

3.3 栈与递归的实现

3.3.1 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**。

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】求 $n!$ (n 为正整数) 的递归函数。

```
int fun(int n)
{   if (n==0)                //语句1
    return 1;                //语句2
    else                      //语句3
    return n*fun(n-1);        //语句4
}
```

在该函数 $\text{fun}(n)$ 求解过程中，直接调用 $\text{fun}(n-1)$ （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。

3.3.2 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。

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 Node
{
    ElemType data;
    struct Node *next;
} LNode, *LinkList;
```

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。

□ 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

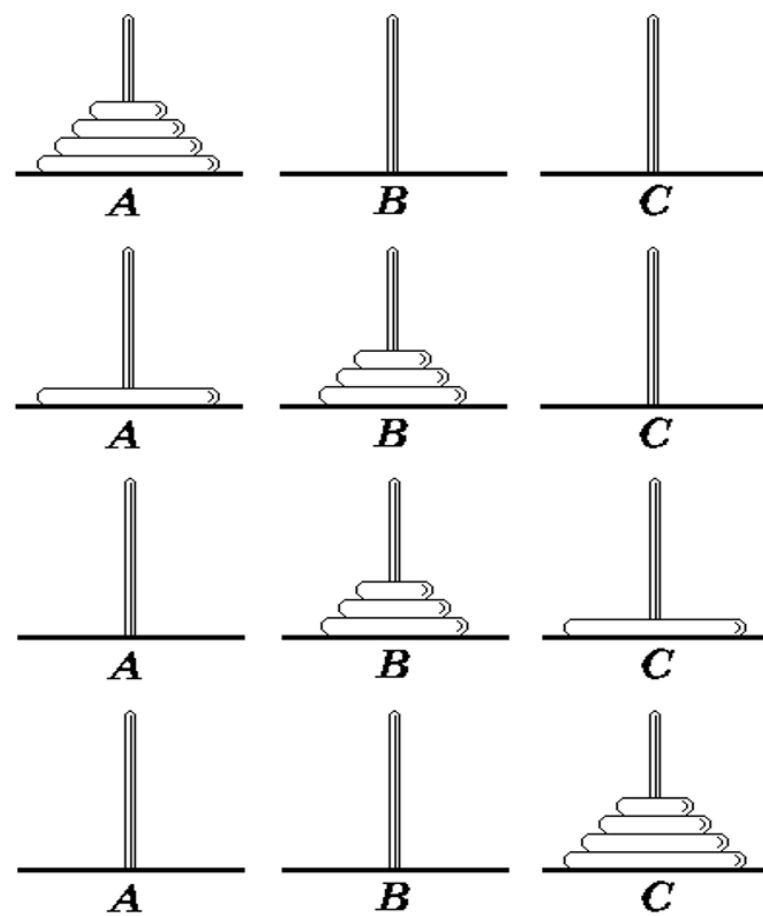
□ 有些问题的解法是递归的

【Hanoi问题】 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的盘片，现要求将A塔座上的n个盘片移到塔座C上并仍按同样顺序叠放。

✓ 盘片移动时必须遵守以下规则：

- (1) 每次只能移动一个盘片；
- (2) 盘片可以插在A、B和C中任一塔座；
- (3) 任何时候都不能将一个较大的盘片放在较小的盘片上。

✓ 设计递归求解算法。



4阶Hanoi塔问题的状态

- 汉诺塔 (Tower of Hanoi) 问题的解法:
 - 如果 $n = 1$, 则将这一个盘子直接从 A 柱移到 C 柱上。否则, 执行以下三步:
 - ① 用 C 柱做过渡, 将 A 柱上的 $(n-1)$ 个盘子移到 B 柱上;
 - ② 将 A 柱上最后一个盘子直接移到 C 柱上;
 - ③ 用 A 柱做过渡, 将 B 柱上的 $(n-1)$ 个盘子移到 C 柱上。
- 这是典型的递归法问题。

【Hanoi问题】递归算法的实现

```
void Hanoi ( int n, char A, char B, char C )
{
    // 用A、B、C代表三个柱子，算法模拟汉诺塔问题
    if (n == 1)
        printf ( " move %s", A, " to  %s ", C );
    else {
        Hanoi ( n-1, A, C, B );
        printf ( " move %s", A, " to  %s ", C );
        Hanoi ( n-1, B, A, C );
    }
}
```

3.3.3 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：

$$\text{fun}(0)=1 \quad (1)$$

$$\text{fun}(n)=n*\text{fun}(n-1) \quad n>0 \quad (2)$$

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 $\text{fun}(n)$ 的值与 $\text{fun}(n-1)$ 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。

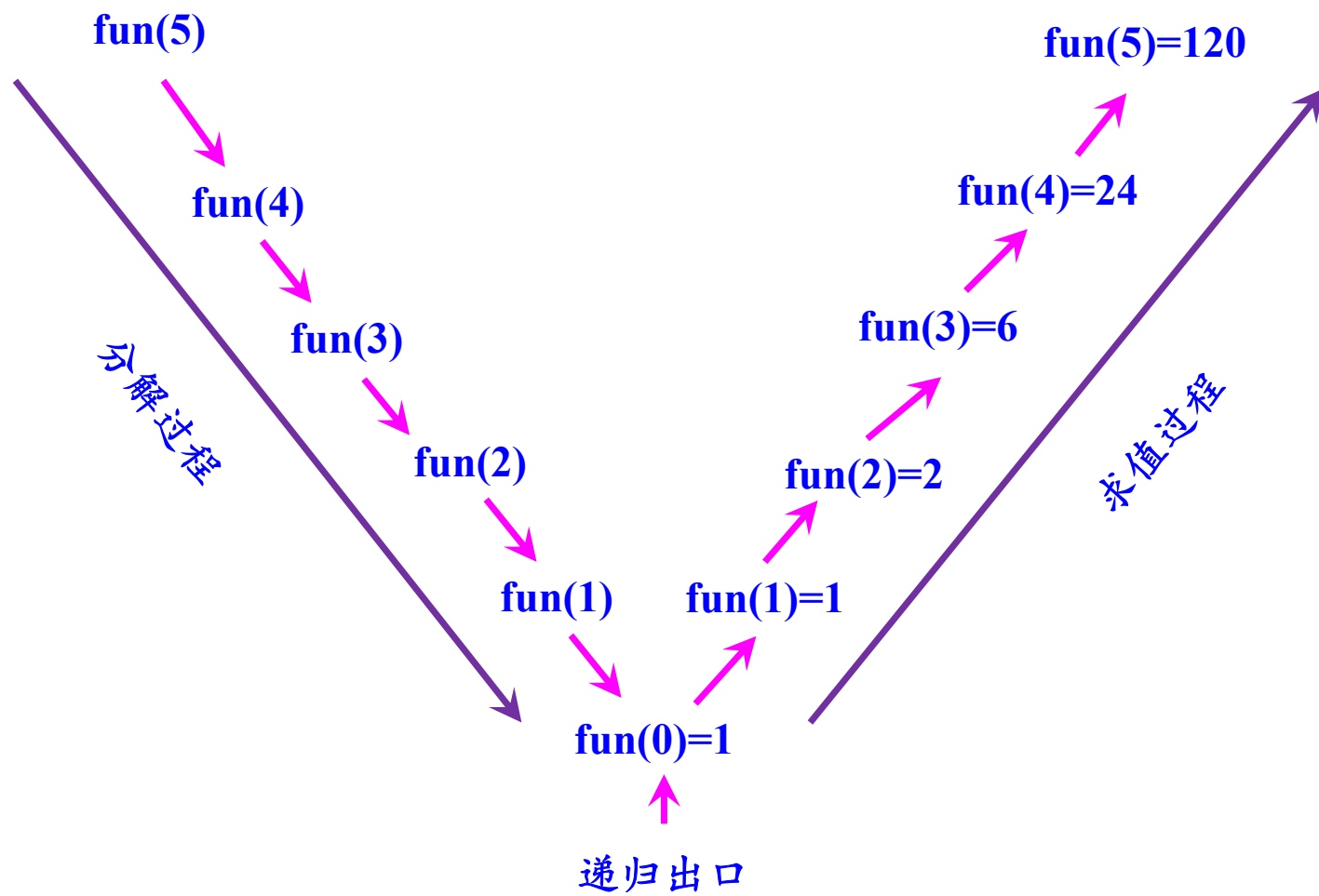
递归思路是：

- 把一个不能或不好直接求解的“大问题”转化为一个或几个“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。

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。

以求解fun(5)的过程为例说明。

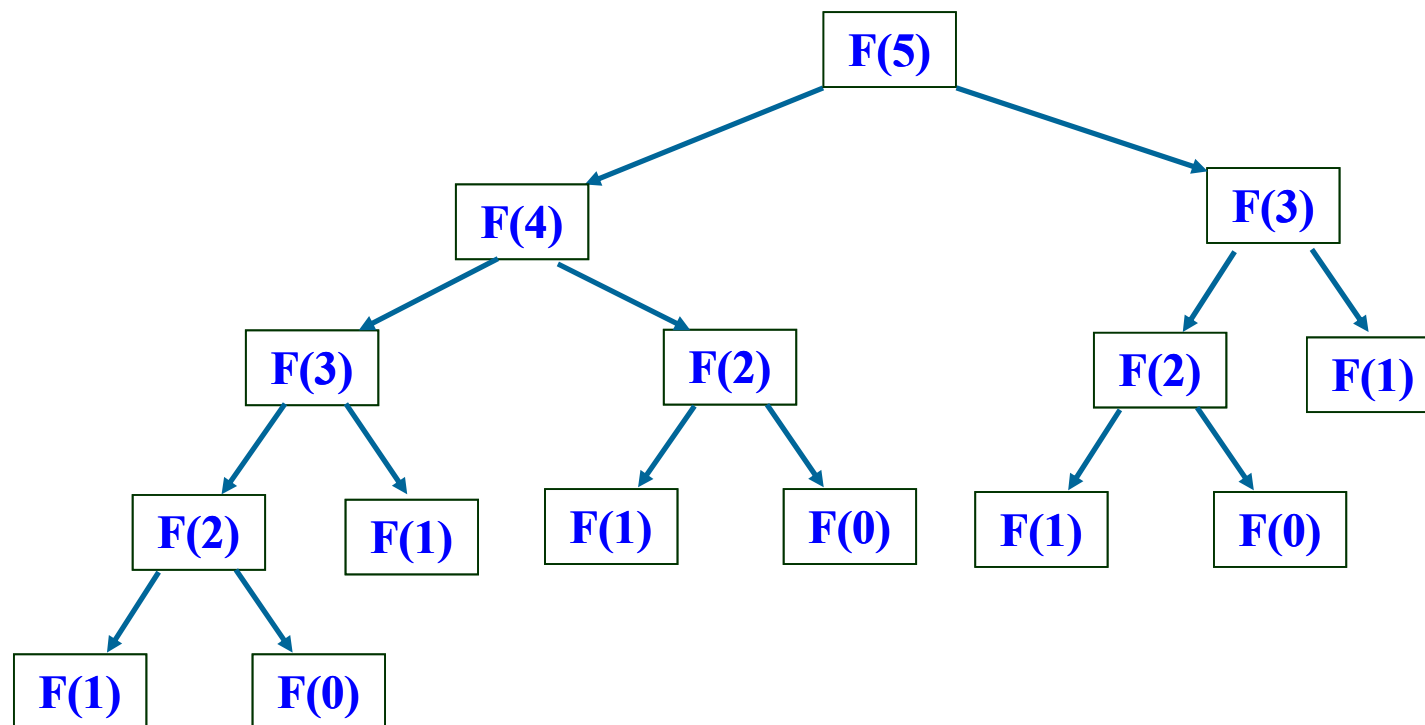
求解 $\text{fun}(5)$ 即 $5!$ 的过程如下：



$F(0)=1$

$F(1)=1$

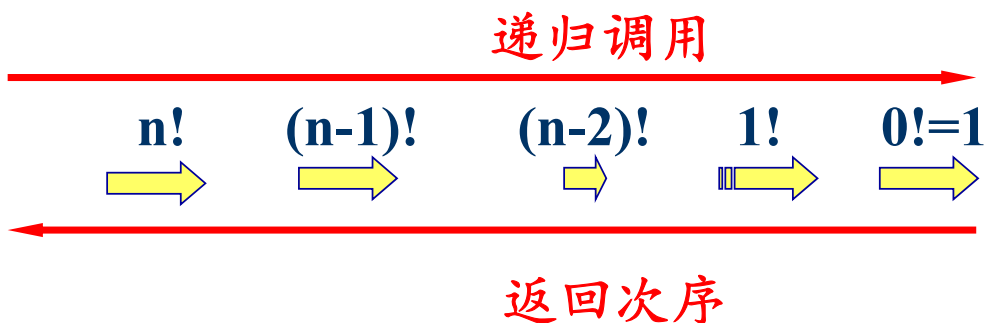
$F(n)=F(n-1)+F(n-2) \quad n \geq 2$



递归树

递归过程与递归工作栈

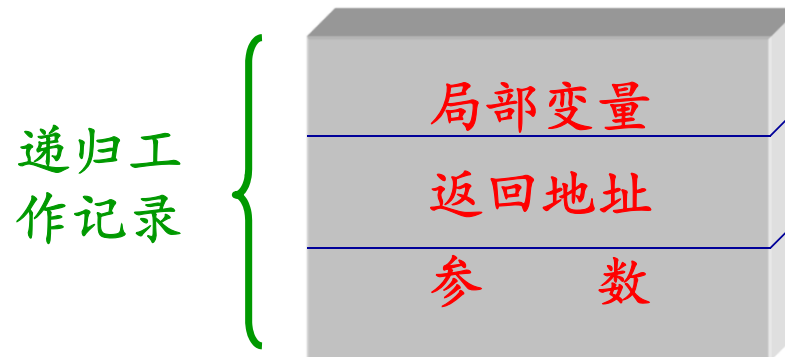
- 递归过程在实现时，需要自己调用自己。
- 层层向下递归，退出时的次序正好相反：



- 主程序第一次调用递归函数为外部调用；递归函数每次递归调用自己为内部调用。它们返回调用它的函数的地址不同。
- 每次调用必须记下返回上层什么地方地址。

递归工作栈

- 每一次递归调用，需要为函数中使用的参数、局部变量等另外分配存储空间，每个函数的工作空间互不干扰，回到上层还可恢复上层原来的值。
- 每层递归调用需分配的空间形成递归的工作记录，按后进先出的栈组织。




```
void main()  
{ printf(“%d\n”, fun(5)); }
```



fun(5)调用：进栈

5	fun(4)*5
---	----------

n 函数值



fun(4)调用：进栈


4	fun(3)*4
5	fun(4)*5



fun(3)调用：进栈


3	fun(2)*3
4	fun(3)*4
5	fun(4)*5

fun(2)调用：进栈




2	fun(1)*2
3	fun(2)*3
4	fun(3)*4
5	fun(4)*5

fun(1)调用：进栈




1	Fun(0)*1
2	fun(1)*2
3	fun(2)*3
4	fun(3)*4
5	fun(4)*5

退栈1次并求fun(1)值




1	1*1
2	fun(1)*2
3	fun(2)*3
4	fun(3)*4
5	fun(4)*5

退栈1次并求fun(2)值




2	$1 * 2 = 2$
3	$\text{fun}(2) * 3$
4	$\text{fun}(3) * 4$
5	$\text{fun}(4) * 5$

退栈1次并求fun(3)值




3	$2 * 3 = 6$
4	$\text{fun}(3) * 4$
5	$\text{fun}(4) * 5$

退栈1次并求fun(4)值



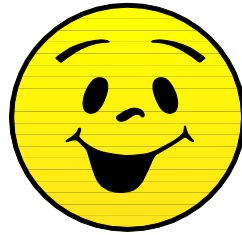
4	$6 * 4 = 24$
5	$\text{fun}(4) * 5$

退栈1次并求fun(5)值



5	$24 * 5 = 120$
---	----------------

退栈1次并输出120



— END —