

算法设计与分析

Algorithm Design and Analysis

赵宏

hongzhao@xidian.edu.cn

2023. 3. 8

西安电子科技大学

广州研究院

第2章 递归算法设计技术

2.1 什么是递归

2.2 递归算法设计

2.3 递归算法设计示例

2.4* 递归算法转化非递归算法

2.5 递推式的计算

2.1 什么是递归

2.1.1 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**。

任何间接递归都可以等价地转换为直接递归。

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2.1】设计求 $n!$ (n 为正整数) 的递归算法。

解：对应的递归函数如下：

```
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 $\text{fun}(n)$ 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。

一般来说，能够用递归解决的问题应该满足以下三个条件：

- 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。
- 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。

2.1.2 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。

2. 数据结构是递归的

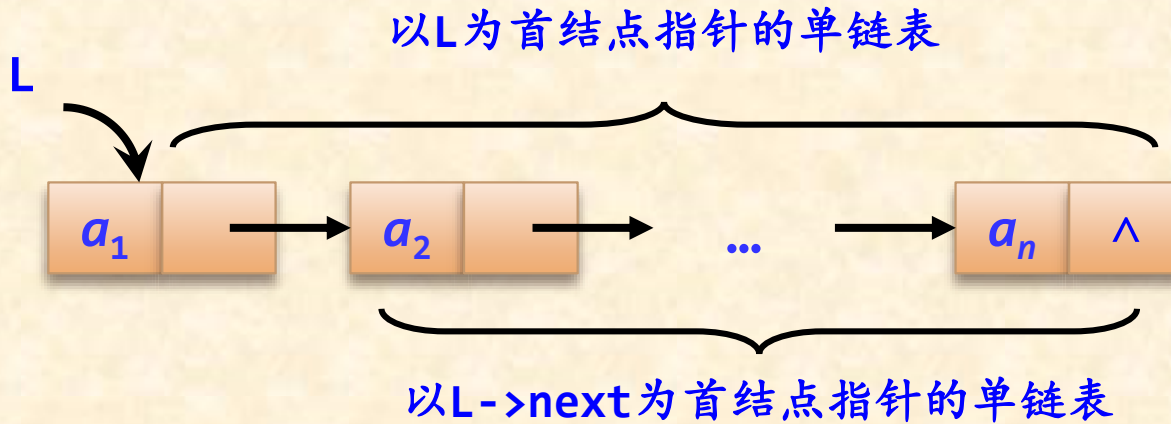
有些数据结构是递归的。例如单链表就是一种递归数据结构，其结点类型声明如下：

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

为什么可以这样
递归定义类型

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。

不带头结点单链表示意图



体现出数据结构的递归性。

如果带有头结点又会怎样呢???

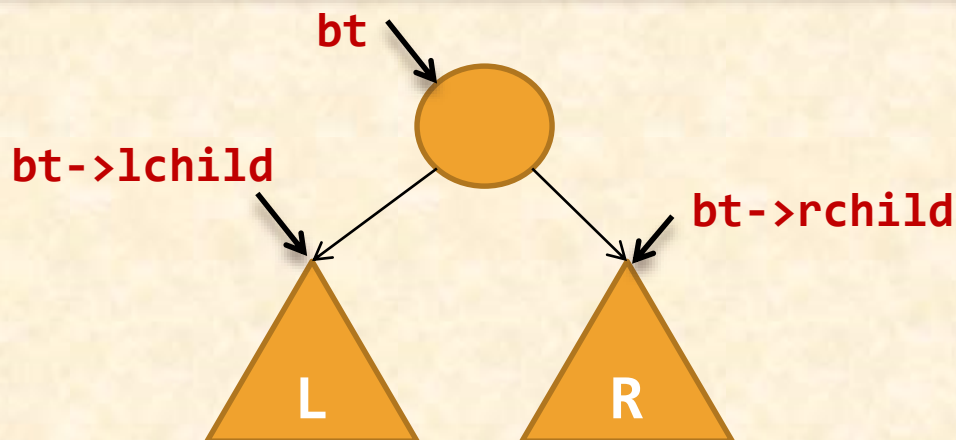
对于递归数据结构，采用递归的方法编写算法既方便又有效。例如，求一个不带头结点的单链表L的所有data域（假设为int型）之和的递归算法如下：

```
int Sum(LinkList *L)
{   if (L==NULL)
        return 0;
    else
        return(L->data+Sum(L->next));
}
```

【例2.2】分析二叉树的二叉链存储结构的递归性，设计求非空二叉链bt中所有结点值之和的递归算法，假设二叉链的data域为int型。

解：二叉树采用二叉链存储结构，其结点类型定义如下：

```
typedef struct BNode
{
    int data;
    struct BNode *lchild, *rchild;
} BNode;           //二叉链结点类型
```



```
int Sumbt(BNode *bt)           //求二叉树bt中所有结点值之和
{
    if (bt->lchild==NULL && bt->rchild==NULL)
        return bt->data;       //只有一个结点时返回该结点值
    else
        return Sumbt(bt->lchild)+ Sumbt(bt->rchild)+bt->data); //否则返回左、右子树结点值之和加上根结点值
}
```

3. 问题的求解方法是递归的

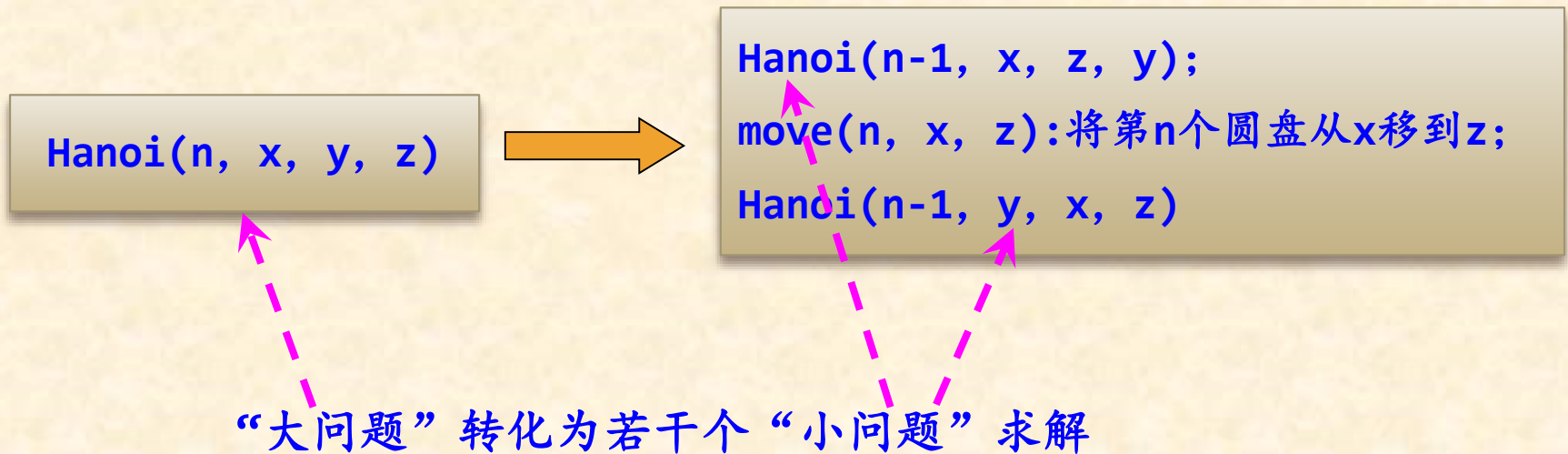
有些问题的解法是递归的，典型的有Hanoi问题求解。



盘片移动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中任一塔座；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，并将其转换为非递归算法。

设 $\text{Hanoi}(n, x, y, z)$ 表示将 n 个盘片从 x 通过 y 移动到 z 上，递归分解的过程是：



2.1.3 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：

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

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

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。

递归出口的一般格式如下：

$$f(s_1)=m_1 \quad (2.1)$$

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：

$$f(s_{n+1})=g(f(s_i), f(s_{i+1}), \dots, f(s_n), c_j, c_{j+1}, \dots, c_m) \quad (2.2)$$

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} 、 \dots 、 s_n 为递归“小问题”， c_j 、 c_{j+1} 、 \dots 、 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。

2.1.4 递归算法的执行过程

- 一个正确的递归程序虽然每次调用的是相同的子程序，但它的参量、输入数据等均有变化。
- 在正常的情况下，随着调用的不断深入，必定会出现调用到某一层的函数时，不再执行递归调用而终止函数的执行，遇到递归出口便是这种情况。

- 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。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。

- 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。
- 这些单元以系统栈的形式存放，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。

对于例2.1的递归算法，求5!即执行fun(5)时内部栈的变化及求解过程如下：

```
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：进栈并求值



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	fun(2)*3
4	fun(3)*4
5	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	$24*5=120$
---	------------

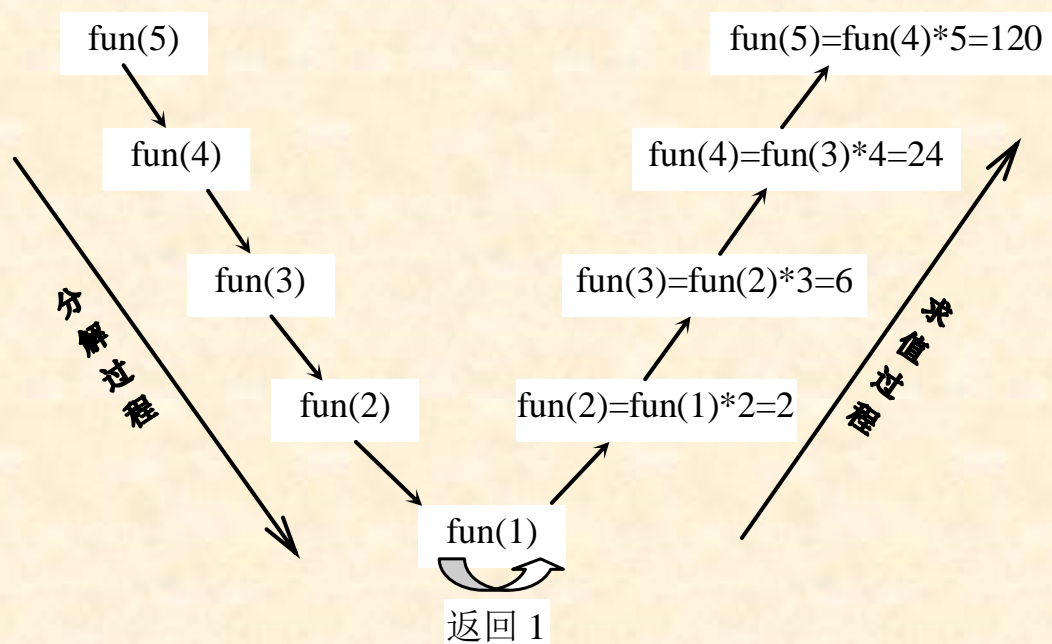


退栈1次并输出120

从以上过程可以得出：

- 每递归调用一次，就需进栈一次，最多的进栈元素个数称为递归深度，当 n 越大，递归深度越深，开辟的栈空间也越大。
- 每当遇到递归出口或完成本次执行时，需退栈一次，并恢复参量值，当全部执行完毕时，栈应为空。

归纳起来，递归调用的实现是分两步进行的，第一步是分解过程，即用递归体将“大问题”分解成“小问题”，直到递归出口为止，然后进行第二步的求值过程，即已知“小问题”，计算“大问题”。前面的 $\text{fun}(5)$ 求解过程如下所示。



【例2.3】 Fibonacci数列定义为：

$\text{Fib}(n)=1$ $n=1$

$\text{Fib}(n)=1$ $n=2$

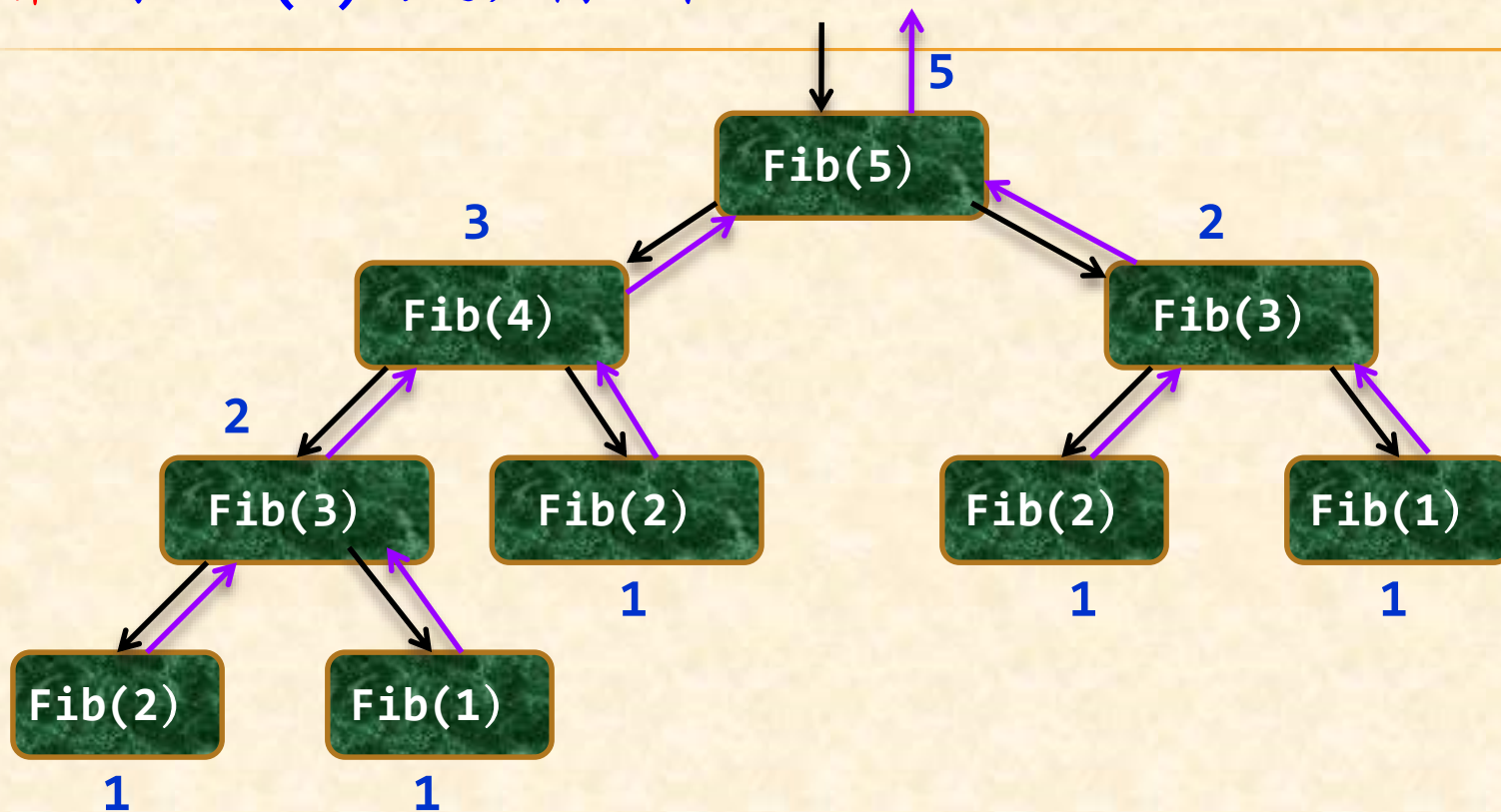
$\text{Fib}(n)=\text{Fib}(n-1)+\text{Fib}(n-2)$ $n>2$

对应的递归算法如下：

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

画出求Fib(5)的递归树以及递归工作栈的变化和求解过程。

解：求Fib(5)的递归树如下：



从上面求Fib(5)的过程看到，对于复杂的递归调用，分解和求值可能交替进行、循环反复，直到求出最终值。

执行Fib(5)时递归工作栈的变化和求解过程:



- 在递归函数执行时，形参会随着递归调用发生变化，但每次调用后会恢复为调用前的形参，将递归函数的非引用型形参的取值称为状态。
- 递归函数的引用型形参在执行后会回传给实参，有时类似全局变量，不作为状态的一部分，在调用过程中状态会发生变化，而一次调用后会自动恢复为调用前的状态。

2.2 递归算法设计

2.2.1 递归与数学归纳法

第一数学归纳法原理：若 $\{P(1), P(2), P(3), P(4), \dots\}$ 是命题序列且满足以下两个性质，则所有命题均为真：

(1) $P(1)$ 为真。

(2) 任何命题均可以从它的前一个命题推导得出。

例如，采用第一数学归纳法证明下式：

$$1+2+\dots+n = \frac{n(n+1)}{2}$$

证明：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 $1+2+\dots+(k-1) = \frac{k(k-1)}{2}$

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

等式成立。即证。

第二数学归纳法原理： 若 $\{P(1), P(2), P(3), P(4), \dots\}$ 是满足以下两个性质的命题序列，则对于其他自然数，该命题序列均为真：

(1) $P(1)$ 为真。

(2) 任何命题均可以从它的前面所有命题推导得出。

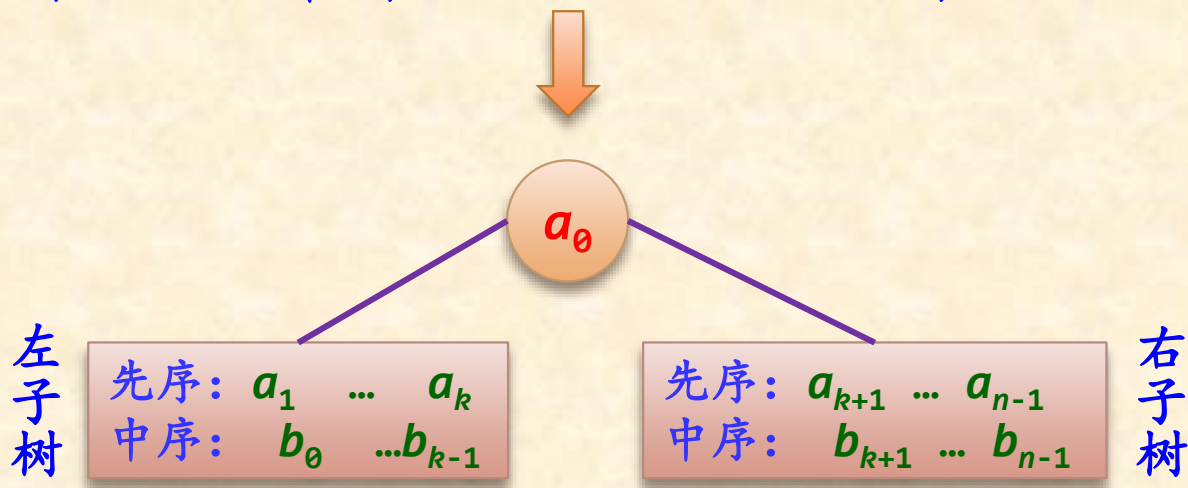
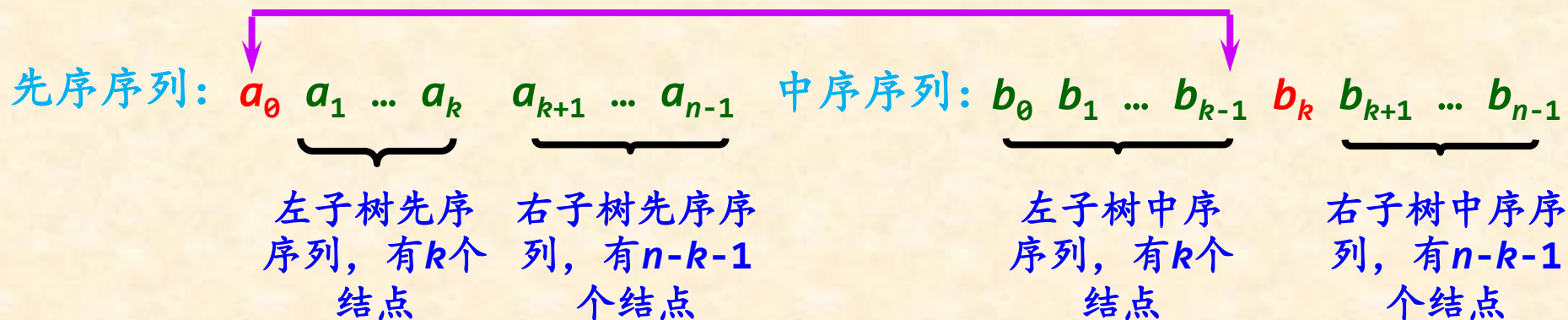
归纳步骤（条件2）的意思是 $P(n)$ 可以从前面所有命题假设 $\{P(1), P(2), P(3), \dots, P(n-1)\}$ 推导得出。

例如，采用第二数学归纳法证明，任何含有 n ($n \geq 0$) 个不同结点的二叉树，都可由它的中序序列和先序序列唯一地确定。

证明：当 $n=0$ 时，二叉树为空，结论正确。

假设结点数小于 n 的任何二叉树（所有结点值不相同），都可以由其先序序列和中序序列唯一地确定。

通过根结点 a_0 在中序序列中找到 b_k



根据归纳假设，由于子先序序列 $a_1 \dots a_k$ 和子中序序列 $b_0 b_1 \dots b_{k-1}$ 可以唯一地确定根结点 a_0 的左子树，而子先序序列 $a_{k+1} \dots a_{n-1}$ 和子中序序列 $b_{k+1} \dots b_{n-1}$ 可以唯一地确定根结点 a_0 的右子树。

综上所述，这棵二叉树的根结点已经确定，而且其左、右子树都唯一地确定了，所以整个二叉树也就唯一地确定了。

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，数学归纳法是递归的基础。

2.2.2 递归算法设计的一般步骤

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。

获取递归模型的步骤如下：

(1) 对原问题 $f(s_n)$ 进行分析，抽象出合理的“小问题” $f(s_{n-1})$ （与数学归纳法中假设 $n=k-1$ 时等式成立相似）；

(2) 假设 $f(s_{n-1})$ 是可解的，在此基础上确定 $f(s_n)$ 的解，即给出 $f(s_n)$ 与 $f(s_{n-1})$ 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 $n=1$ 或 $n=0$ 时等式成立相似）。

【例2.5】 用递归法求一个整数数组 a 的最大元素。

解： 设 $f(a, i)$ 求解数组 a 中前 i 个元素即 $a[0..i-1]$ 中的最大元素，则 $f(a, i-1)$ 求解数组 a 中前 $i-1$ 个元素即 $a[0..i-2]$ 中的最大元素，前者为“大问题”，后者为“小问题”。

假设 $f(a, i-1)$ 已求出，则有 $f(a, i) = \text{MAX}\{f(a, i-1), a[i-1]\}$ 。递推方向是朝 a 中元素减少的方向推进，当 a 中只有一个元素时，该元素就是最大元素，所以 $f(a, 1) = a[0]$ 。

由此得到递归模型如下：

$f(a, i) = a[0]$

当 $i=1$ 时

$f(a, i) = \text{MAX}\{f(a, i-1), a[i-1]\}$

当 $i > 1$ 时

对应的递归算法如下：

```
int fmax(int a[], int i)
{
    if (i==1)
        return a[0];
    else
        return(fmax(a, i-1), a[i-1]);
}
```

2.2.3 递归数据结构及其递归算法设计

1. 递归数据结构的定义

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。

归纳起来，递归数据结构定义为：

$$RD=(D, Op)$$

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 是基本递归运算的集合， $Op=\{op_j\}$ ($1\leq j\leq m$ ，共 m 个基本递归运算)，对于 $\forall d_i\in D$ ，不妨设 op_j 为一元运算符，则有 $op_j(d_i)\in D$ ，也就是说，递归运算符具有封闭性。

二叉树的定义中， D 是给定二叉树及其子树的集合（对于一棵给定的二叉树，其子树的个数是有限的）， $Op=\{op_1, op_2\}$ 由基本递归运算符构成，它们的定义如下：

$op1(p) = p \rightarrow lchild$

$op2(p) = p \rightarrow rchild$

其中， p 指向二叉树中的一个非空结点。

2. 基于递归数据结构的递归算法设计

1) 单链表的递归算法设计

在设计不带头结点的单链表的递归算法时：

设求解以L为首结点指针的整个单链表的某功能为“大问题”。

而求解除首结点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）的相同功能为“小问题”。

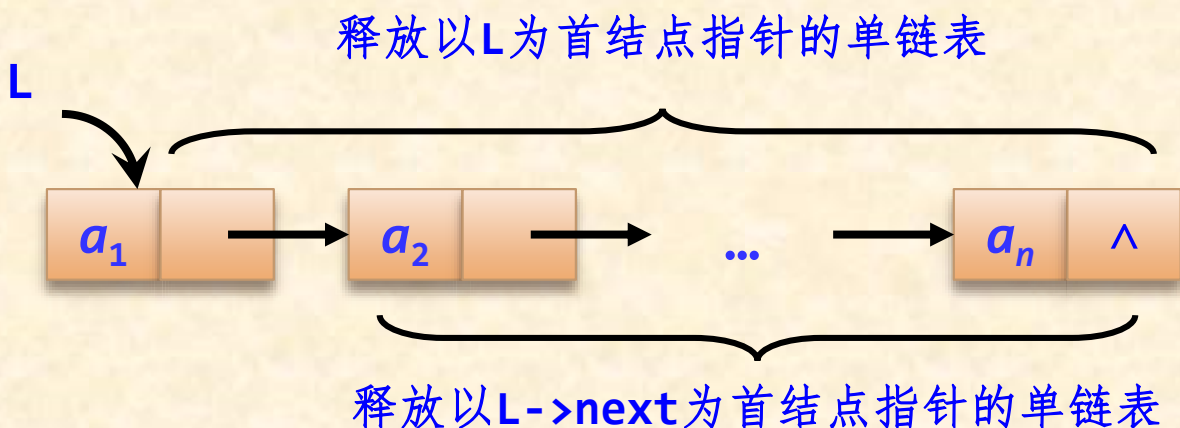
由大小问题之间的解关系得到递归体。

再考虑特殊情况，通常是单链表为空或者只有一个结点时，这时很容易求解，从而得到递归出口。

【例2.6】 有一个不带头结点的单链表L，设计一个算法释放其中所有结点。

解： 设 $L=\{a_1, a_2, \dots, a_n\}$ ， $f(L)$ 的功能是释放 $a_1 \sim a_n$ 的所有结点，则 $f(L \rightarrow next)$ 的功能是释放 $a_2 \sim a_n$ 的所有结点，前者是“大问题”，后者是“小问题”。

假设 $f(L \rightarrow next)$ 是已实现，则 $f(L)$ 就可以采用先调用 $f(L \rightarrow next)$ ，然后释放L所指结点来求解。



对应的递归模型如下：

$f(L) \equiv$ 不做任何事件

$f(L) \equiv f(L \rightarrow \text{next});$ 释放L结点

当 $L = \text{NULL}$ 时

其他情况



```
void DestroyList(LinkNode *&L)
//释放单链表L中所有结点
{  if (L!=NULL)
    {  DestroyList(L->next);
        free(L);
    }
}
```


2) 二叉树的递归算法设计

二叉树是一种典型的递归数据结构，当一棵二叉树采用二叉链**b**存储时：

设求解以**b**为根结点的整个二叉树的某功能为“大问题”。

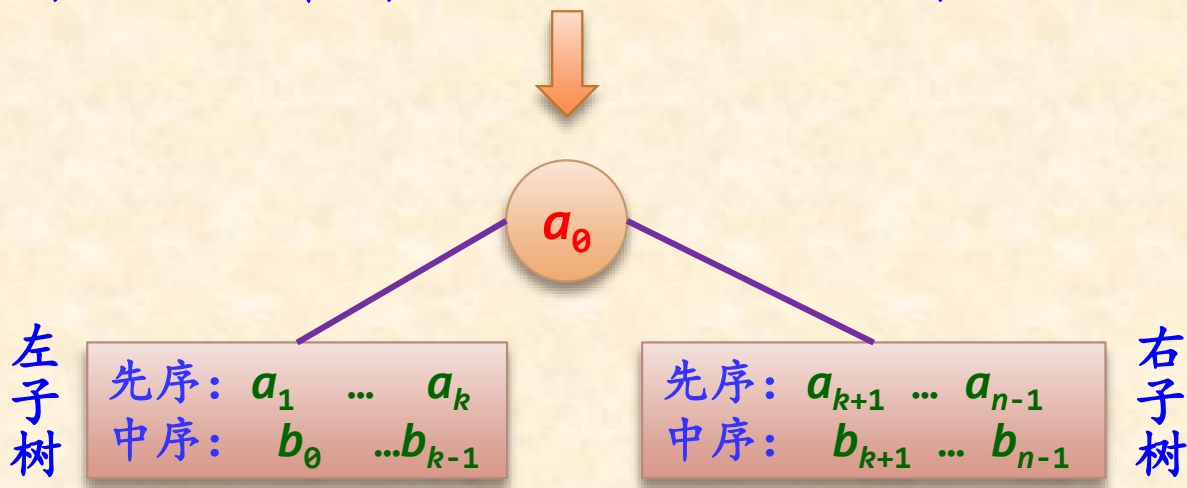
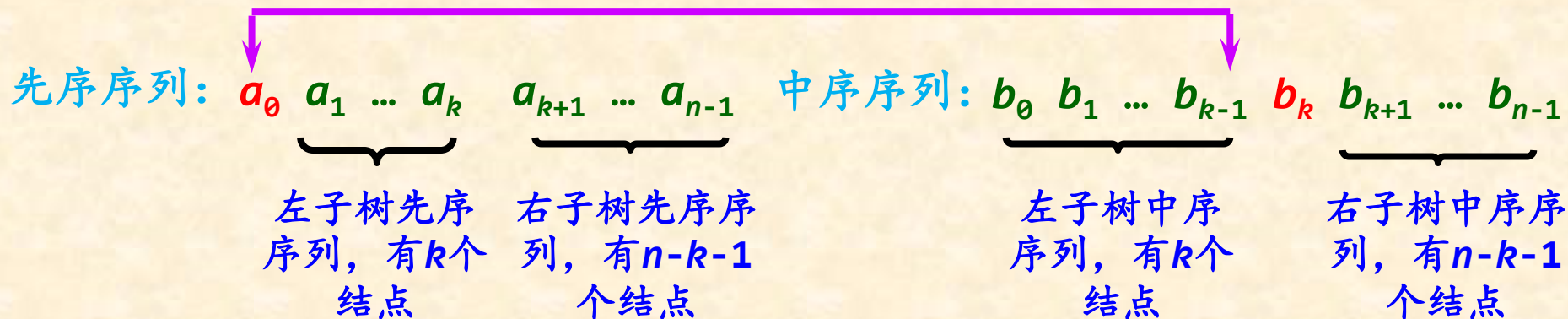
求解其左、右子树的相同功能为“小问题”。

由大小问题之间的解关系得到**递归体**。

再考虑特殊情况，通常是二叉树为空或者只有一个结点时，这时很容易求解，从而得到**递归出口**。

【例2.8】 对于含 n ($n>0$) 个结点的二叉树，所有结点值为`int`类型，设计一个算法由其先序序列 a 和中序序列 b 创建对应的二叉链存储结构。

通过根结点 a_0 在中序序列中找到 b_k



```

BTNode *CreateBTree(ElemType a[],ElemType b[],int n)
//由先序序列a[0..n-1]和中序序列b[0..n-1]建立二叉链存储结构bt
{   int k;
    if (n<=0) return NULL;
    ElemType root=a[0];                                //根结点值
    BTNode *bt=(BTNode *)malloc(sizeof(BTNode));
    bt->data=root;
    for (k=0;k<n;k++)                                  //在b中查找b[k]=root的根结点
        if (b[k]==root)
            break;

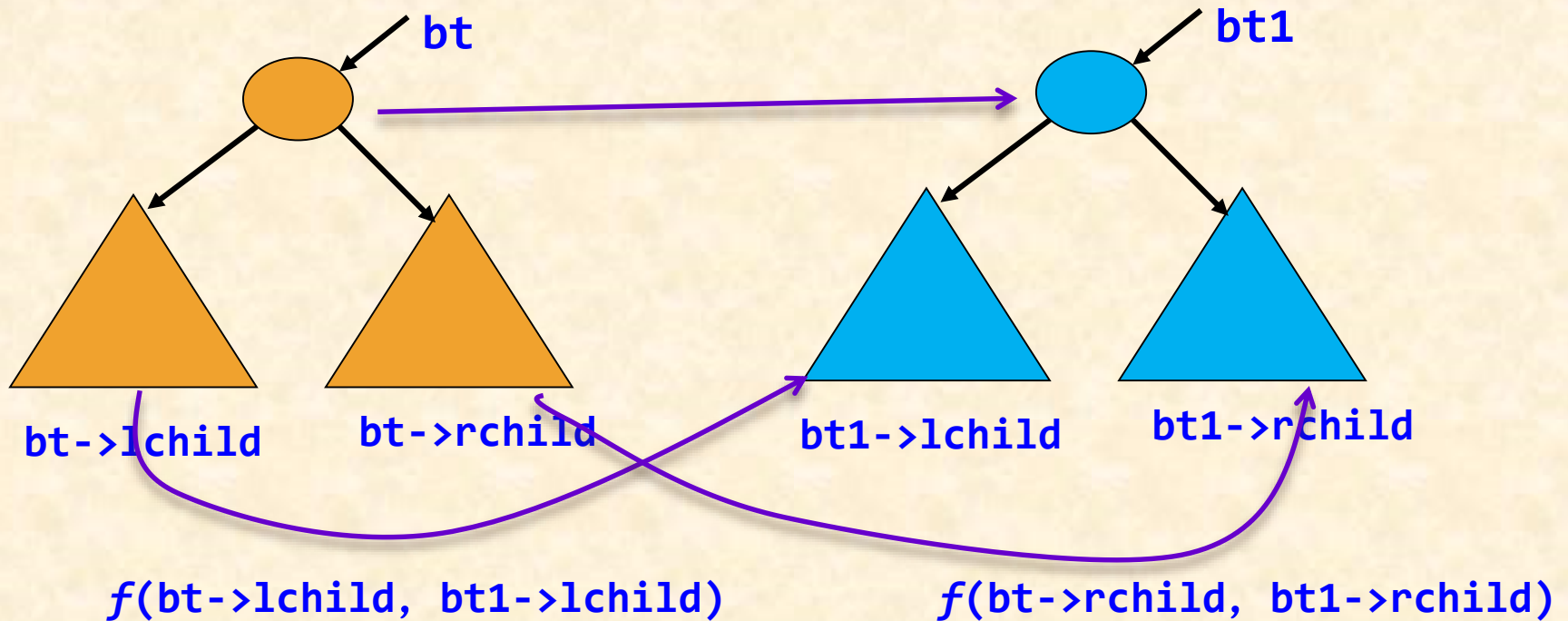
    bt->lchild=CreateBTree(a+1,b,k);                    //递归创建左子树
    bt->rchild=CreateBTree(a+k+1,b+k+1,n-k-1);         //递归创建右子树

    return bt;
}

```

【例2.10】 假设二叉树采用二叉链存储结构，设计一个递归算法由二叉树bt复制产生另一棵二叉树bt1。

解：设 $f(bt, bt1)$ 的功能是由二叉树 bt 复制产生另一棵二叉树 $bt1$ ，它是“大问题”，则 $f(bt \rightarrow lchild, bt1 \rightarrow lchild)$ 的功能就是由 bt 的左子树复制产生 $bt1$ 的左子树， $f(bt \rightarrow rchild, bt1 \rightarrow rchild)$ 的功能就是由 bt 的右子树复制产生 $bt1$ 的右子树，它们是“小问题”。



对应的递归模型如下：

$f(bt, bt1) \equiv bt1=NULL$

$f(bt, bt1) \equiv$ 由bt结点复制产生bt1结点;

$f(bt \rightarrow lchild, bt1 \rightarrow lchild);$

$f(bt \rightarrow rchild, bt1 \rightarrow rchild)$

当b=NULL时
其他情况



```
void CopyBTree(BTNode *bt, BTNode *&bt1)
```

```
//由二叉树bt复制产生bt1
```

```
{ if (bt==NULL)
```

```
    bt1=NULL;
```

```
else
```

```
{ bt1=(BTNode *)malloc(sizeof(BTNode));
```

```
  bt1->data=bt->data;
```

```
  CopyBTree(bt->lchild, bt1->lchild);
```

```
  CopyBTree(bt->rchild, bt1->rchild);
```

```
}
```

```
}
```

2.2.4 基于归纳思想的递归算法设计

基于归纳思想的递归算法设计通常不像基于递归数据结构的递归算法设计那样直观，需要通过对求解问题的深入分析，提炼出求解过程中的相似性而不是数据结构的相似性，这就增加了算法设计的难度。

但现实世界中的许多问题的求解都隐含这种相似性，并体现计算思维的特性。

【例2.12】设计一个递归算法，输出一个大于零的十进制数 n 的各数字位，如 $n=123$ ，输出各数字位为123。

解：设 n 为 m 位十进制数 $a_{m-1}a_{m-2}\dots a_1a_0$ ($m>0$)，则有：

$$n\%10=a_0, \quad n/10=a_{m-1}a_{m-2}\dots a_1。$$

设 $f(n)$ 的功能是输出十进制数 n 的各数字位，则 $f(n/10)$ 的功能是输出除 a_0 （即 $n\%10$ ）外的各数字位，前者是“大问题”，后者是“小问题”。

$f(n) \equiv$ 不做任何事件

$f(n) \equiv f(n/10)$; 输出 $n\%10$

当 $n=0$ 时

其他情况

```
void digits(int n)
{  if (n!=0)
    {  digits(n/10);
        printf("%d", n%10);
    }
}
```

2.3 递归算法设计示例

2.3.1 简单选择排序和冒泡排序

【问题描述】 对于给定的含有 n 个元素的数组 a ，分别采用简单选择排序和冒泡排序方法对其按元素值递增排序。

冒泡和选择的区别：

冒泡排序是左右两个数相比较，而选择排序是用后面的数和每一轮的第一个数相比较；

冒泡排序每轮交换的次数比较多，而选择排序每轮只交换一次；

冒泡排序是通过数去找位置，选择排序是给定位置去找数；

当一个数组遇到相同的数时，冒泡排序相对而言是稳定的，而选择排序便不稳定；

在时间效率上，选择排序优于冒泡排序。

https://blog.csdn.net/a745233700/article/details/86683603?spm=1001.2101.3001.6650.3&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-3-86683603-blog-122094220.pc_relevant_multi_platform_whitelistv4&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-3-86683603-blog-122094220.pc_relevant_multi_platform_whitelistv4&utm_relevant_index=6

1. 简单选择排序

设 $f(a, n, i)$ 用于对 $a[i..n-1]$ 元素序列（共 $n-i$ 个元素）进行简单选择排序，是“大问题”。

$f(a, n, i+1)$ 用于对 $a[i+1..n-1]$ 元素序列（共 $n-i-1$ 个元素）进行简单选择排序，是“小问题”。

当 $i=n-1$ 时所有元素有序，算法结束。

$f(a, n, i) \equiv$ 不做任何事情，算法结束

当 $i=n-1$

$f(a, n, i) \equiv$ 通过简单比较挑选 $a[i..n-1]$ 中的最

小元素 $a[k]$ 放在 $a[i]$ 处；

否则

$f(a, n, i+1)$;

```

void SelectSort(int a[], int n, int i)
{   int j, k;
    if (i==n-1) return;           //满足递归出口条件

    else
    {   k=i;                       //k记录a[i..n-1]中最小元素的下标
        for (j=i+1;j<n;j++)       //在a[i..n-1]中找最小元素
            if (a[j]<a[k])
                k=j;
        if (k!=i)                 //若最小元素不是a[i]
            swap(a[i],a[k]);       //a[i]和a[k]交换

        SelectSort(a, n, i+1);
    }
}

```

2. 冒泡排序

设 $f(a, n, i)$ 用于对 $a[i..n-1]$ 元素序列（共 $n-i$ 个元素）进行冒泡排序，是“大问题”，则 $f(a, n, i+1)$ 用于对 $a[i+1..n-1]$ 元素序列（共 $n-i-1$ 个元素）进行冒泡排序，是“小问题”。当 $i=n-1$ 时所有元素有序，算法结束。

$f(a, n, i) \equiv$ 不做任何事情，算法结束

当 $i=n-1$

$f(a, n, i) \equiv$ 对 $a[i..n-1]$ 元素序列，从 $a[n-1]$ 开始

进行相邻元素比较；

否则

若相邻两元素反序则将两者交换；

若没有交换则返回，否则执行 $f(a, n, i+1)$ ；

```

void BubbleSort(int a[], int n, int i)
{   int j;
    bool exchange;
    if (i==n-1) return;           //满足递归出口条件

    else
    {   exchange=false;           //置exchange为false
        for (j=n-1;j>i;j--)
            if (a[j]<a[j-1])       //当相邻元素反序时
            {   swap(a[j],a[j-1]);
                exchange=true;    //发生交换置exchange为true
            }
        if (exchange==false)      //未发生交换时直接返回
            return;
        else                       //发生交换时继续递归调用
            BubbleSort(a, n, i+1);
    }
}

```


2.4* 递归算法转化非递归算法

把递归算法转化为非递归算法有如下两种基本方法：

(1) 直接用循环结构的算法替代递归算法。

(2) 用栈模拟系统的运行过程，通过分析只保存必须保存的信息，从而用非递归算法替代递归算法。

第(1)种是直接转化法，不需要使用栈。第(2)种是间接转化法，需要使用栈。

2.4.1 用循环结构替代递归过程

采用循环结构消除递归这种直接转化法没有通用的转换算法，对于具体问题要深入分析对应的递归结构，设计有效的循环语句进行递归到非递归的转换。

直接转化法特别适合于尾递归。尾递归只有一个递归调用语句，而且是处于算法的最后。

例如，采用循环结构求 $n!$ 的非递归算法fun1(n)如下：

```
int fun1(int n)
{   int f=1, i;
    for (i=2;i<=n;i++)
        f=f*i;
    return(f);
}
```

除尾递归外，直接转化法也适合于单向递归。

单向递归是指递归函数中虽然有一处以上的递归调用语句，但各次递归调用语句的参数只和主调用函数有关，相互之间参数无关，并且这些递归调用语句也和尾递归一样处于算法的最后。

采用循环结构求解Fibonacci数列的非递归算法如下：

```
int Fib1(int n)
{   int i, f1, f2, f3;
    if (n==1 || n==2)
        return(1);
    f1=1;f2=1;
    for (i=3;i<=n;i++)
    {   f3=f1+f2;
        f1=f2;
        f2=f3;
    }
    return(f3);
}
```

2.4.2 用栈消除递归过程

通常使用栈保存中间结果，从而将递归算法转化为非递归算法的过程。

在设计栈时，除了保存递归函数的参数等外，还增加一个标志成员（tag），对于某个递归小问题 $f(s')$ ，其值为1表示对应递归问题尚未求出，需进一步分解转换，为0表示对应递归问题已求出，需通过该结果求解大问题 $f(s)$ 。

为了方便讨论，将递归模型分为等值关系和等价关系两种。

1. 等值关系

等值关系是指“大问题”的函数值等于“小问题”的函数值的某种运算结果。

例如求 $n!$ 对应的递归模型就是等值关系。

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

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

以上 (2) 式中有一次分解过程: $f(n) \Rightarrow f(n-1)$, 对应的求值过程是: $f(n-1) \Rightarrow f(n)=n*f(n-1)$ 。

设计一个栈St, 其结构如下:

```
typedef struct
{
    int n;           //保存n值
    int f;           //保存f(n)值
    int tag;         //标识是否求出f(n)值,1:未求出,0:已求出
} NodeType;         //栈元素类型
```


设求 $n!$ 的非递归算法为 $\text{fun2}(n)$ ($n \geq 1$) 值的过程如下:

```
将(n, *, 1)进栈;                //其中*表示没有设定值
while (栈不空)
{
    if (栈顶元素未计算出f值即st.top().tag==1)
    {
        if (栈顶元素满足(1)式, 即st.top().n=1)
            求出栈顶元素的f值为1, 并置栈顶元素的tag=0表示已求出对应的函数值;
        else
            //栈顶元素满足(2)式
            将子任务(st.top().n-1, *, 1)进栈;    //分解过程
    }
    else
        //栈顶元素f值已求出即st.top().tag=0
        退栈栈顶元素, 由其f值计算出新栈顶元素的f值; //求值过程
    if (栈中只有一个已求出f值的元素)
        退出循环;
}
st.top()f即为所求的fun2(n)值;
```

```

int fun2(int n)           //求n!的递归算法转换成的非递归算法
{   NodeType e,e1,e2;
    stack<NodeType> st;
    e.n=n;
    e.tag=1;
    st.push(e);           //初值进栈
    while (!st.empty())   //栈不空时循环
    {   if (st.top().tag==1) //未计算出栈顶元素的f值
        {   if (st.top().n==1) // (1)式即递归出口
            {   st.top().f=1;
                st.top().tag=0;
            }
            else // (2)式分解过程
            {   e1.n=st.top().n-1;
                e1.tag=1;
                st.push(e1); //子任务(n-1)!进栈
            }
        }
    }
}

```

```

else                                     //st.top().tag=0即已计算出f值
{
    e2=st.top();
    st.pop();                          //退栈e2
    st.top().f=st.top().n*e2.f;         //(2)式求值过程
    st.top().tag=0;                    //表示栈顶元素的f值已求出
}
if (st.size()==1 && st.top().tag==0)
    //栈中只有一个已求出f的元素时退出循环
    break;
}
return(st.top().f);
}

```

2. 等价关系

等价关系是指“大问题”的求解过程转化为“小问题”求解而得到的，它们之间不是值的相等关系，而是解的等价关系。

例如，求梵塔问题对应的递归模型就是等价关系，也就是说， $\text{Hanoi}(n, x, y, z)$ 与 $\text{Hanoi}(n-1, x, z, y)$ 、 $\text{move}(n, x, z)$ 和 $\text{Hanoi}(n-1, y, x, z)$ 是等价的。

设计一个栈St，其结构如下：

```
typedef struct
{   int n;           //保存n值
    char x,y,z;      //保存f(n)值
    int tag;         //标识是否求出f(n)值,1:未求出,0:已求出
} NodeType;
```

对应的非递归求解过程如下：

```
定义一个栈；
将初始问题进栈；
while (栈不空)
{   if (栈顶元素的tag==1)           //不能直接操作
    {   出栈一个元素；
        将Hanoi( $n-1$ ,  $y$ ,  $x$ ,  $z$ )进栈(若满足递归出口条件则将tag置为0；
            否则置为1)；
        将“将第 $n$ 个圆盘从 $x$ 移动到 $z$ 上”操作进栈(将tag置为0)；
        将Hanoi( $n-1$ ,  $x$ ,  $z$ ,  $y$ )进栈(若满足递归出口条件则将tag置为0；
            否则置为1)；
    }
    if (栈顶元素满足递归出口条件)
        直接操作并置tag=0;
}
```

注意：上述过程中进栈的次序与递归体中三步的求解次序正好相反，这是由于梵塔问题和栈的特点决定的。

```

void Hanoi1(int n,char x,char y,char z)
//求Hanoi递归算法转换成的非递归算法
{   NodeType e,e1,e2,e3;
    stack<NodeType> st;
    e.n=n;
    e.x=x; e.y=y; e.z=z;
    e.tag=1;
    st.push(e);           //初值进栈
    while (!st.empty())   //栈不空循环
    {   if (st.top().tag==1) //当不能直接操作时
        {   e=st.top();
            st.pop();           //退栈hanoi(n,x,y,z)

```

```

    e1.n=e.n-1;           //产生子任务3: Hanoi(n-1,y,x,z)
    e1.x=e.y; e1.y=e.x; e1.z=e.z;
    if (e1.n==1)          //只有一个盘片时直接操作
        e1.tag=0;
    else                   //否则需要继续分解
        e1.tag=1;
    st.push(e1);           //子任务3进栈
    e2.n=e.n;              //产生子任务2: move(n,x,z)进栈
    e2.x=e.x; e2.z=e.z;
    e2.tag=0;
    st.push(e2);           //子任务2进栈
    e3.n=e.n-1;           //产生子任务1: Hanoi(n-1,x,z,y)
    e3.x=e.x; e3.y=e.z; e3.z=e.y;
    if (e3.n==1)          //只有一个盘片时直接操作
        e3.tag=0;
    else                   //否则需要继续分解
        e3.tag=1;
    st.push(e3);           //子任务1进栈
}
else if (st.top().tag==0) //当可以直接操作时
{
    printf("\t将第%d个盘片从%c移动到%c\n",
           st.top().n,st.top().x,st.top().z);
    st.pop();              //移动盘片后退栈
}
}
}
}

```


2.5 递推式的计算

2.5.1 用特征方程求解递归方程

1. 线性齐次递推式的求解

常系数的线性齐次递推式的一般格式如下：

$$f(n)=a_1f(n-1)+a_2f(n-2)+\dots+a_kf(n-k) \quad (2.5)$$

$$f(i)=b_i \quad 0 \leq i < k$$

$$f(n)=a_1f(n-1)+a_2f(n-2)+ \dots +a_kf(n-k) \quad (2.5)$$

$$f(i)=b_i \quad 0 \leq i < k$$

等式(2.5)的一般解含有 $f(n)=x^n$ 形式的特解的和，用 x^n 来代替该等式中的 $f(n)$ ，则 $f(n-1)=x^{n-1}$ ，...， $f(n-k)=x^{n-k}$ ，所以有：

$$x^n=a_1x^{n-1}+a_2x^{n-2}+\dots+a_kx^{n-k}$$

两边同时除以 x^{n-k} 得到：

$$x^k=a_1x^{k-1}+a_2x^{k-2}+\dots+a_k$$

或者写成：

$$x^k-a_1x^{k-1}-a_2x^{k-2}-\dots-a_k=0 \quad (2.6)$$

$$x^k - a_1 x^{k-1} - a_2 x^{k-2} - \dots - a_k = 0 \quad (2.6)$$

等式(2.6)称为递推关系(2.5)的特征方程。可以求出特征方程的根, 如果该特征方程的 k 个根互不相同, 令其为 r_1 、 r_2 、...、 r_k , 则得到递归方程的通解为:

$$f(n) = c_1 r_1^n + c_2 r_2^n + \dots + c_k r_k^n$$

再利用递归方程的初始条件 ($f(i) = b_i$, $0 \leq i < k$), 确定通解中的待定系数, 从而得到递归方程的解。

下面仅讨论几种简单且常用的齐次递推式的求解过程。

(1) 对于一阶齐次递推关系，如 $f(n)=af(n-1)$ ，假定序列从 $f(0)$ 开始，且 $f(0)=b$ ，可以直接递推求解，即：

$$f(n)=af(n-1)=a_2f(n-2)=\dots=a_nf(0)=a^nb$$

可以看出 $f(n)=a^nb$ 是递推式的解。

(2) 对于二阶齐次递推关系, 如 $f(n)=a_1f(n-1)+a_2f(n-2)$, 假定序列从 $f(0)$ 开始, 且 $f(0)=b_1, f(1)=b_2$ 。

其特征方程为 $x^2-a_1x-a_2=0$, 令这个二次方程的根是 r_1 和 r_2 , 可以求解递推式的解是:

$$f(n)=c_1r_1^n+c_2r_2^n \quad \text{当 } r_1 \neq r_2$$

$$f(n)=c_1r^n+c_2nr^n \quad \text{当 } r_1=r_2=r$$

代入 $f(0)=b_1, f(1)=b_2$ 求出 c_1 和 c_2 。

【例2.13】 分析求解Fibonacci数列的递归算法的时间复杂度。

解： 对于求Fibonacci数列的递归算法，有以下递归关系式 $f(n)$ ：

$$f(n)=1 \quad \text{当 } n=1 \text{ 或 } 2 \text{ 时}$$

$$f(n)=f(n-1)+f(n-2)+1 \quad \text{当 } n>2 \text{ 时}$$

为了简化解，可以引入额外项 $f(0)=0$ 。其特征方程是 $x^2-x-1=0$ ，求得根为：

$$r_1 = \frac{1+\sqrt{5}}{2}, \quad r_2 = \frac{1-\sqrt{5}}{2}$$

由于 $r_1 \neq r_2$ ，这样递推式的解是 $f(n) = c_1 \left(\frac{1+\sqrt{5}}{2} \right)^n + c_2 \left(\frac{1-\sqrt{5}}{2} \right)^n$

为求 c_1 和 c_2 ，求解下面两个联立方程：

$$f(0)=0=c_1+c_2, \quad f(1)=1=c_1\left(\frac{1+\sqrt{5}}{2}\right)+c_2\left(\frac{1-\sqrt{5}}{2}\right)$$

$$\text{求得： } c_1 = \frac{1}{\sqrt{5}}, \quad c_2 = -\frac{1}{\sqrt{5}}$$

$$\text{所以， } f(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n = O(\phi^n),$$

$$\text{其中 } \phi = \frac{1+\sqrt{5}}{2}。$$

2. 非齐次递推式的求解

常系数的线性非齐次递推式的一般格式如下：

$$f(n)=a_1f(n-1)+a_2f(n-2)+ \dots +a_kf(n-k)+g(n) \quad (2.7)$$

$$f(i)=b_i \quad 0 \leq i < k$$

其通解形式如下：

$$f(n)=f'(n)+f''(n)$$

其中， $f'(n)$ 是对应齐次递归方程的通解， $f''(n)$ 是原非齐次递归方程的特解。

现在还没有有一种寻找特解的有效方法。一般是根据 $g(n)$ 的形式来确定特解。

假设 $g(n)$ 是 n 的 m 次多项式, 即 $g(n)=c_0n^m+...+c_{m-1}n+c_m$

则特解 $f''(n)=A_0n^m+A_1n^{m-1}+...+A_{m-1}n+A_m$ 。

代入原递归方程求出 A_0 、 A_1 、...、 A_m 。

再代入初始条件 ($f(i)=b_i$, $0 \leq i < k$) 求出系数得到最终通解。

有些情况下非齐次递推式的系数不一定是常系数。下面仅讨论几种简单且常用的非齐次递推式的求解过程。

$$(1) f(n)=f(n-1)+g(n) \quad (n \geq 1) \quad \text{且} \quad f(0)=0 \quad (2.8)$$

其中 $g(n)$ 是另一个序列。通过递推关系容易推出 (2.8) 的解是

$$f(n)=f(0)+\sum_{i=1}^n g(i)$$

例如，递推式 $f(n)=f(n-1)+1$ ，且 $f(0)=0$ 的解是 $f(n)=n$ 。

$$(2) \ f(n)=g(n)f(n-1) \ (n \geq 1) \text{ 且 } f(0)=1 \quad (2.9)$$

通过递推关系推出 (2.9) 的解是

$$f(n)=g(n)g(n-1) \dots g(1)f(0)$$

例如, 递推式 $f(n)=nf(n-1)$, 且 $f(0)=1$ 的解是 $f(n)=n!$ 。

$$(3) f(n)=nf(n-1)+n! \quad (n \geq 1) \text{ 且 } f(0)=0$$

(2.10)

其求解过程如下：

$$f(n) = nf(n-1)+n!=n[(n-1)f(n-2)+(n-1)!]+n!$$

$$= n(n-1)f(n-2)+2n! = n!(f(n-2)/(n-2)!+2)$$

构造一个辅助函数 $f'(n)$ ，令 $f(n)=n!f'(n)$ ， $f(0)=f'(0)=0$ ，代入(2.10)式有

$$n!f'(n)=n(n-1)!f'(n-1)+n!$$

$$\text{简化为：} f'(n) = f'(n-1)+1$$

$$\text{它的解为：} f'(n) = f'(0) + \sum_{i=1}^n 1 = 0+n = n$$

$$\text{因此，} f(n) = n!f'(n) = nn!。$$

【例2.14】求以下非齐次方程的解：

$$f(n)=7f(n-1)-10f(n-2)+4n^2$$

$$f(0)=1$$

$$f(1)=2$$

$$f(n)=7f(n-1)-10f(n-2)+4n^2$$

$$f(0)=1$$

$$f(1)=2$$

解：对应的齐次方程为： $f(n)=7f(n-1)-10f(n-2)$ ，其特征方程为： $x^2-7x+10=0$ ，求得其特征根为： $q_1=2$ ， $q_2=5$ 。所以对应的齐次递归方程的通解为 $f'(n)=c_1 2^n+c_2 5^n$ 。

由于 $g(n)=4n^2$ ，则令非齐式递归方程的特解为

$$f''(n)=A_0 n^2+A_1 n+A_2$$

代入原递归方程，得：

$$A_0 n^2+A_1 n+A_2=7[A_0(n-1)^2+A_1(n-1)+A_2]-10[A_0(n-2)^2+A_1(n-2)+A_2]+4n^2$$

化简后得到：

$$4A_0 n^2+(-26A_0+4A_1)n+33A_0-13A_1+4A_2=4n^2$$

由此得到联立方程：

$$4A_0=4$$

$$-26A_0+4A_1=0$$

$$33A_0-13A_1+4A_2=0$$

求得： $A_0=1$ ， $A_1=13/2$ ， $A_2=103/8$

所以非齐次递归方程的通解为：

$$f(n)=f'(n)+f''(n)=c_1 2^n+c_2 5^n+n^2+13n/2+103/8$$

代入初始条件 $f(0)=1$ ， $f(1)=2$ ，求得 $c_1=-41/3$ ， $c_2=43/24$ 。

最后非齐次递归方程的通解为：

$$f(n)=-41/3 \times 2^n+43/24 \times 5^n+n^2+13n/2+103/8。$$

2.5.2 递归树方法求解递归方程

用递归树求解递归方程的基本过程是：

- (1) 展开递归方程，构造对应的递归树。
- (2) 把每一层的时间进行求和，从而得到算法时间复杂度的估计。

【例2.15】分析以下递归方程的时间复杂度：

$$T(n)=1$$

当 $n=1$

$$T(n)=2T(n/2)+n^2$$

当 $n>1$

解：构造的递归树如下图所示，当递归树展开时，子问题的规模逐步缩小，当到达递归出口时，即当子问题的规模为1时，递归树不再展开。

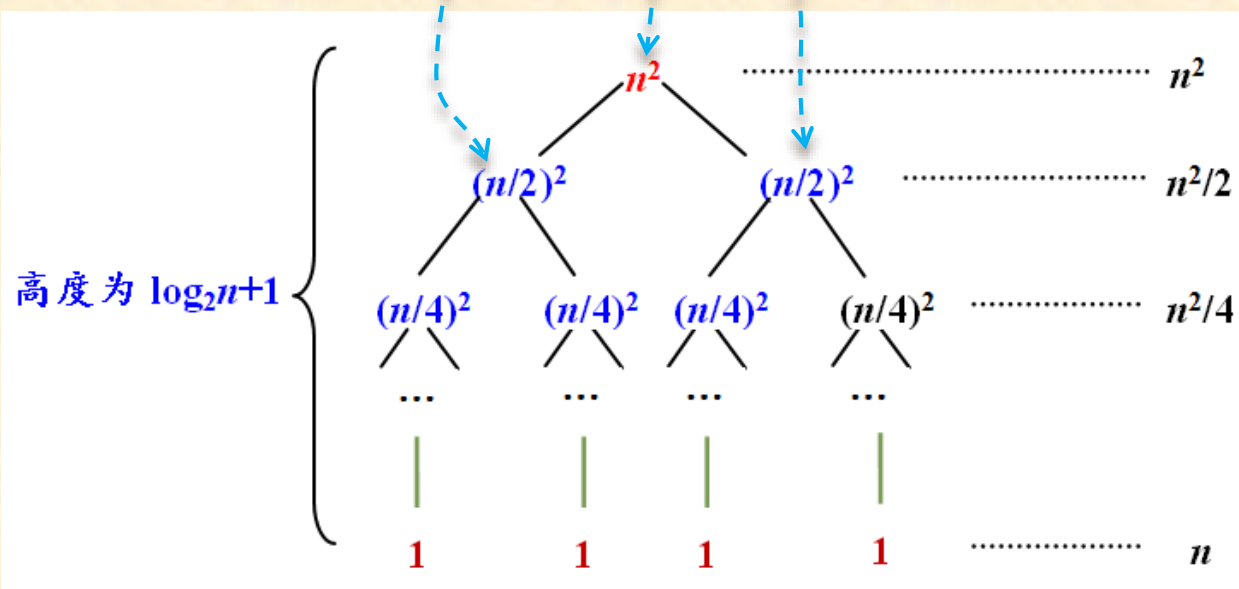
$$T(n)=1$$

当 $n=1$

$$T(n)=2T(n/2)+n^2$$

当 $n>1$

$$\begin{aligned} T(n) &= 2T(n/2) + n^2 = 2[2T(n/4) + (n/2)^2] + n^2 \\ &= 4T(n/4) + 2(n/2)^2 + n^2 \end{aligned}$$



显然在递归树中，第1层的问题规模为 n ，第2的层的问题规模为 $n/2$ ，依此类推，当展开到第 $k+1$ 层，其规模为 $n/2^k=1$ ，所以递归树的高度为 $\log_2 n+1$ 。

第1层有1个结点，其时间为 n^2 ，第2层有2个结点，其时间为 $2(n/2)^2=n^2/2$ ，依次类推，第 k 层有 2^k 个结点，其时间为 $2(n/2^k)^2=n^2/2^{k-1}$ 。

叶子结点的个数为 n 个，其时间为 n 。将递归树每一层的时间加起来，可得：

$$T(n)=n^2+n^2/2+ \dots +n^2/2^{k-1}+ \dots +n=O(n^2)。$$

【例2.16】 分析以下递归方程的时间复杂度：

$$T(n)=1 \quad \text{当 } n=1$$

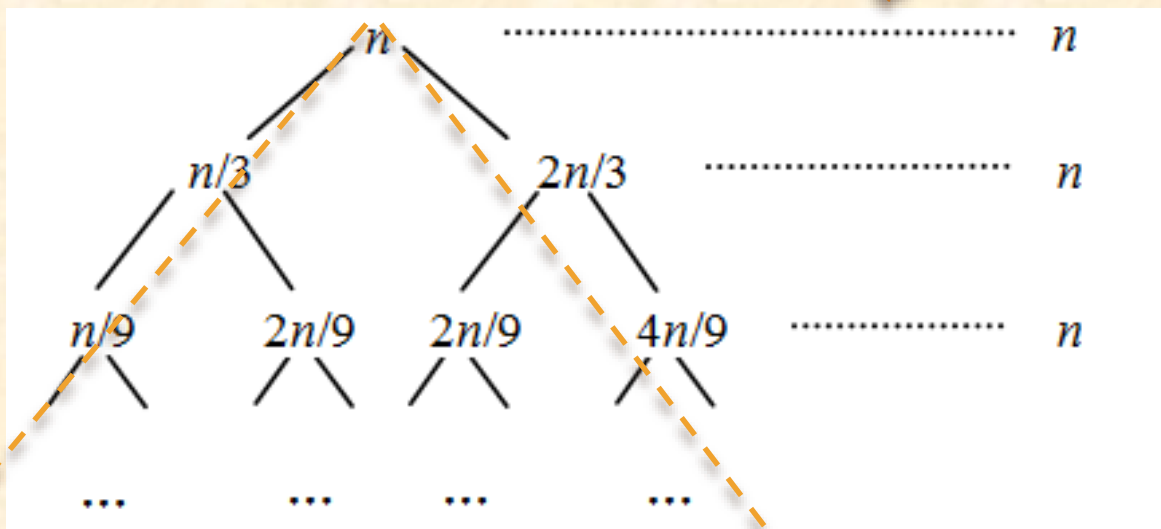
$$T(n)=T(n/3)+T(2n/3)+n \quad \text{当 } n>1$$

$$T(n)=1$$

当 $n=1$

$$T(n)=T(n/3)+T(2n/3)+n \quad \text{当 } n>1$$

递归树



最短路径

最长路径

在最坏情况下，考虑最长的路径。

设最长路径的长度为 h ，有 $n(2/3)^h=1$ ，求出 $h=\log_{3/2}n$ ，因此这棵递归树有 $\log_{3/2}n$ 层，每层结点的数值和为 n ，所以：

$$T(n)=O(n\log_{3/2}n)=O(n\log_2n)。$$

2.5.3 主方法

主方法 (master method) 提供了解如下形式递归方程的一般方法：

$$T(n)=aT(n/b)+f(n) \quad (2.11)$$

其中 $a \geq 1$, $b > 1$ 为常数, 该方程描述了算法的执行时间, 算法将规模为 n 的问题分解成 a 个子问题, 每个子问题的大小为 n/b 。

例如, 对于递归方程 $T(n)=3T(n/4)+n^2$, 有: $a=3$, $b=4$, $f(n)=n^2$ 。

主定理： 设 $a \geq 1$, $b > 1$ 为常数, $f(n)$ 为一个函数, $T(n)$ 由 (2.11) 的递归方程定义, 其中 n 为非负整数, 则 $T(n)$ 计算如下:

(1) 若对某些常数 $\epsilon > 0$, 有 $f(n) = O(n^{\log_b a - \epsilon})$, 那么 $T(n) = O(n^{\log_b a})$ 。

(2) 若 $f(n) = O(n^{\log_b a})$, 那么 $T(n) = O(n^{\log_b a} \log_2 n)$ 。

(3) 若对某些常数 $\epsilon > 0$, 有 $f(n) = O(n^{\log_b a + \epsilon})$, 并且对常数 $c < 1$ 与所有足够大的 n , 有 $af(n/b) \leq cf(n)$, 那么 $T(n) = O(f(n))$ 。

应用该定理的过程是，首先把函数 $f(n)$ 与函数进行比较，递归方程的解由这两个函数中较大的一个决定：

情况 (1)，函数 $n^{\log_b a}$ 比函数 $f(n)$ 更大，则 $T(n) = O(n^{\log_b a})$ 。

情况 (2)，函数 $n^{\log_b a}$ 和函数 $f(n)$ 一样大，则 $T(n) = O(n^{\log_b a} \log_2 n)$ 。

情况 (3)，函数 $n^{\log_b a}$ 比函数 $f(n)$ 小，则 $T(n) = O(f(n))$ 。

【例2.17】分析以下递归方程的时间复杂度：

$$T(n)=1 \quad \text{当 } n=1$$

$$T(n)=4T(n/2)+n \quad \text{当 } n>1$$

解：这里 $a=4$ ， $b=2$ ， $f(n)=n$ 。

因此， $n^{\log_2 4} = n^2$ ，比 $f(n)$ 大，满足情况（1），

所以 $T(n) = O(n^{\log_2 4})$

$=O(n^2)$ 。

【例2.18】采用主方法求例2.15递归方程的时间复杂度。

$$T(n)=1$$

当 $n=1$

$$T(n)=2T(n/2)+n^2$$

当 $n>1$

解：这里 $a=2$, $b=2$, $f(n)=n^2$ 。因此, $n^{\log_2 2} = n$, 比 $f(n)$ 小, 满足情况 (3), 所以 $T(n)=O(f(n)) = O(n^2)$, 与采用递归树的结果相同。

