第5章　递归

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。

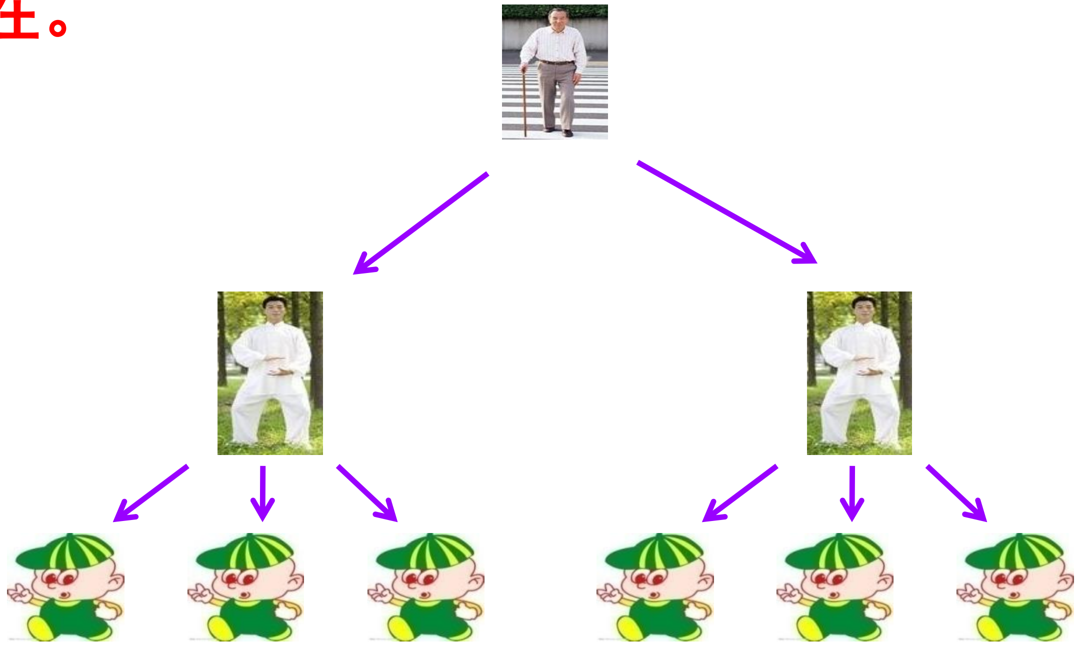
```
int fun(int n)
{     if (n==1)                    //语句1
          return 1;                //语句2
      else                         //语句3
          return n*fun(n-1);       //语句4
}
```

直接递归函数、尾递归

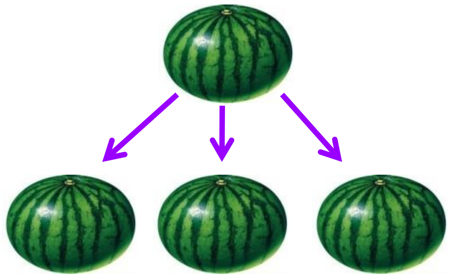
- 尾递归算法：可以用循环语句转换为等价的非递归算法
- 其他递归算法：可以通过栈来转换为等价的非递归算法

递归：无处不在。

实例1：家谱



实例2：种瓜得瓜



第一年种瓜

第n年种瓜

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。

思考题：

请你给出正整数的定义。

- 1是正整数。
- 如果n是正整数，则$n+1$也是正整数。

2、数据结构是递归的

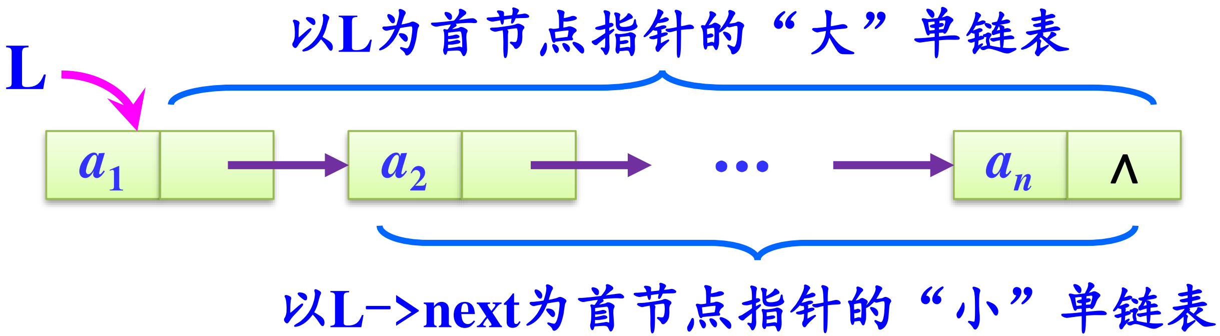
有些数据结构是递归的。例如，第2章中介绍过的单链表就是一种递归数据结构，其节点类型定义如下：

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

指向同类型节点的指针

递归数据结构

不带头节点单链表示意图



体现出这种单链表的递归性。

思考： 如果带有头节点又会怎样呢？？？

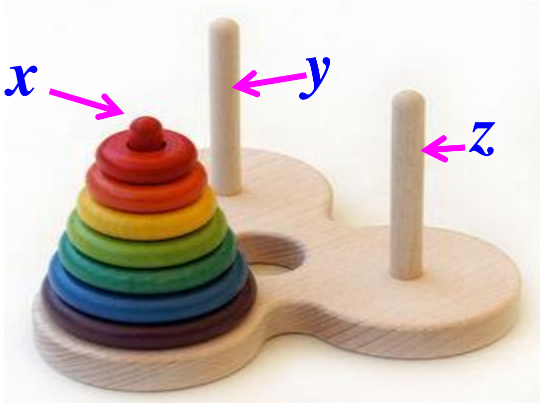
3、问题的求解方法是递归的

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		❶	递归出口
fun(n)=n*fun(n-1)	n>1	❷	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。

递归出口的一般格式如下：

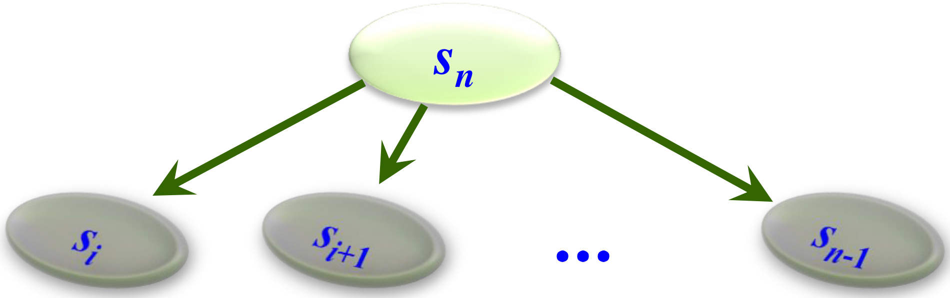
$$f(s_1) = m_1$$

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：

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

g是一个非递归函数



大问题求解

转化

若干个相似子问题求解

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；

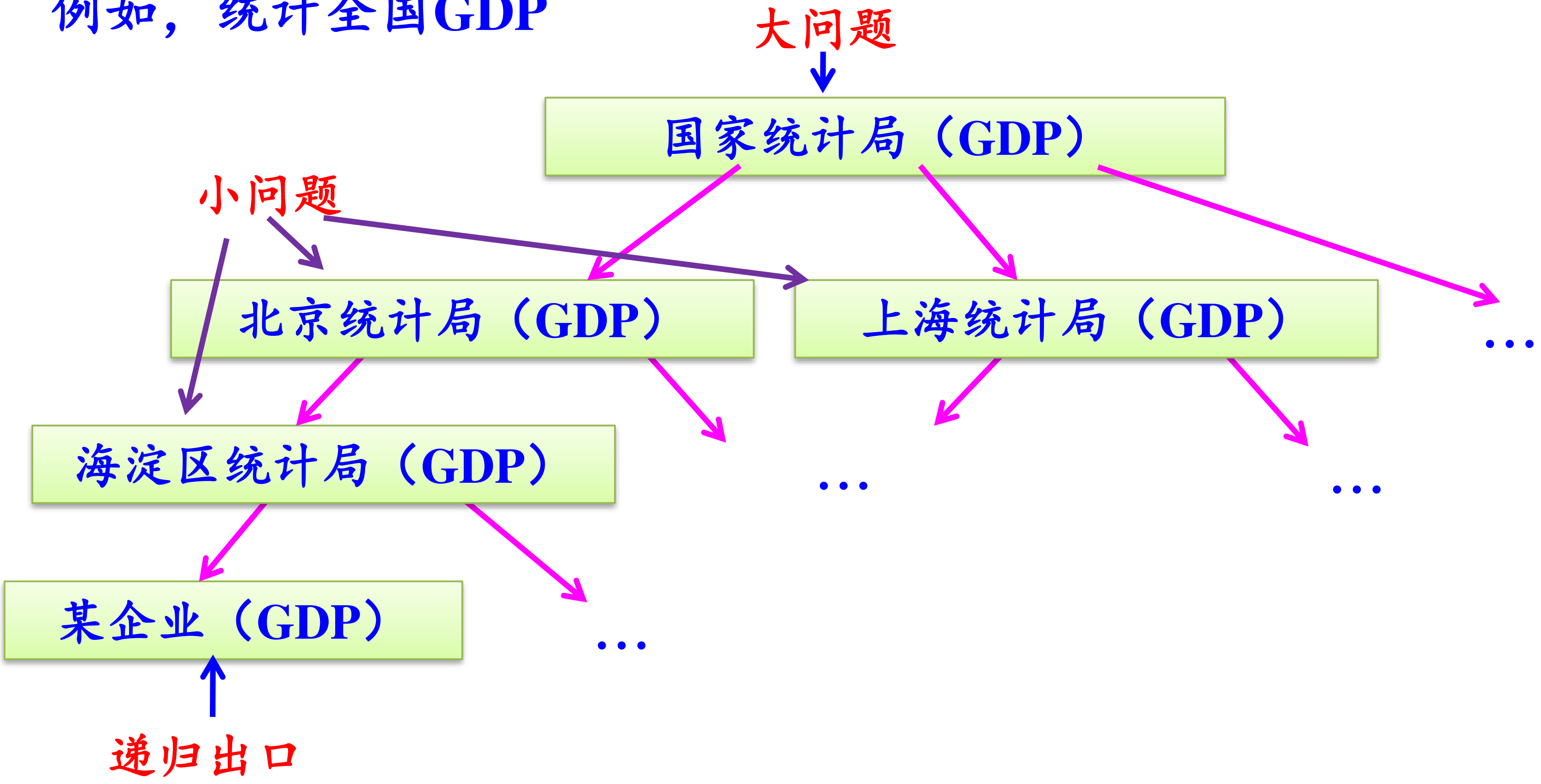
再把这些“小问题”进一步分解成更小的“小问题”来解决。

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：

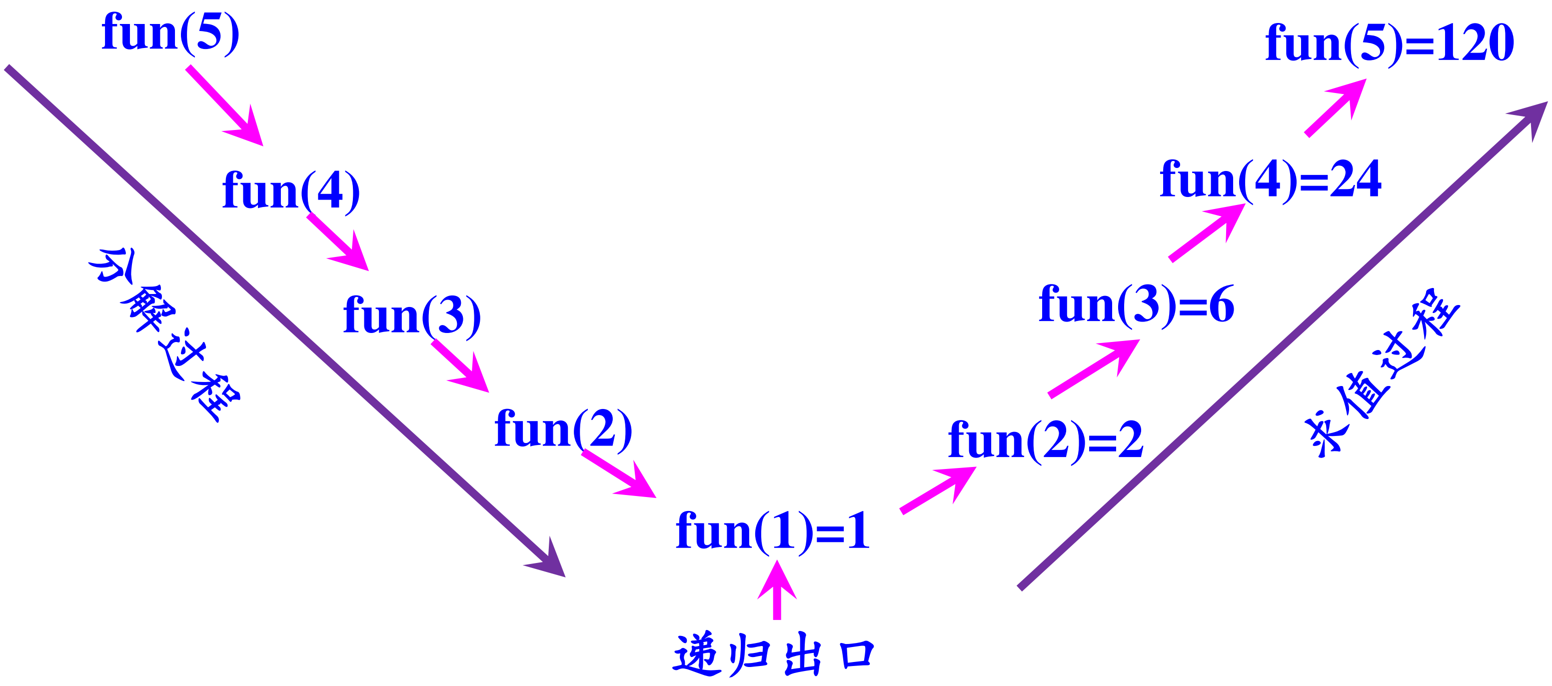
$$f(s_n)$$
$$\downarrow$$
$$f(s_{n-1})$$
$$\downarrow$$
$$\cdots$$
$$\downarrow$$
$$f(s_2)$$
$$\downarrow$$
$$f(s_1)$$

遇到递归出口发生“质变”，即原递归问题便转化成可以直接求解的问题。求值过程：

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

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)即5!的过程如下：

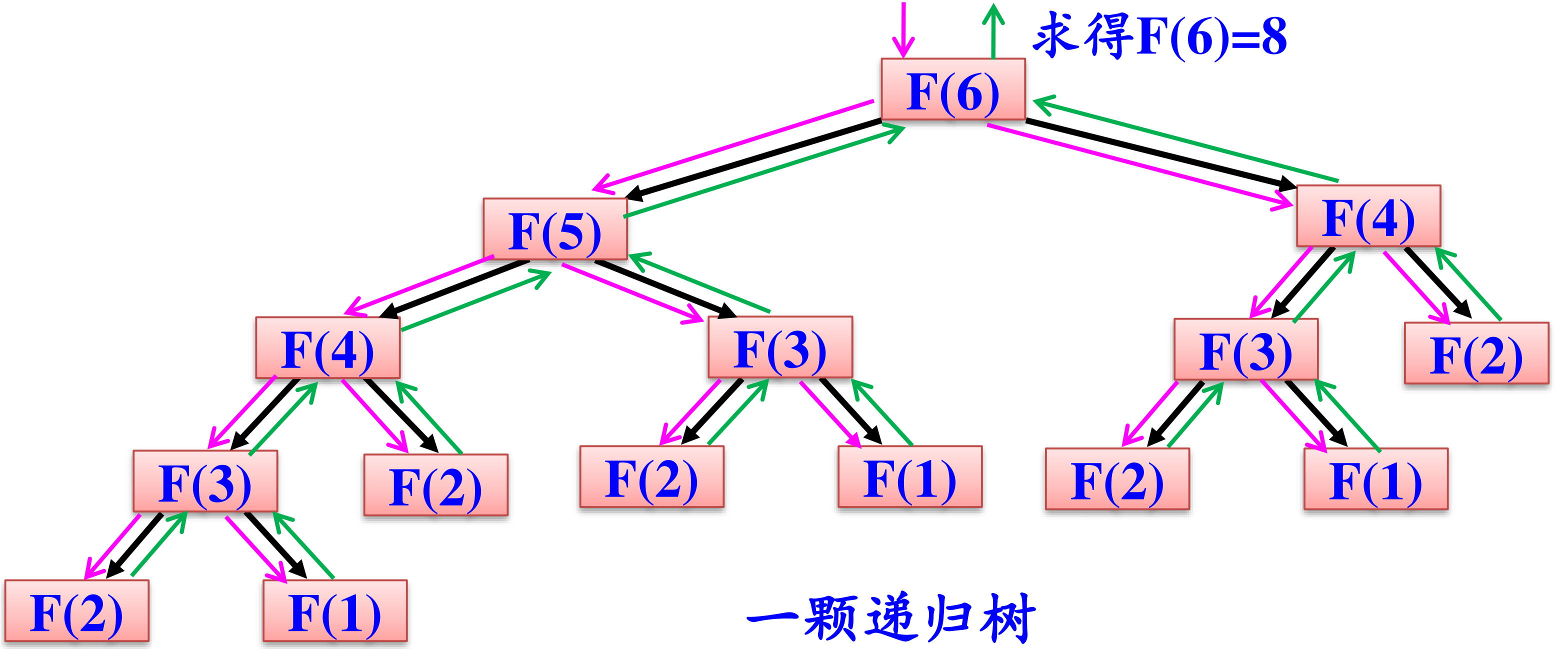


对于复杂的递归问题，在求解时需要进行多次分解和求值。

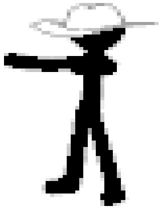
例如：

$$F(1)=1，F(2)=1$$

$$F(n)=F(n-1)+F(n-2) \quad n>2 \qquad 求F(6)=？$$



一颗递归树



——本讲完——

5.2 递归算法的设计

5.2.1 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。

递归模型 → 递归算法

求递归模型的步骤

（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”$f(s')$；

（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**。

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

数学归纳法

⟺ 假设$n=k-1$时等式成立，求证$n=k$时等式成立

⟺ 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。

$$\underbrace{\overbrace{A[0]\quad A[1]\quad \cdots\cdots\quad A[i-1]}^{f(A,i-1)\text{：小问题，处理}i\text{个元素}}\ A[i]}_{f(A,i)\text{：大问题，处理}i+1\text{个元素}}\quad \cdots\cdots\quad A[n-1]$$

假设$f(A,i-1)$已求出，则$f(A,i)=\mathrm{MIN}(f(A,i-1), A[i])$，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：

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

由此得到如下递归求解算法：

```
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

5.2.2 基于递归数据结构的递归算法设计

递归数据结构的数据特别适合递归处理 ⇨递归算法

种瓜得瓜：递归性

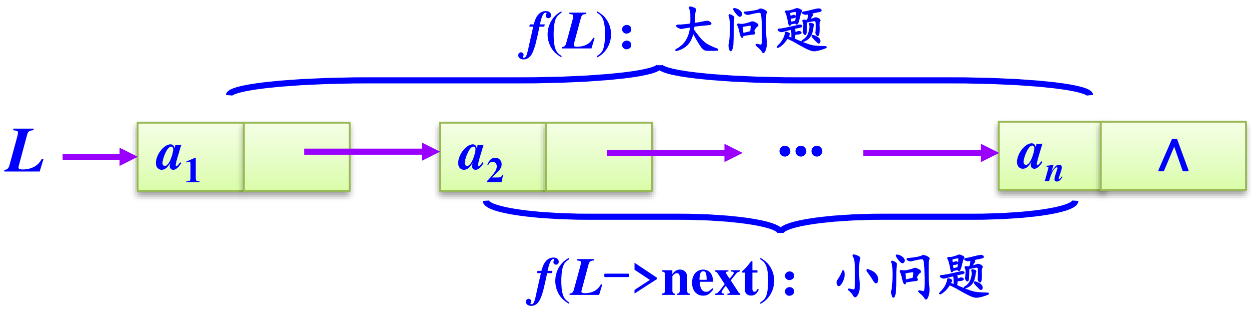
数据：D={瓜的集合}

运算：Op={种瓜}

递归性：

$$\mathrm{Op}(x \in D) \in D$$

【例5-1】设计不带头节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。

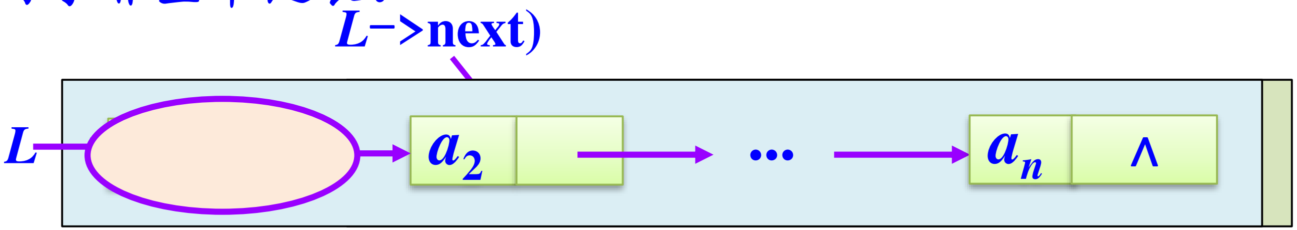
为什么在这里设计单链表的递归算法时不带头节点？

❶ 求单链表中数据节点个数。

设$f(L)$为单链表中数据节点个数。

☑ 空单链表的数据节点个数为0 ⟶ $f(L)=0$　当L=NULL

☑ 对于非空单链表：



$$f(L) \quad = \quad f(L\text{->next}) \quad + \ 1$$

递归模型如下：

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

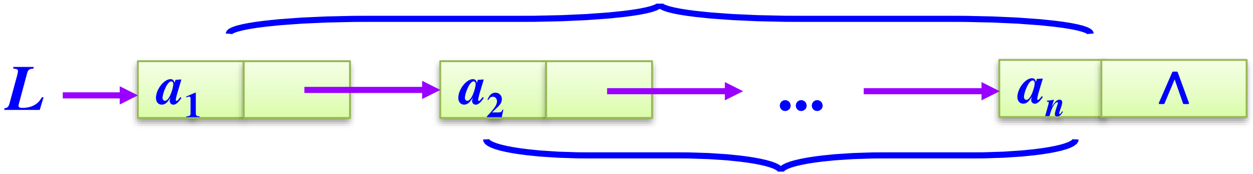
求单链表中数据节点个数递归算法如下：

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

不带头节点单链表L

❷ 正向显示所有节点值。 ❸ 反向显示所有节点值。

$f(L)$：大问题，输出$a_n,\ldots,a_2a_1$



$f(L\text{->}next)$：小问题，输出$a_n,\ldots,a_2$

假设$f(L\text{->}next)$已求解

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

不带头节点单链表L

正向显示所有节点值。

递归模型如下：

$f(L) \equiv$ 不做任何事件　　当L=NULL

$f(L) \equiv$ 输出L->data;$f(L$->next)　　其他情况

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

反向显示所有节点值。

递归模型如下：

$f(L) \equiv$ 不做任何事件　　当L=NULL

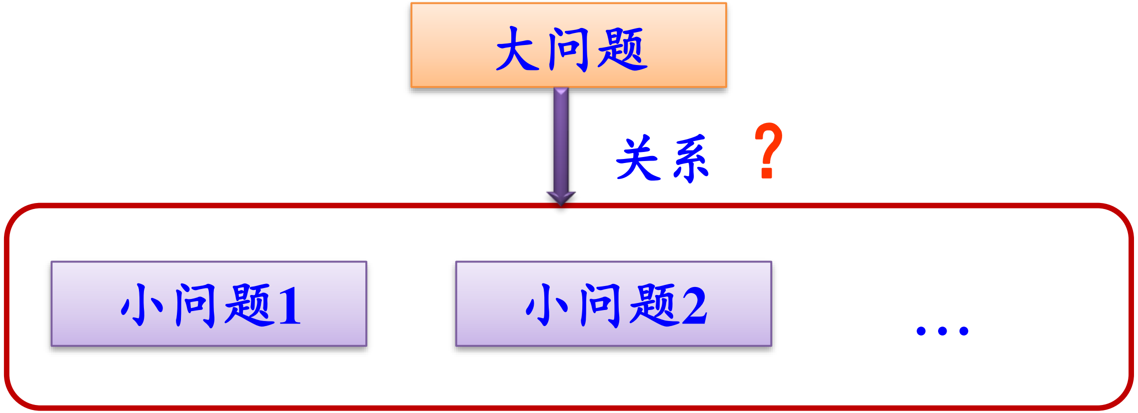
$f(L) \equiv$ $f(L$->next);输出L->data　　其他情况

```
void traverseR(Node *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。

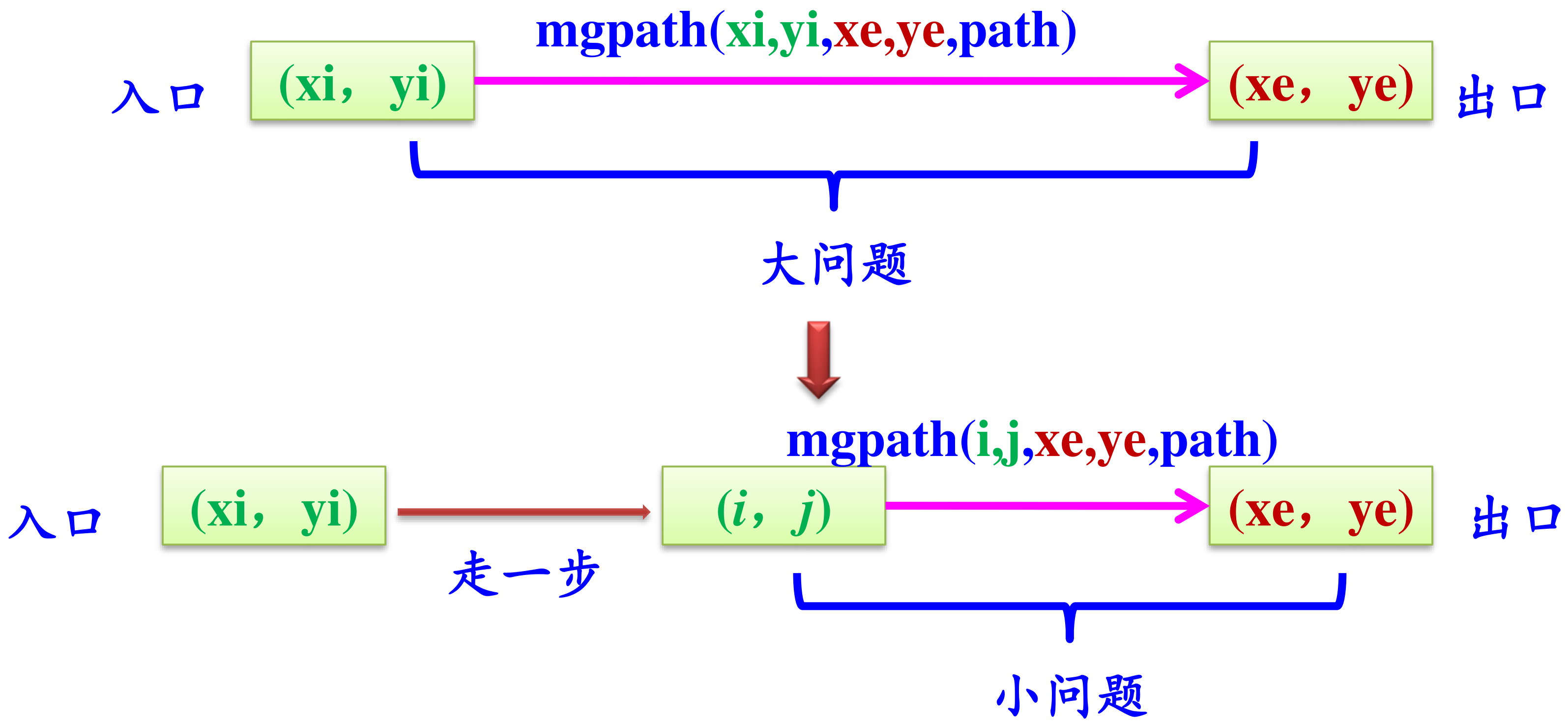


【例5-2】 采用递归算法求解迷宫问题，并输出从入口到出口的所有迷宫路径。

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)
(i，j)
(xe，ye)
出口
走一步
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");
      }
```

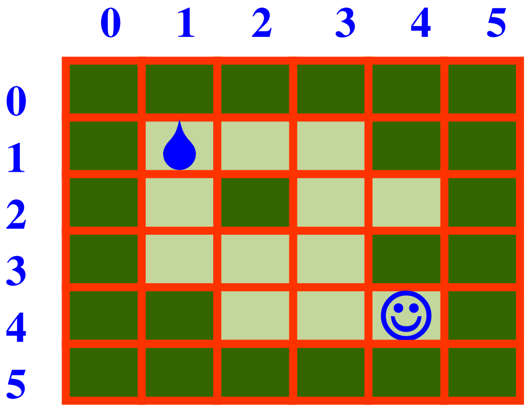
找到了出口，输出一条路径（递归出口）

```
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

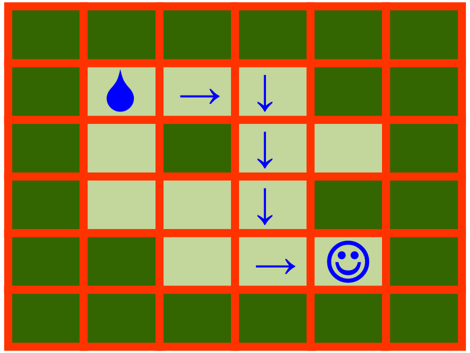



```
int mg[M+2][N+2]=        //M=4,  N=4
{   {1, 1, 1, 1, 1, 1},
    {1, 0, 0, 0, 1, 1},
    {1, 0, 1, 0, 0, 1},
    {1, 0, 0, 0, 1 ,1},
    {1, 1, 0, 0, 0, 1},
    {1, 1, 1, 1, 1, 1}  };
```

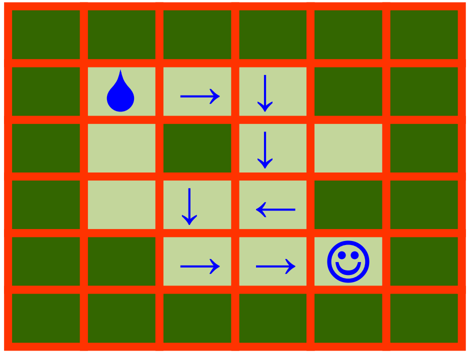
```
void main()
{      PathType path;
       path.length=0;
       mgpath(1,1,4,4,path);
}
```

得到如下4条迷宫路径：

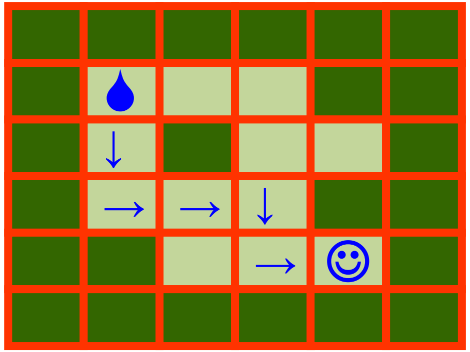
迷宫路径1



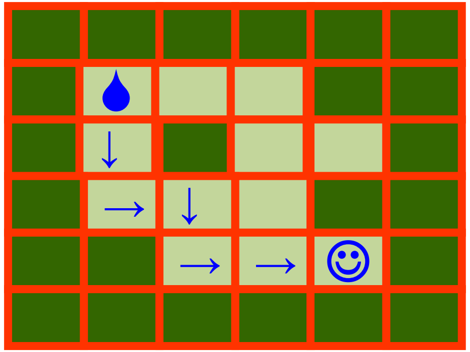
迷宫路径2



迷宫路径3



迷宫路径4



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?

——本章完——

第6周小结

1 递归基础

❶ 一个递归模型由哪两部分构成？

- **递归出口**—确定递归结束情况
- **递归体**—确定大小问题的求解情况

❷ 递归算法如何转换为非递归算法？

- 对于尾递归，可以用循环递推方法来转换。
- 对于其他递归，可以用栈模拟执行过程来转换。

❸ 在Hanoi问题的递归算法中，当移动6个盘片时递归次数是多少？

$m(n) = 1$ 当$n=1$

$m(n) = 2m(n-1)+1$ 当$n>1$

$$
\begin{aligned}
t(6) &= 2t(5) + 1 \\
&= 2^2t(4) + 1 + 2 \\
&= 2^3t(3) + 1 + 2 + 2^2 \\
&= 2^4t(2) + 1 + 2 + 2^2 + 2^3 \\
&= 2^5t(1) + 1 + 2 + 2^2 + 2^3 + 2^4 \\
&= 1 + 2 + 2^2 + 2^3 + 2^4 + 2^5 = 2^6 - 1 = 63
\end{aligned}
$$

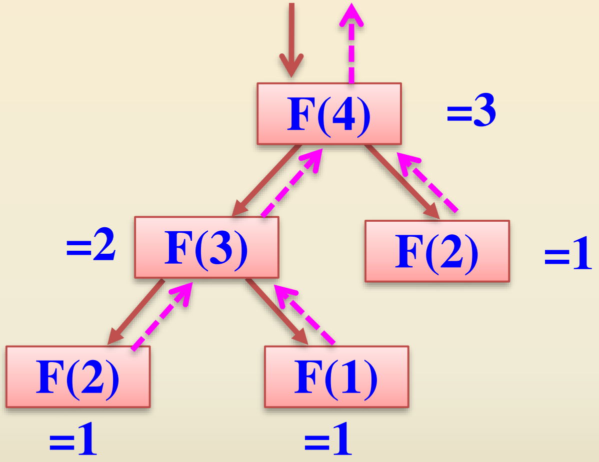
④ 分析递归求Fibonacci数列时，栈的变化情况？

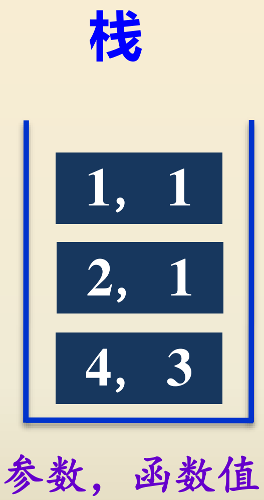
F(1)=1

F(2)=1

F(n)=F(n−1)+F(n−2)　n>2

求F(4) = ？





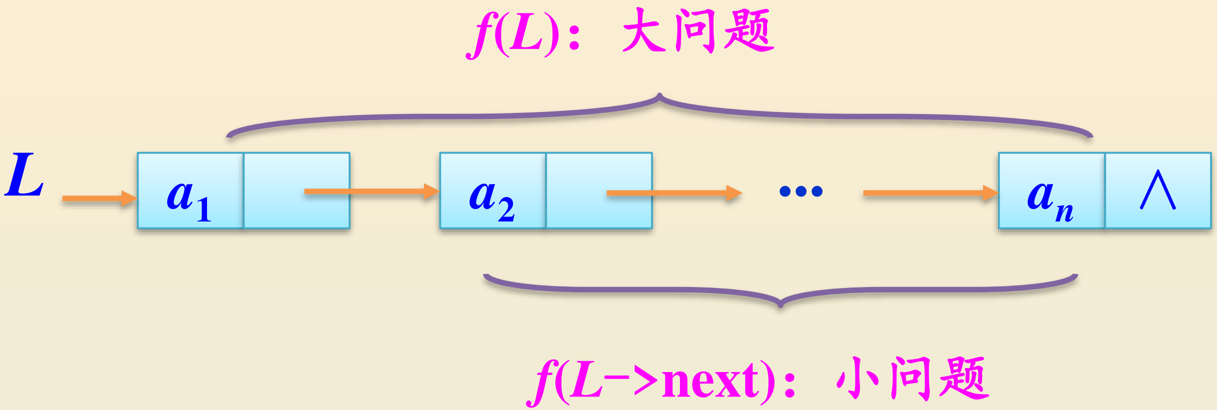
求出F(4)=3

2 递归算法设计

❶ 基于递归数据结构的递归算法设计

- 利用递归数据结构的递归特性建立递归模型
- 编写对应的递归算法

设计递归算法销毁一个不带头节点的单链表。



$f(L)$ ⇔ 不做任何事件	当L为空
$f(L)$ ⇔ $f(L\text{->next})$; free(L);	当L非空

算法如下：

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

❷ 基于递归方法的递归算法设计

如何将递归特性不明显的问题转化为递归问题求解

- 确定问题的形式化描述
- 确定哪些是大问题，哪些是小问题
- 确定大、小问题的关系
- 确定特殊（递归结束）情况

假设a数组含有1，2，…，n，求其全排列。

解： 设$f(a, n, k)$为$a[0..k]$（$k+1$个元素）的所有元素的全排序，为大问题。

则$f(a, n, k-1)$为$a[0..k-1]$（k个元素）的所有元素的全排序，为小问题。

假设$f(a, n, k-1)$可求，对于$a[k]$位置，可以取$a[0..k]$任何元素值，再组合$f(a, n, k\text{-}1)$，则得到$f(a, n, k)$。

$$\underbrace{\overbrace{a[0]\quad a[1]\quad \cdots\quad a[k\text{-}1]}^{f(a,\ n,\ k-1)}\qquad a[k]}_{f(a,\ n,\ k)}$$

此位置可以取$a[0]$～$a[k]$中任何值，但不重复！

采用循环i：0～k，$a[i]$⇔$a[k]$

$f(a, n, k)$ ⇔ 输出a　　　当k=0时(一个元素的全排列)

$f(a, n, k)$ ⇔ $a[k]$位置取$a[0..k]$任何之值，　　　其他情况

并组合$f(a, n, k\text{-}1)$的结果;

```c
void perm(int a[], int n, int k)
{     int i, j;
      if (k==0)
      {     for (j=0;j<n;j++)
                  printf("%d", a[j]);
            printf("\n");
      }
      else
      {     for (i=0;i<=k;i++)
            {     swap(a[k], a[i]);
                  perm(a, n, k-1);
                  swap(a[k], a[i]);
            }
      }
}
```

```c
void main()
{     int n=3, k=2;
      int a[]={1,2,3};
      perm(a, n, k);
}
```

输出结果：

231

321

312

132

213

123

3 递归函数设计中几个问题

❶ 递归函数中的引用形参可以用全局变量代替

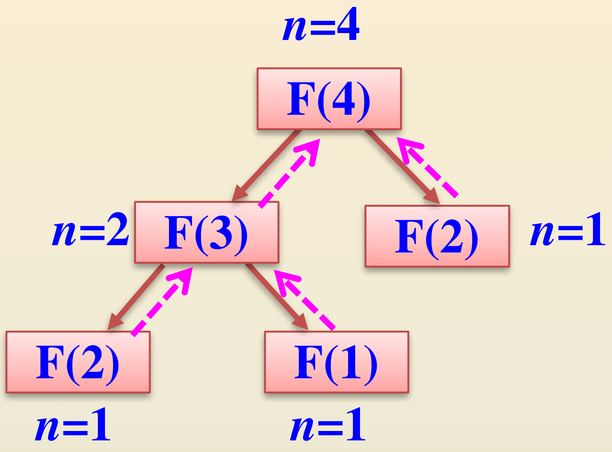
例如，求1+2+…+n

```
void add(int n， int &s)   //s=1+2+…+n
{    int s1;
     if (n==1)
          s=1;
     else
     {     add(n-1， s1);
           s=s1+n;
     }
}
```

可以用全局变量代替：

```
int s=0;                 //全局变量
void add1(int n)    //理解为:s与add(n)绑定， s=1+2+…+n
{     if (n==1)
            s=1;
      else
      {     add1(n-1);
            s+=n;
      }
}
```

❷ 递归函数中的非引用形参作为状态变量，可以自动回溯



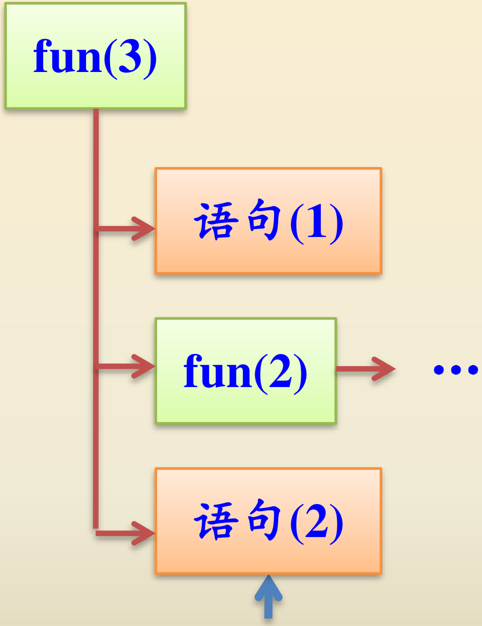
- *n*表示状态
- 状态自动回溯

❸ 递归调用后面的语句表示该子问题执行完毕后要完成的功能

```c
void fun(int n)
{
      if (n>=1)
      {
            printf("n1=%d\n",n);  //(1)
            fun(n-1);
            printf("n2=%d\n",n);  //(2)
      }
}
```

fun(3)的输出结果

```
n1=3
n1=2
n1=1
n2=1
n2=2
n2=3
```



在fun(2)的全部功能执行后才执行

掌握递归函数的执行过程有助于递归算法设计