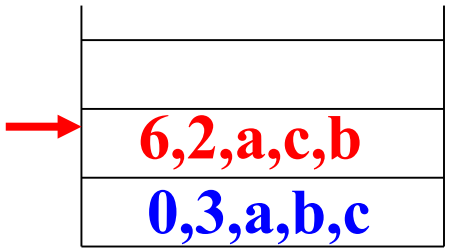


递归层次	运行语句序号	递归工作栈状态（返址，盘号，x,y,z）
3	1,2,3,9	

塔与圆盘的状态

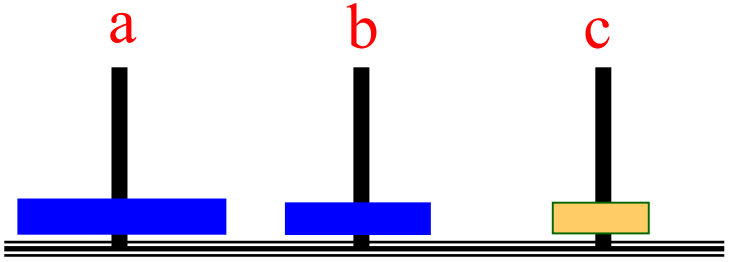
```
0 void Hanoi(int n, char x, char y, char z)
1 {
2   if(n = = 1)
3     move(x, 1, z);
4   else{
5         Hanoi(n-1, x, z, y);
6           move(x, n, z);
7           Hanoi(n-1, y, x, z);
8           }
9 }
```





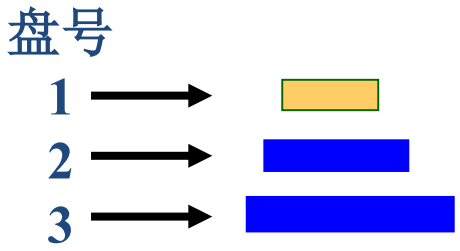
2　　6,7

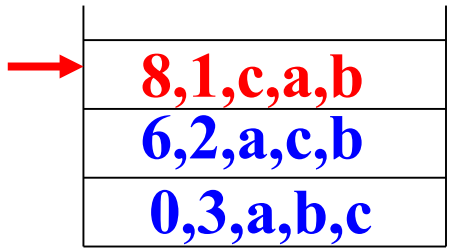
递归层次　运行语句序号　递归工作栈状态（返址，盘号，x,y,z）



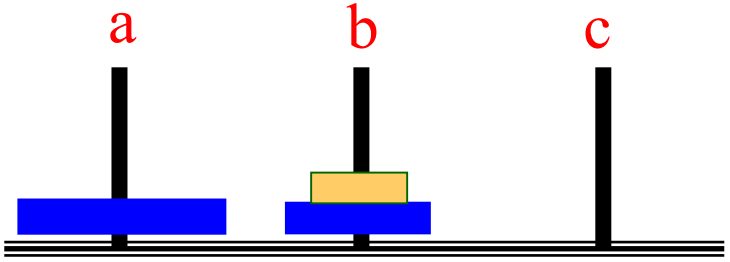
塔与圆盘的状态

```
0 void Hanoi(int n, char x, char y, char z)
1 {
2   if(n = = 1)
3     move(x, 1, z);
4   else{
5         Hanoi(n-1, x, z, y);
6           move(x, n, z);
7           Hanoi(n-1, y, x, z);
8           }
9 }
```



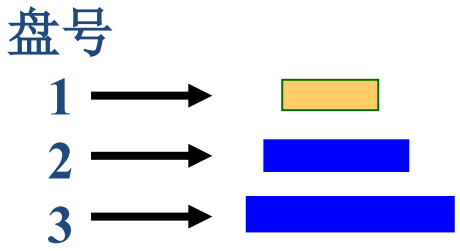


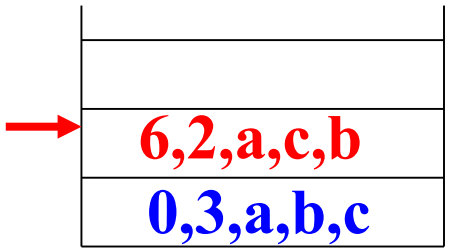
递归层次	运行语句序号	递归工作栈状态（返址，盘号，x,y,z）
3	1,2,3,9	



塔与圆盘的状态

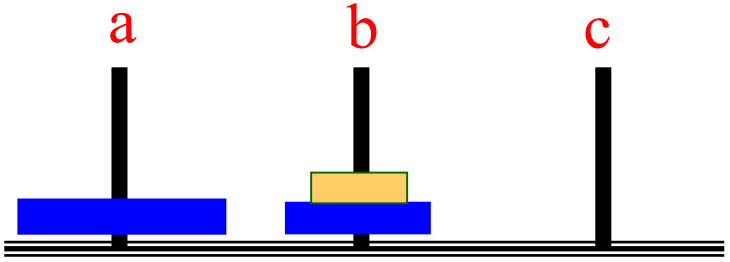
```
0 void Hanoi(int n, char x, char y, char z)
1 {
2   if(n = = 1)
3     move(x, 1, z);
4   else{
5         Hanoi(n-1, x, z, y);
6           move(x, n, z);
7           Hanoi(n-1, y, x, z);
8           }
9 }
```





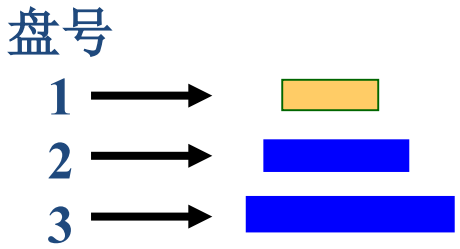
2　　8,9

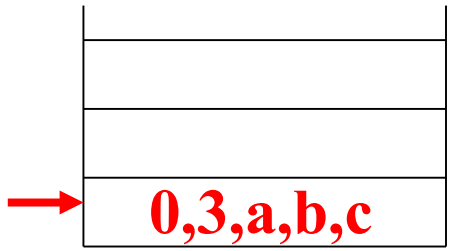
递归层次　运行语句序号　递归工作栈状态（返址，盘号，x,y,z）



塔与圆盘的状态

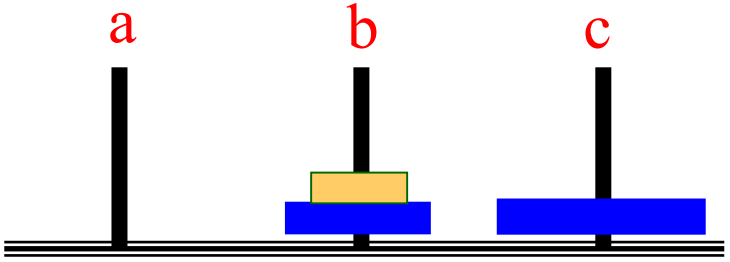
```
0 void Hanoi(int n, char x, char y, char z)
1 {
2   if(n = = 1)
3     move(x, 1, z);
4   else{
5         Hanoi(n-1, x, z, y);
6           move(x, n, z);
7           Hanoi(n-1, y, x, z);
8           }
9 }
```





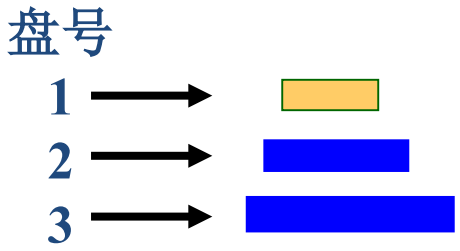
1　　6,7

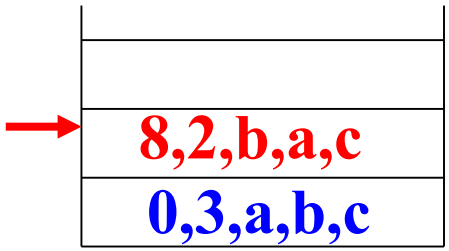
递归层次　运行语句序号　递归工作栈状态（返址，盘号，x,y,z）



塔与圆盘的状态

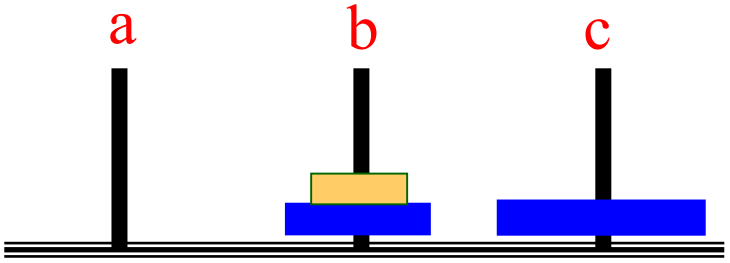
```
0 void Hanoi(int n, char x, char y, char z)
1 {
2   if(n = = 1)
3     move(x, 1, z);
4   else{
5         Hanoi(n-1, x, z, y);
6           move(x, n, z);
7           Hanoi(n-1, y, x, z);
8           }
9 }
```





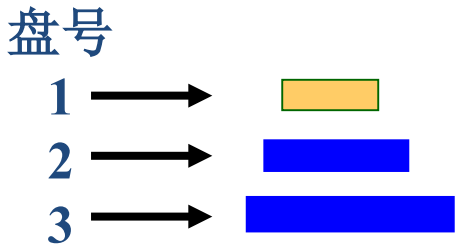
2　　1,2,4,5

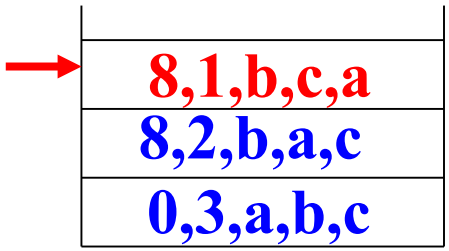
递归层次　运行语句序号　递归工作栈状态（返址，盘号，x,y,z）



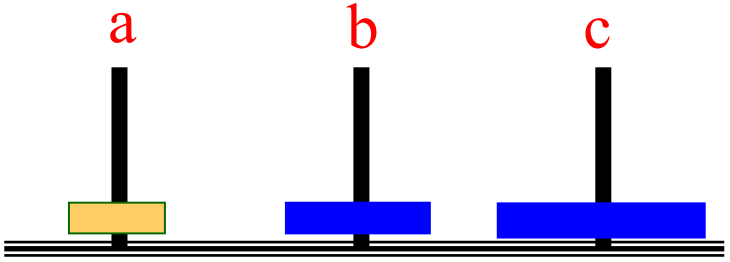
塔与圆盘的状态

```
0 void Hanoi(int n, char x, char y, char z)
1 {
2   if(n = = 1)
3     move(x, 1, z);
4   else{
5         Hanoi(n-1, x, z, y);
6           move(x, n, z);
7           Hanoi(n-1, y, x, z);
8           }
9 }
```



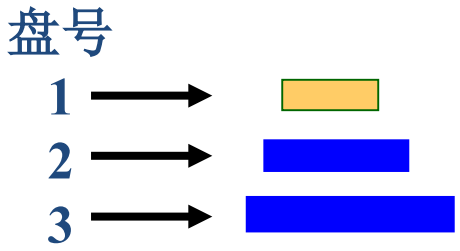


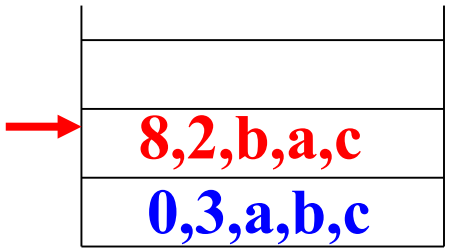
递归层次	运行语句序号	递归工作栈状态（返址，盘号，x,y,z）
3	1,2,3,9	



塔与圆盘的状态

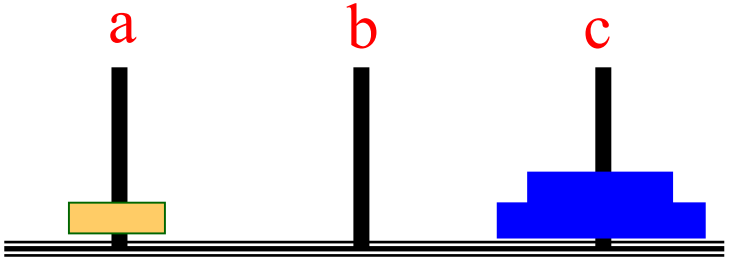
```
0 void Hanoi(int n, char x, char y, char z)
1 {
2   if(n = = 1)
3     move(x, 1, z);
4   else{
5         Hanoi(n-1, x, z, y);
6           move(x, n, z);
7           Hanoi(n-1, y, x, z);
8           }
9 }
```





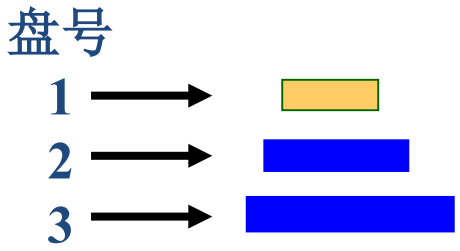
2　　6,7

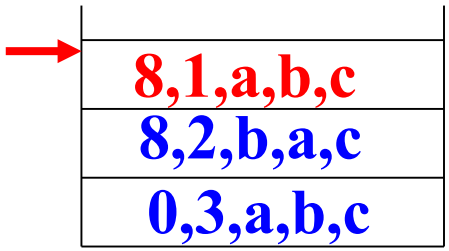
递归层次　运行语句序号　递归工作栈状态（返址，盘号，x,y,z）



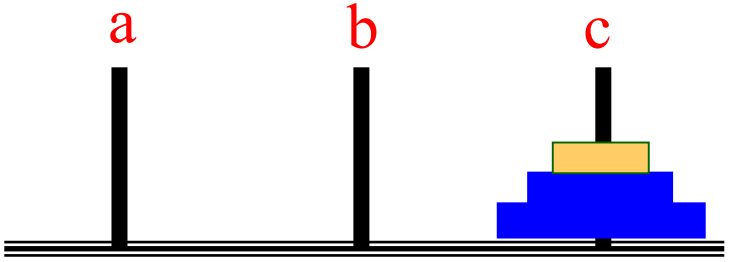
塔与圆盘的状态

```
0 void Hanoi(int n, char x, char y, char z)
1 {
2   if(n = = 1)
3     move(x, 1, z);
4   else{
5         Hanoi(n-1, x, z, y);
6           move(x, n, z);
7           Hanoi(n-1, y, x, z);
8           }
9 }
```



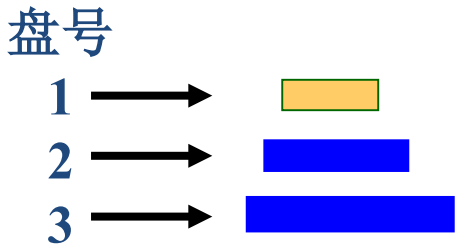


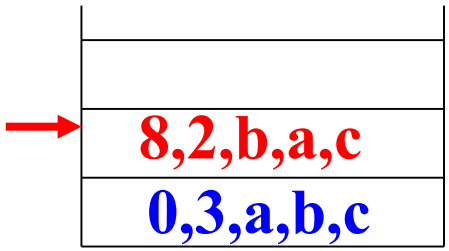
递归层次	运行语句序号	递归工作栈状态（返址，盘号，x,y,z）
3	1,2,3,9	



塔与圆盘的状态

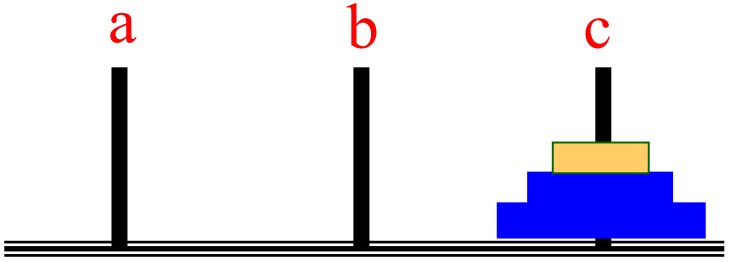
```
0 void Hanoi(int n, char x, char y, char z)
1 {
2   if(n = = 1)
3     move(x, 1, z);
4   else{
5         Hanoi(n-1, x, z, y);
6           move(x, n, z);
7           Hanoi(n-1, y, x, z);
8           }
9 }
```





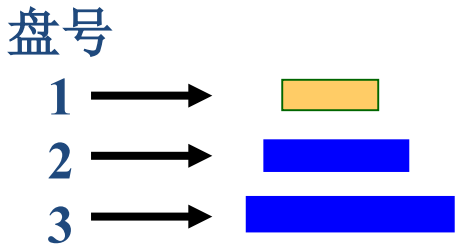
2　　8,9

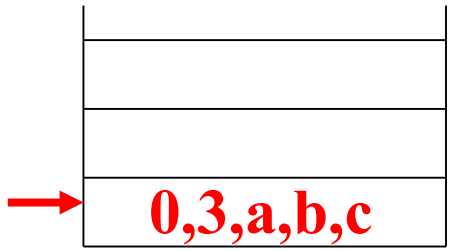
递归层次　运行语句序号　递归工作栈状态（返址，盘号，x,y,z）



塔与圆盘的状态

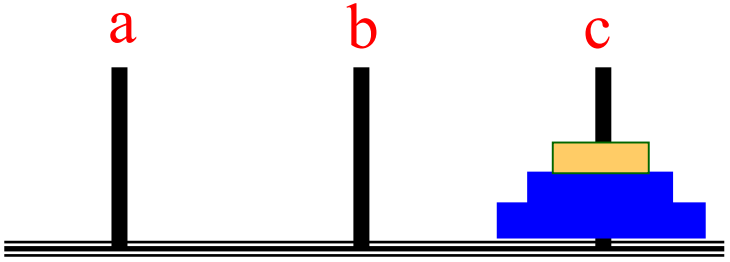
```
0 void Hanoi(int n, char x, char y, char z)
1 {
2   if(n = = 1)
3     move(x, 1, z);
4   else{
5         Hanoi(n-1, x, z, y);
6           move(x, n, z);
7           Hanoi(n-1, y, x, z);
8           }
9 }
```





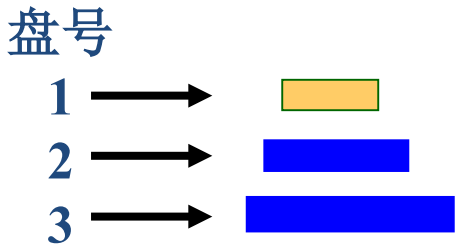
1　　8,9

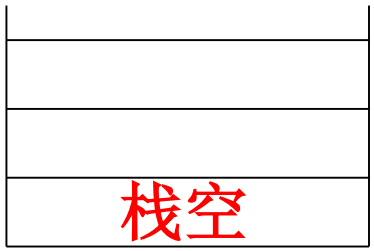
递归层次　运行语句序号　递归工作栈状态（返址，盘号，x,y,z）

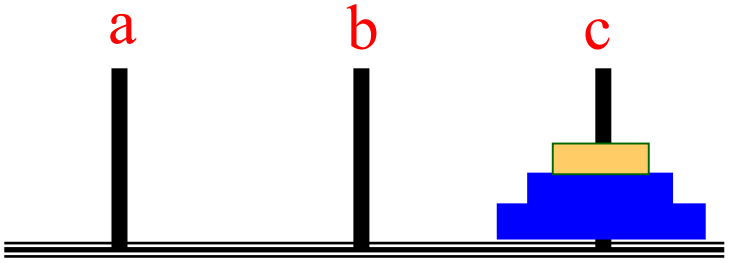


塔与圆盘的状态

```
0 void Hanoi(int n, char x, char y, char z)
1 {
2   if(n = = 1)
3     move(x, 1, z);
4   else{
5         Hanoi(n-1, x, z, y);
6           move(x, n, z);
7           Hanoi(n-1, y, x, z);
8           }
9 }
```







0

递归层次	运行语句序号	递归工作栈状态（返址，盘号，x,y,z）

塔与圆盘的状态

5.2.3 递归到非递归的转换

1 尾递归算法的转换

通常尾递归算法可以通过循环或者迭代方式转换为等价的非递归算法

求Fibonacci数列的第n项

```
int Fib1(int n)
{
   if (n==1 || n==2)
      return(1);
   else
      return(Fib1(n-1)+Fib1(n-2));
}
```

```
int Fib2(int n)
{  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 非尾递归算法的转换

非尾递归算法，在理解递归调用实现过程的基础上，可以用栈模拟递归执行过程，从而将其转换为等价的非递归算法。

Hanoi问题求解递归算法

```
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);
   }
}
```

Hanoi问题求解非递归算法

设计顺序栈的类型如下

```
typedef struct
{   int n;                          //盘片个数
    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;                         //顺序栈中元素类型

typedef struct
{   ElemType data[MaxSize];         //存放元素
    int top;                        //栈顶指针
} StackType;                        //顺序栈类型
```

```
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
```

Hanoi(n, X, Y, Z)任务进栈

```
while (!StackEmpty(st))                    //栈不空循环
{  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
```

Hanoi(n, X, Y, Z)任务

❶ Hanoi(n-1, X, Z, Y)任务

❷ Move(n, X, Z): flag=true

❸ Hanoi(n-1, Y, X, Z)任务

```
    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
}
```

Hanoi(n, X, Y, Z)任务

❶ Hanoi(n-1, X, Z, Y)任务进栈

❷ Move(n, X, Z): flag=true

❸ Hanoi(n-1, Y, Z, X)任务进栈

```
    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
}
```

求解 Move(n, X, Z): flag=true

5.3 递归算法的设计

5.3.1 递归算法设计的步骤

- 设计求解问题的递归模型。
- 转换成对应的递归算法。

求递归模型的步骤如下：

（1）对原问题$f(s)$进行分析，称为“大问题”，假设出合理的“小问题”$f(s')$；

数学归纳法

（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 ⇨ **递归体**。

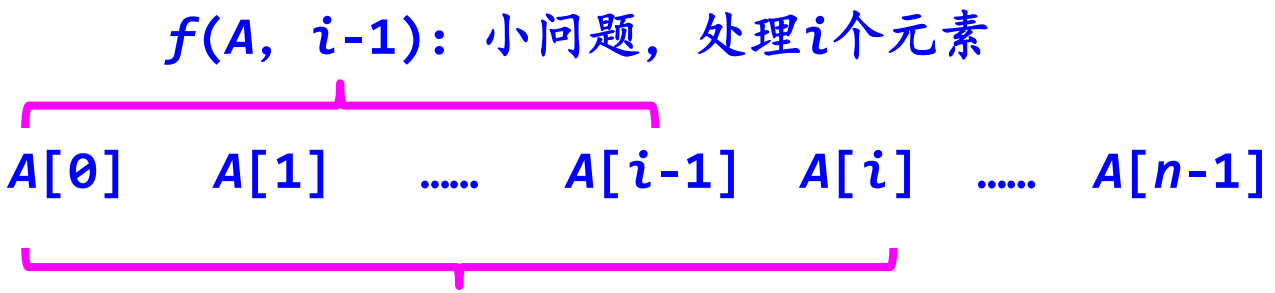
⟺ 假设$n=k-1$时等式成立
求证$n=k$时等式成立

（3）确定一个特定情况（如$f(1)$或$f(0)$）的解 ⇨ **递归出口**。

⟺ 求证$n=1$时等式成立

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)求数组元素A[0]～A[i]（i+1个元素）中的最小值。



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。

当i=0时，只有一个元素，有f(A，i)=A[0]。

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：

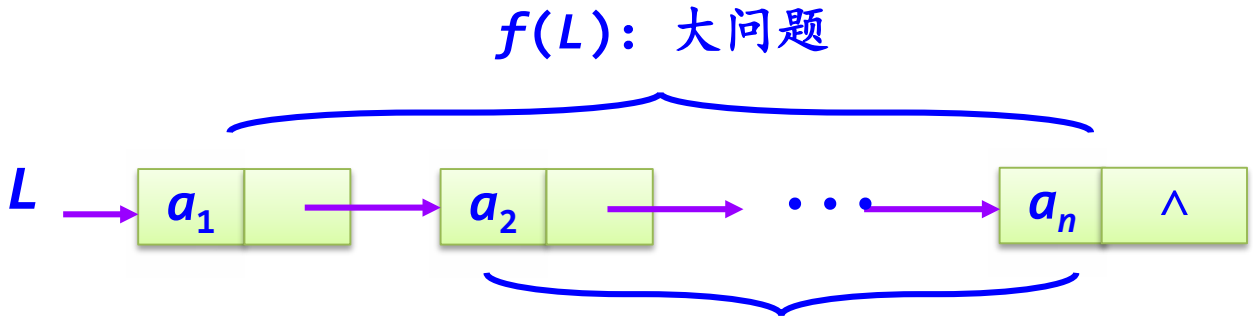
```
float f(float A[], int i)
{  float m;
   if (i==0)
      return A[0];
   else
   {  m=f(A, i-1);
      if (m>A[i])
         return A[i];
      else
         return m;
   }
}
```

递归出口

递归体

5.3.2 基于递归数据结构的递归算法设计

【例】设计不带头结点的单链表的相关递归算法。



把“大问题”转化为若干个相似的“小问题”来求解。

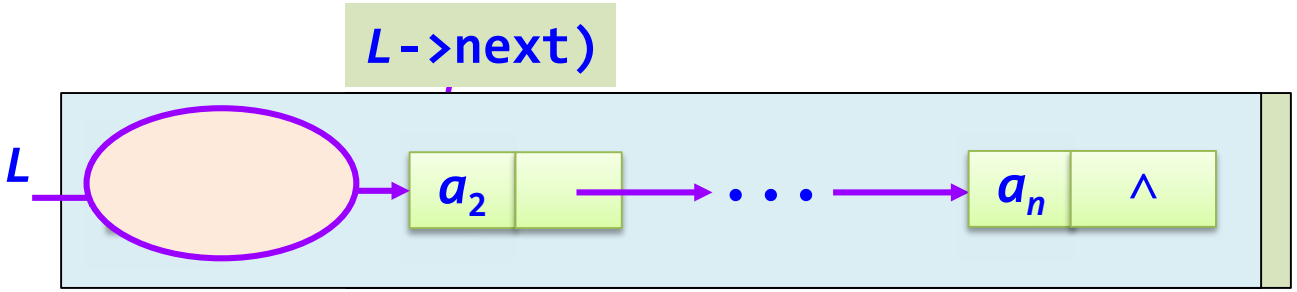
为什么在这里设计单链表的递归算法时不带头结点？

❶ 求单链表中数据结点个数。

设$f(L)$为单链表中数据结点个数。

☑ 空单链表的数据结点个数为0 ⇒ $f(L)=0$ 当L=NULL

☑ 对于非空单链表：



$f(L) = f(L->next) + 1$

递归模型如下：

$f(L)=0$	当L=NULL
$f(L)=f(L->next)+1$	其他情况

求单链表中数据结点个数递归算法如下：

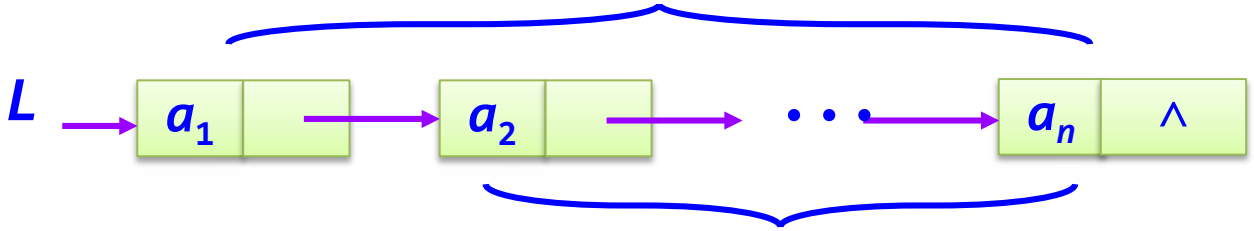
```
int count(LinkNode *L)
{  if (L==NULL)
      return 0;
   else
      return count(L->next)+1;
}
```

不带头结点单链表L

❷ 正向显示所有结点值。

❸ 反向显示所有结点值。

$f(L)$：大问题，输出a_n，…，a_2a_1



$f(L\text{->next})$：小问题，输出a_n，…，a_2

假设$f(L\text{->next})$已求解

$f(L)$ ⇨ $f(L\text{->next})$;输出L->data;

不带头结点单链表L

正向显示所有结点值。

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

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

反向显示所有结点值。

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

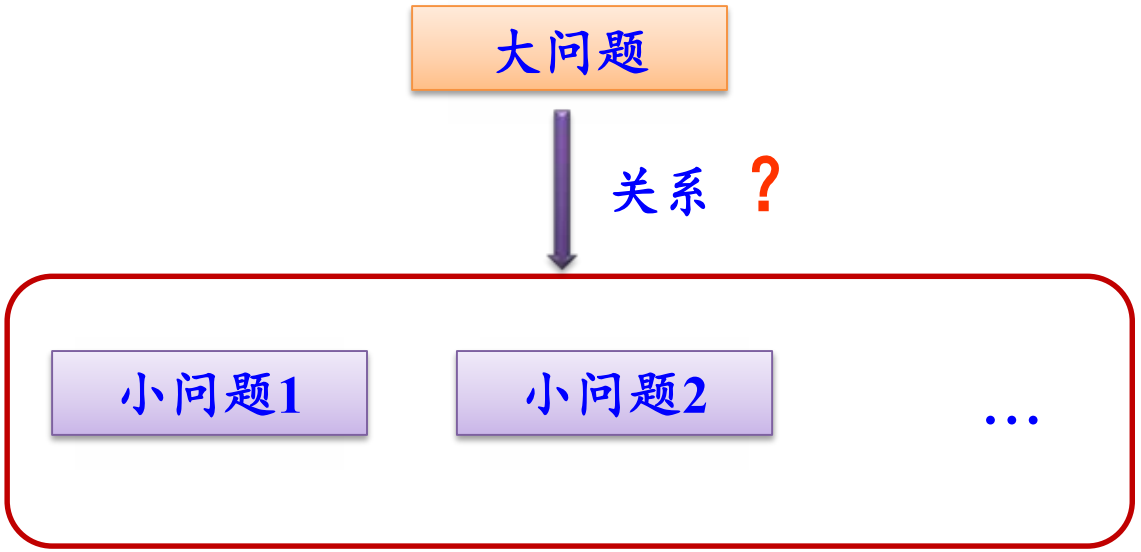
递归算法

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

5.3.3 基于递归求解方法的递归算法设计

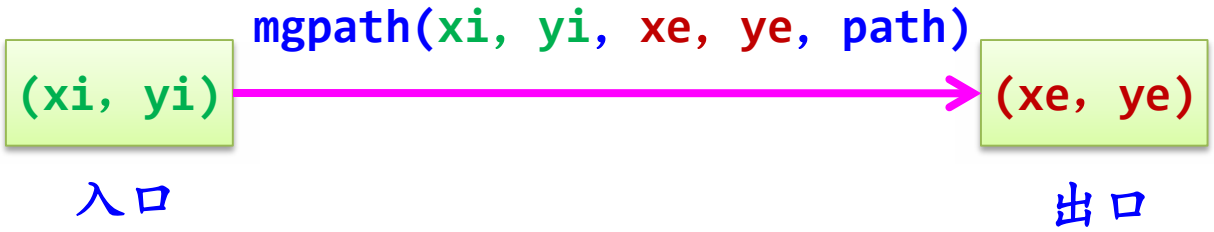
有些问题可以采用递归方法求解（求解方法之一）。

采用递归方法求解问题时，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。



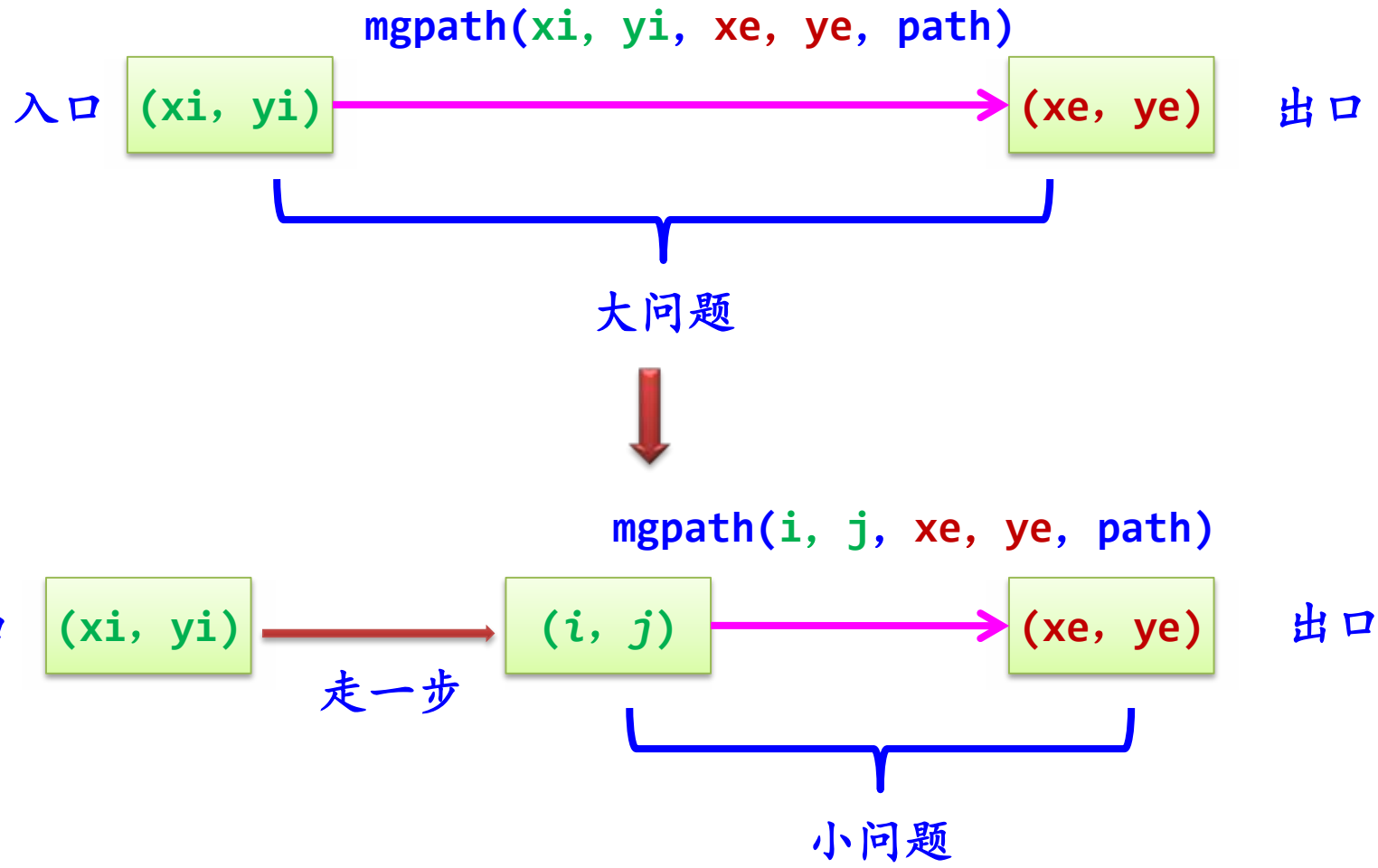
【例】 采用递归算法求解迷宫问题，并输出从入口到出口的所有迷宫路径。

求解问题描述：



mgpath(int xi, int yi, int xe, int ye, PathType path):

求从(xi, yi)到(xe, ye)的迷宫路径，用path变量保存迷宫路径。


mgpath(xi, yi, xe, ye, path)
入口
(xi, yi)
(xe, ye)
出口
大问题
mgpath(i, j, xe, ye, path)
入口
(xi, yi)
走一步
(i, j)
(xe, ye)
出口
小问题

大问题 ≡ 走一步 + 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) ≡ 对于(xi，yi)四周的每一个相邻方块(i，j)：
❶将(xi，yi)添加到path中；
❷ 置mg[xi][yi]=-1；
❸ mgpath(i，j，xe，ye，path)；
❹ path回退一步并置mg[xi][yi]=0；
若(xi，yi)不为出口且可走

在一个“小问题”执行完后回退找所有解

迷宫路径用顺序表存储，它的元素由方块构成的。

其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
```

```
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)
   {  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");
   }
```

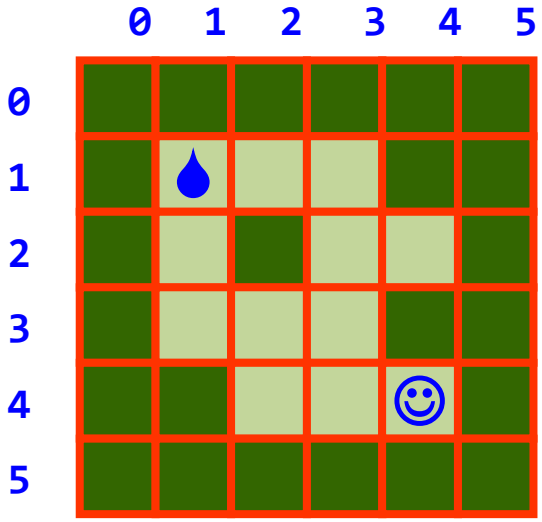
找到了出口，输出路径（递归出口）

```
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)                  //对于(xi, yi)四周的每一个相邻方位di
      {  switch(di)                 //找方位di对应的方块(i, j)
         {
           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;
         }
         ❶ path.data[path.length].i = xi;
           path.data[path.length].j = yi;
           path.length++;           //路径长度增1
         ❷ mg[xi][yi]=-1;           //避免来回重复找路径
```

```
                ❸ mgpath(i, j, xe, ye, path);
                ❹ path.length--;          //回退一个方块
                  mg[xi][yi]=0;           //恢复(xi, yi)为可走
                  di++;
              }   //-while
          }     //- if (mg[xi][yi]==0)
      }    //-递归体
}
```

本算法输出所有的迷宫路径，可以通过进一步比较找出最短路径（可能存在多条最短路径）。

应用

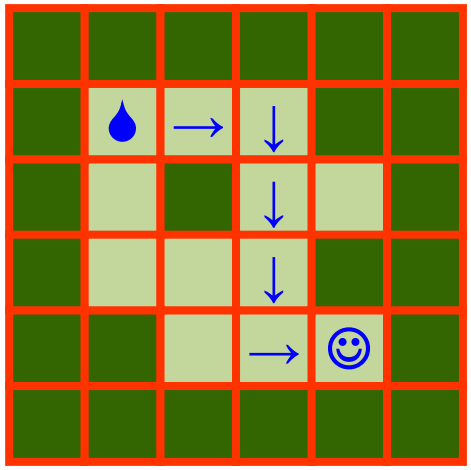



```
int mg[M+2][N+2]=  //M=4, N=4
{ {1, 1, 1, 1, 1, 1},
  {1, 0, 0, 0, 1, 1},
  {1, 0, 1, 0, 0, 1},
  {1, 0, 0, 0, 1 , 1},
  {1, 1, 0, 0, 0, 1},
  {1, 1, 1, 1, 1, 1} };
```

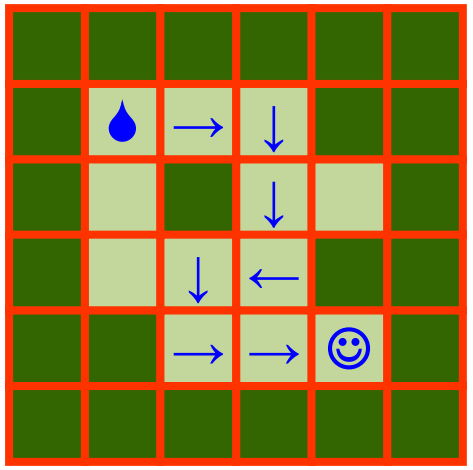
```
void main()
{  PathType path;
   path.length=0;
   mgpath(1, 1, 4, 4, path);
}
```

得到如下4条迷宫路径：

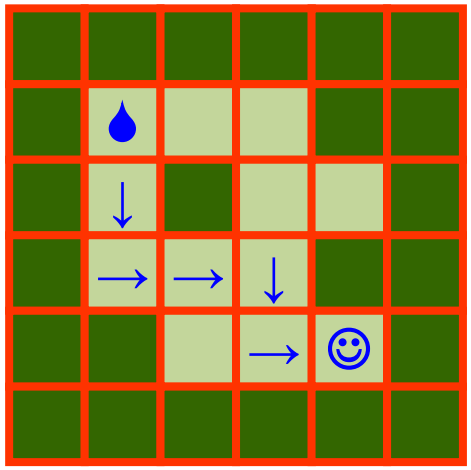
迷宫路径1



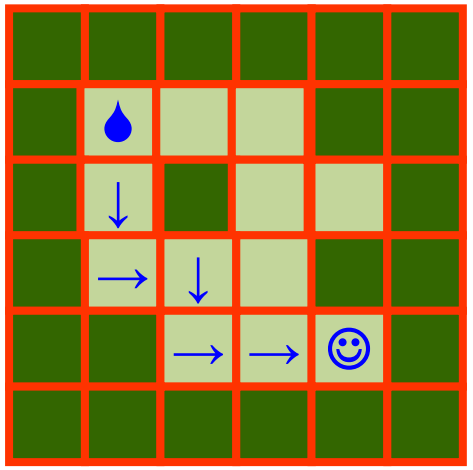
迷宫路径2



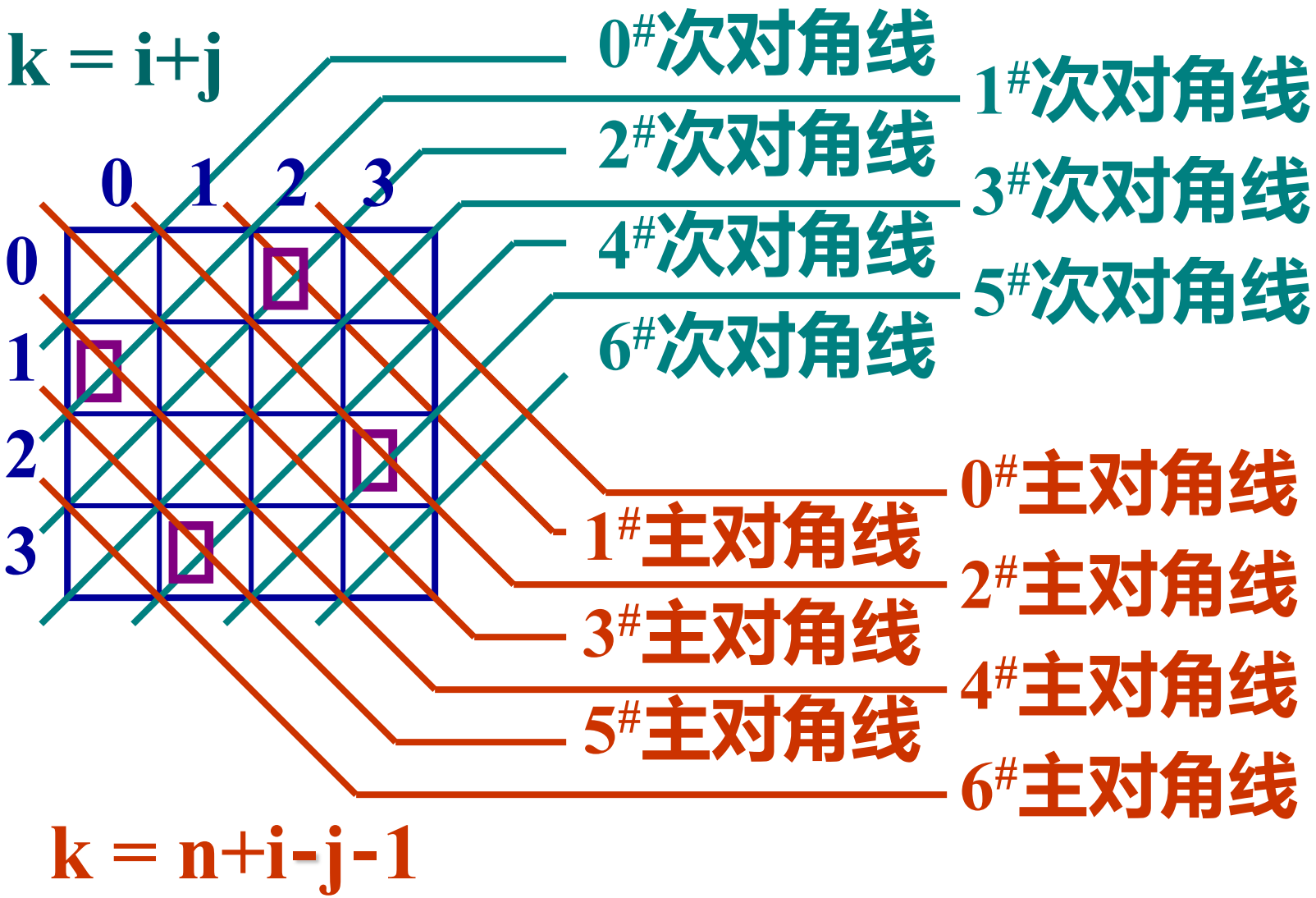
迷宫路径3



迷宫路径4



八皇后问题，一个古老而著名的问题，是回溯算法的典型案例。该问题由国际西洋棋棋手马克斯·贝瑟尔于 1848 年提出：在 8×8 格的国际象棋上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法。高斯认为有 76 种方案。1854 年在柏林的象棋杂志上不同的作者发表了 40 种不同的解，后来有人用图论的方法解出 92 种结果。


k = i+j
0 1 2 3
0
1
2
3
0#次对角线
1#次对角线
2#次对角线
3#次对角线
4#次对角线
5#次对角线
6#次对角线
0#主对角线
1#主对角线
2#主对角线
3#主对角线
4#主对角线
5#主对角线
6#主对角线
k = n+i-j-1

解题思路

- 安放第 i 行皇后时，需要在列的方向从 0 到 n-1 试探 (j = 0, …, n-1)
- 在第 j 列安放一个皇后：
  - 如果在列、主对角线、次对角线方向有其它皇后，则出现攻击，撤消在第 j 列安放的皇后。
  - 如果没有出现攻击，在第 j 列安放的皇后不动，递归安放第 i+1行皇后。

- 设置 4 个数组
  - col [n]： col[i] 标识第 i 列是否安放了皇后
  - md[2n-1] : md[k] 标识第 k 条主对角线是否安放了皇后
  - sd[2n-1] : sd[k] 标识第 k 条次对角线是否安放了皇后
  - q[n] : q[i] 记录第 i 行皇后在第几列

```
void Queen( int i ) {
    for ( int j = 0;  j < n;  j++ ) {
        if ( 第 i 行第 j 列没有攻击 ) {
            在第 i 行第 j 列安放皇后;
            if ( i == n-1 ) 输出一个布局;
            else Queen ( i+1 );
        }
        撤消第 i 行第 j 列的皇后;
    }
}
```

算法明细

```
void Queen( int i ) {
    for ( int j = 0;  j < n;  j++ ) {
      if ( !col[j] && !md[n+i-j-1] && !sd[i+j] )
      {
          /*第 i 行第 j 列没有攻击 */
          col[j] = md[n+i-j-1] = sd[i+j] = 1;
          q[i] = j;
          /*在第 i 行第 j 列安放皇后*/
```

```
        if ( i == n-1 ) {        /*输出一个布局*/
            for ( j = 0; j < n; j++ )
                cout << q[j] << ',';
            cout << endl;
        }
        else  Queen ( i+1 );
    }
    col[j] = md[n+i-j-1] = sd[i+j] = 0;
    q[i] = 0;      /*撤消第 i 行第 j 列的皇后*/
  }
}
```

——本讲完——