



# 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 递归算法分析





## 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**。



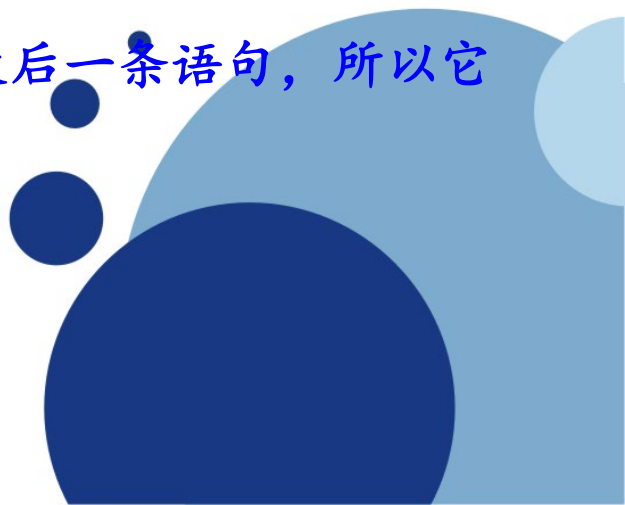


【例】设计求 $n!$  ( $n$ 为正整数) 的递归算法。

解：对应的递归函数如下：

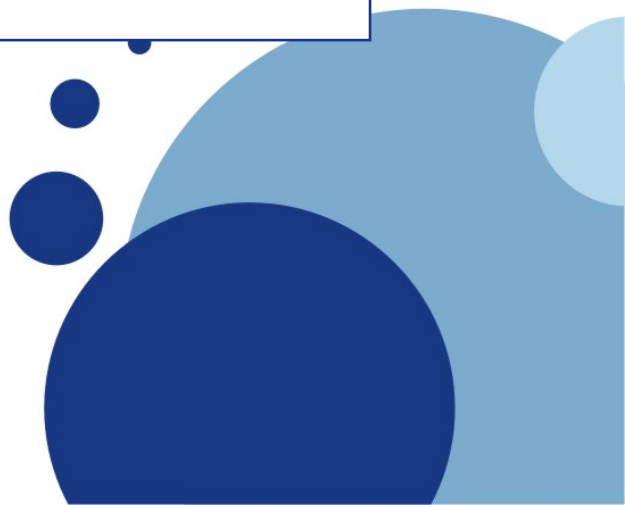
```
int fun(int n)
{  if (n==1)           //语句1
    return(1);         //语句2
    else               //语句3
        return(fun(n-1)*n); //语句4
}
```

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

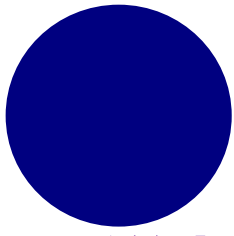
在以下三种情况下，常常要用到递归的方法。

- ✓ 定义是递归的
- ✓ 数据结构是递归的
- ✓ 问题的求解方法是递归的

### 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。

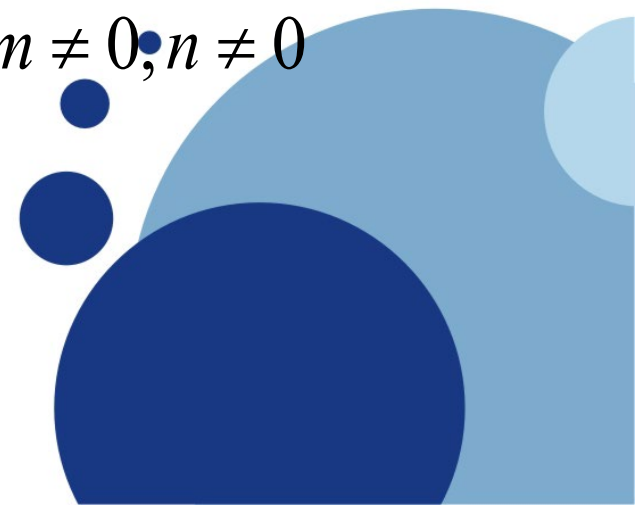




## 递归定义的Ackerman函数

:

$$Ack(m, n) \begin{cases} n + 1 & m = 0 \\ Ack(m - 1, 1) & m \neq 0, n = 0 \\ Ack(m - 1, Ack(m, n - 1)) & m \neq 0, n \neq 0 \end{cases}$$



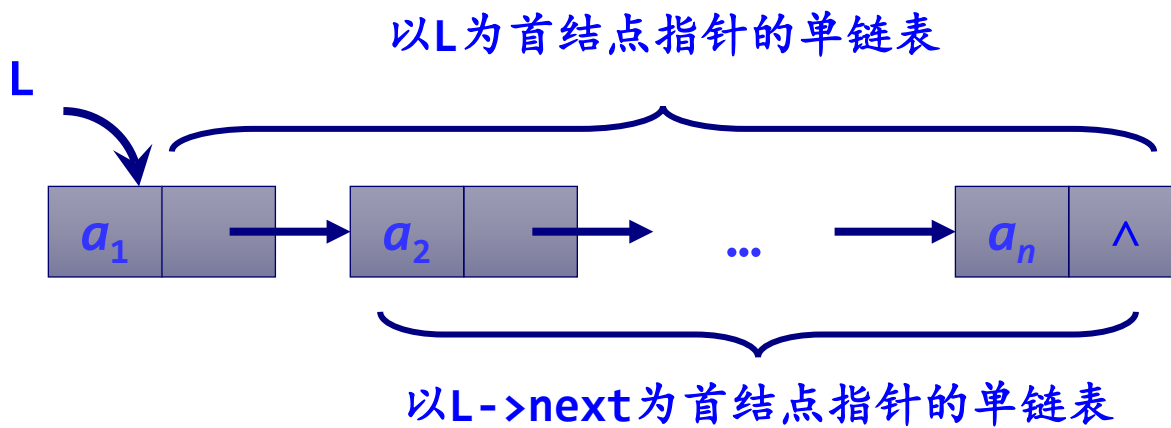
## 2. 数据结构是递归的

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;
```

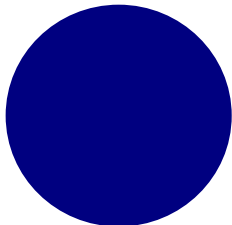
结构体LNode的定义中用到了它自身，即指针域next是一种指向自身类型的指针，所以它是一种递归数据结构。

## 不带头结点单链表示意图




体现出数据结构的递归性。

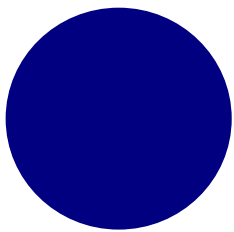




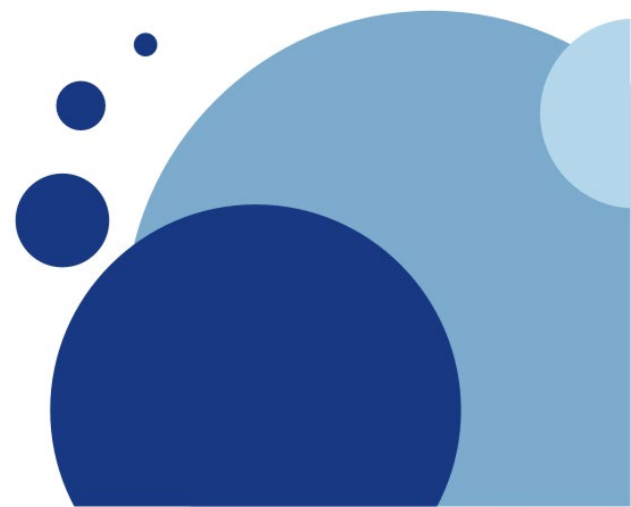
对于递归数据结构，采用递归的方法编写算法既方便又有效。例如，求一个不带头结点的单链表L的所有data域（假设为int型）之和的递归算法如下：

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



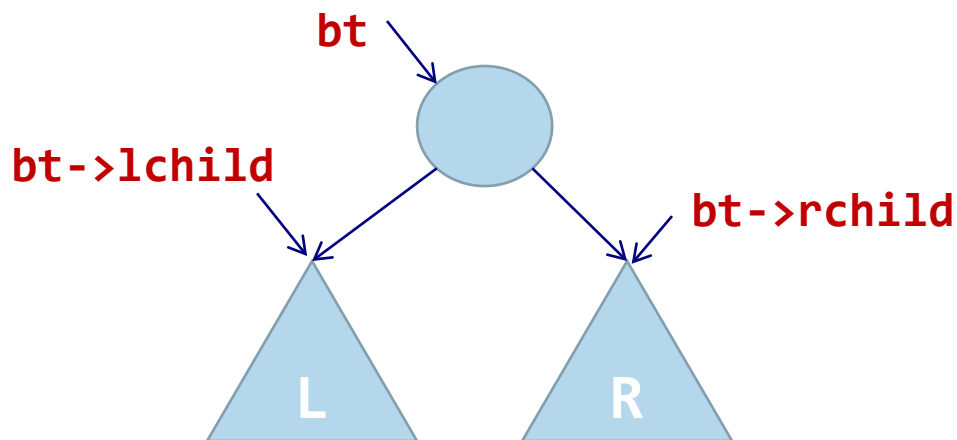


**【例】** 分析二叉树的二叉链存储结构的递归性，设计求非空二叉链bt中所有结点值之和的递归算法，假设二叉链的data域为int型。



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);
}
```

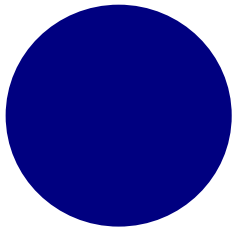
### 3. 问题的求解方法是递归的

有些问题的解法是递归的，典型的有Hanoi问题求解。



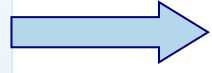
盘片移动时必须遵守以下规则：每次只能移动一个盘片；盘片可以插在X、Y和Z中任一塔座；任何时候都不能将一个较大的盘片放在较小的盘片上。

设计递归求解算法，并将其转换为非递归算法。

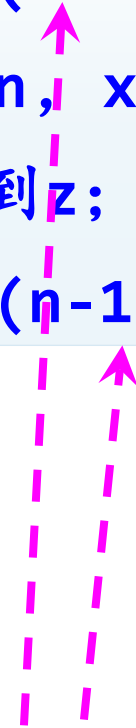
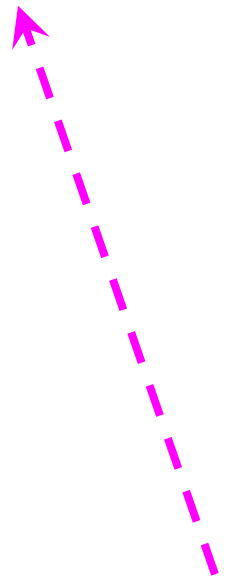


设 $\text{Hanoi}(n, x, y, z)$ 表示将 $n$ 个盘片从 $x$ 通过 $y$ 移动到 $z$ 上，递归分解的过程是：

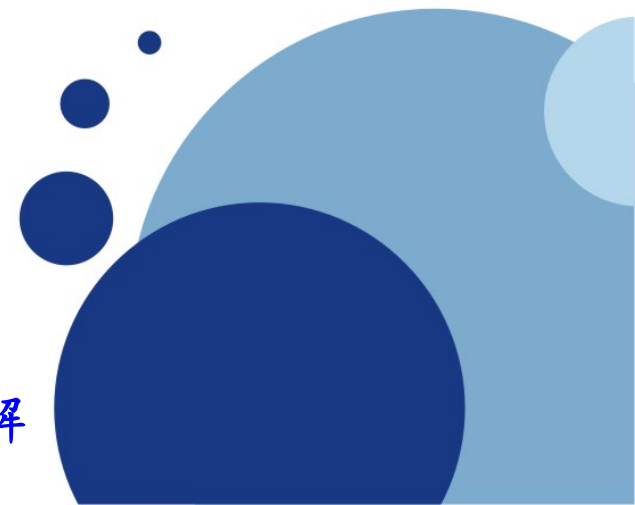
$\text{Hanoi}(n, x, y, z)$



```
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



### 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

:

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

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

递归出口

递归体

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。

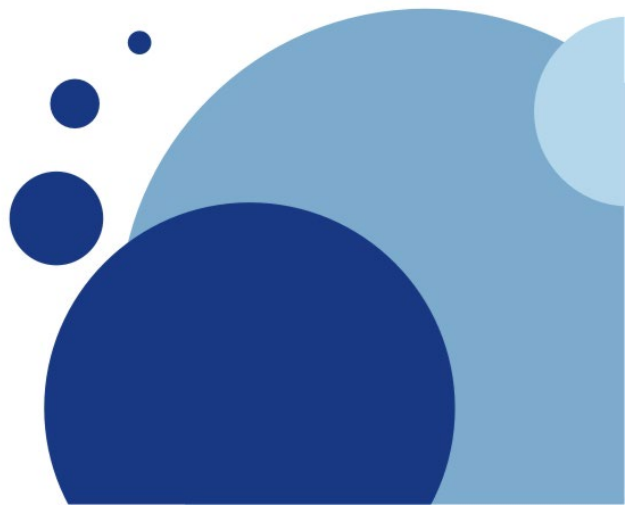
递归体的一般格式如下：

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

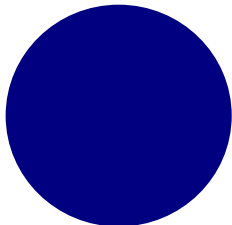


其中， $n$ 、 $i$ 、 $j$ 和 $m$ 均为正整数。这里的 $s_{n+1}$ 是一个递归“大问题”， $s_i$ 、 $s_{i+1}$ 、 $\dots$ 、 $s_n$ 为递归“小问题”， $c_j$ 、 $c_{j+1}$ 、 $\dots$ 、 $c_m$ 是若干个可以直接（用非递归方法）解决的问题， $g$ 是一个非递归函数，可以直接求值。

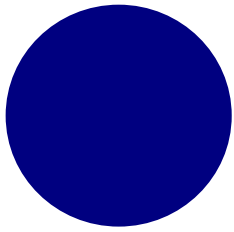


## 2.1.4 递归算法的执行过程

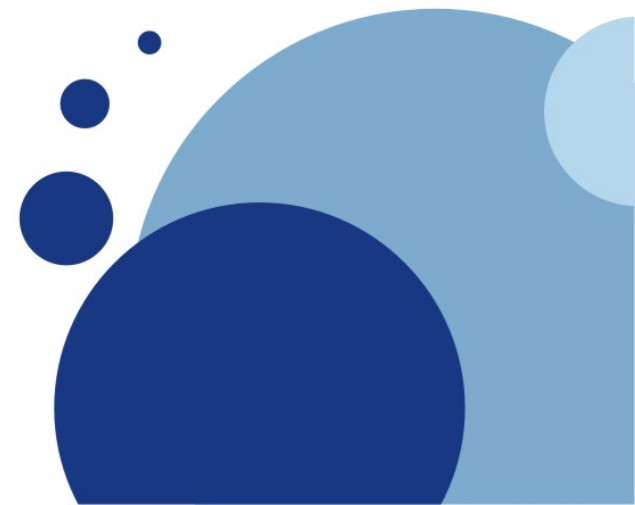
- 一个正确的递归程序虽然每次调用的是相同的子程序，但它的参量、输入数据等均有变化。
  - 在正常的情况下，随着调用的不断深入，必定会出现调用到某一层的函数时，不再执行递归调用而终止函数的执行，遇到递归出口便是这种情况。
- 



- 
- **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
---	----------

$n$

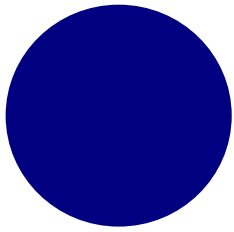
函数值

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

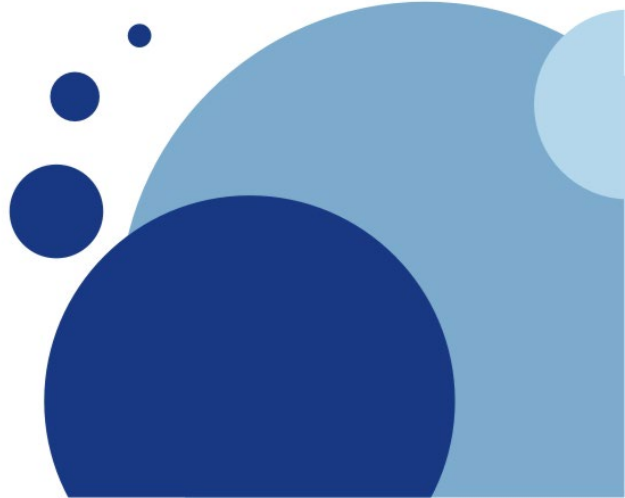


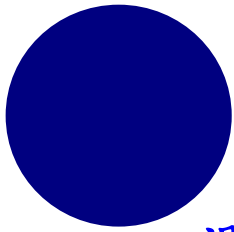
1	1
2	fun(1)*2
3	fun(2)*3
4	fun(3)*4
5	fun(4)*5

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



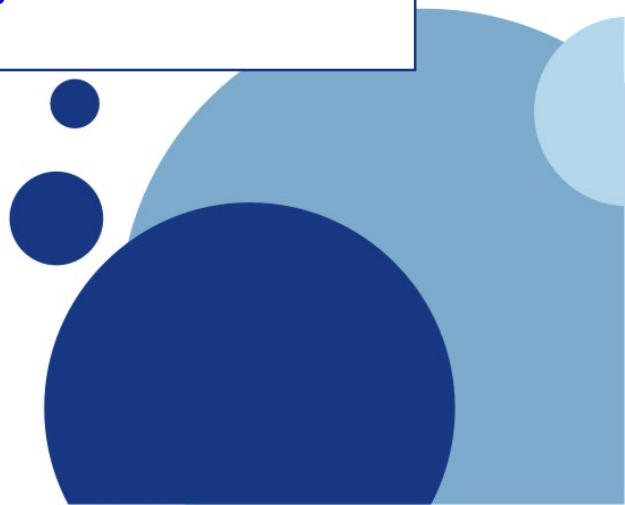
5	$24*5=120$
---	------------

退栈1次并输出120

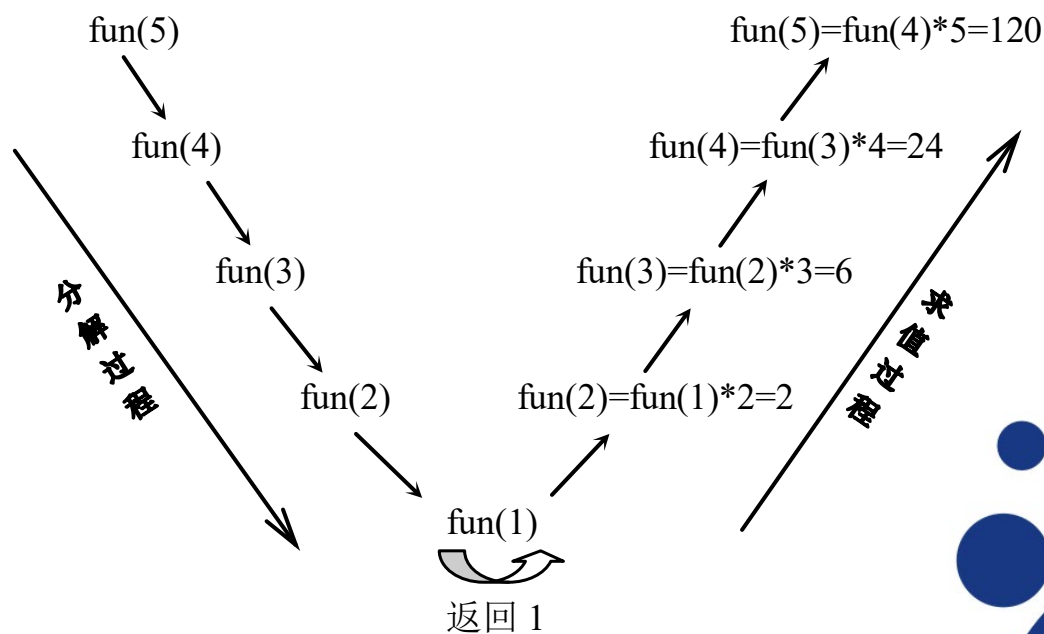




从以上过程可以得出：

- 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】** Fibonacci数列定义为：

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

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

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

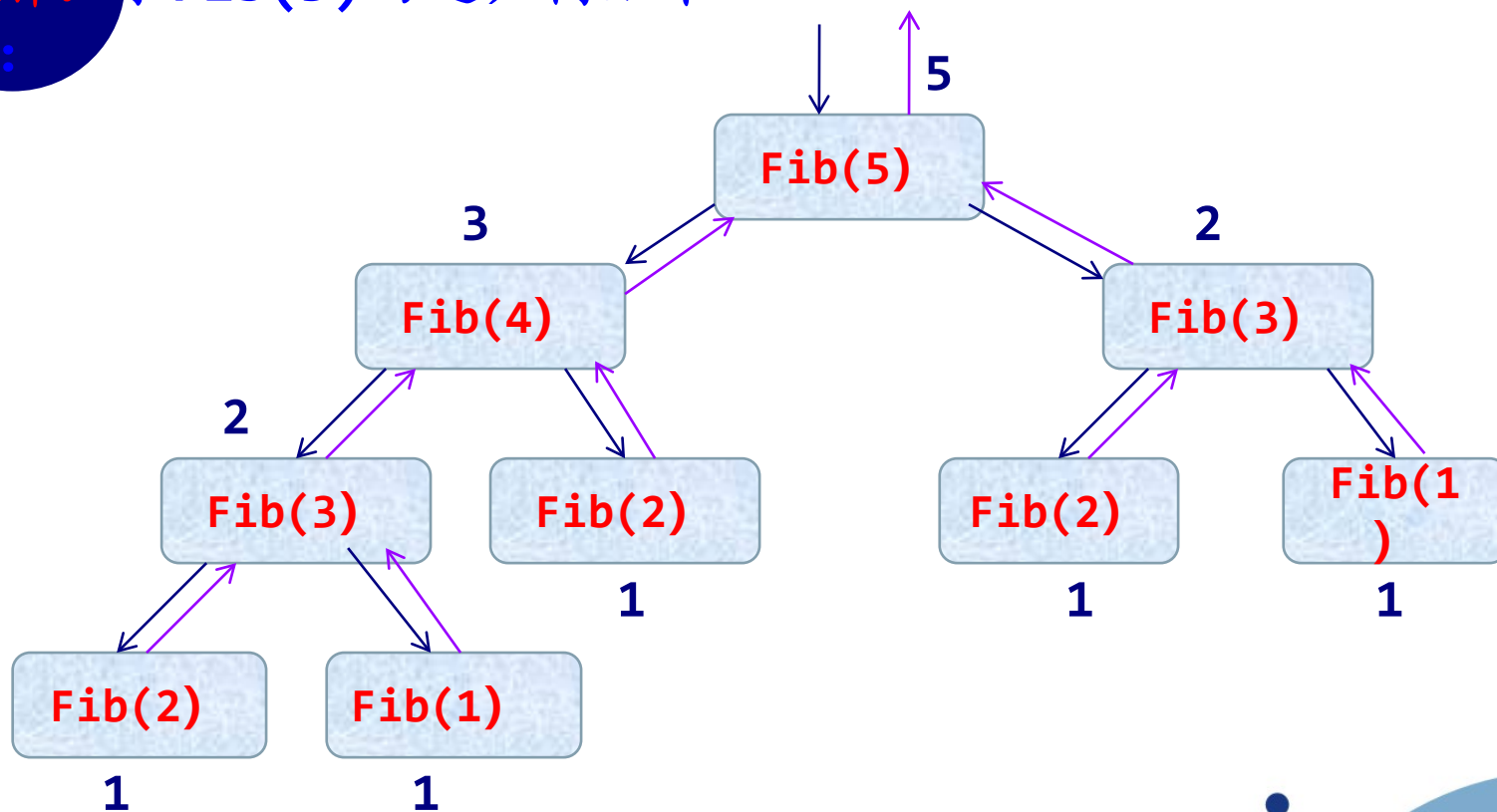
对应的递归算法如下：

```
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



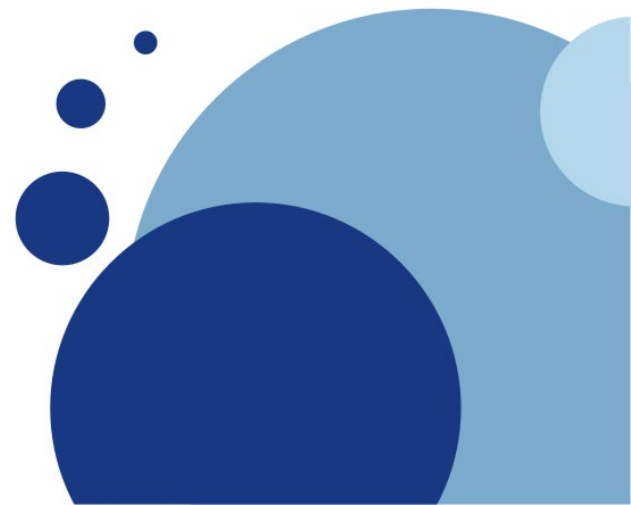
从上面求Fib(5)的过程看到，对于复杂的递归调用，分解和求值可能交替进行、循环反复，直到求出最终值。



## 2.2递归算法设计


### 2.2.1 递归算法设计的一般步骤

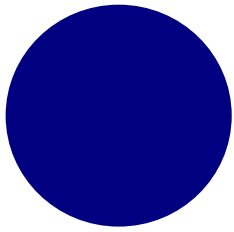
递归算法设计先要给出**递归模型**，再转换成对应的C/C++语言函数。





## 获取递归模型的步骤如下：

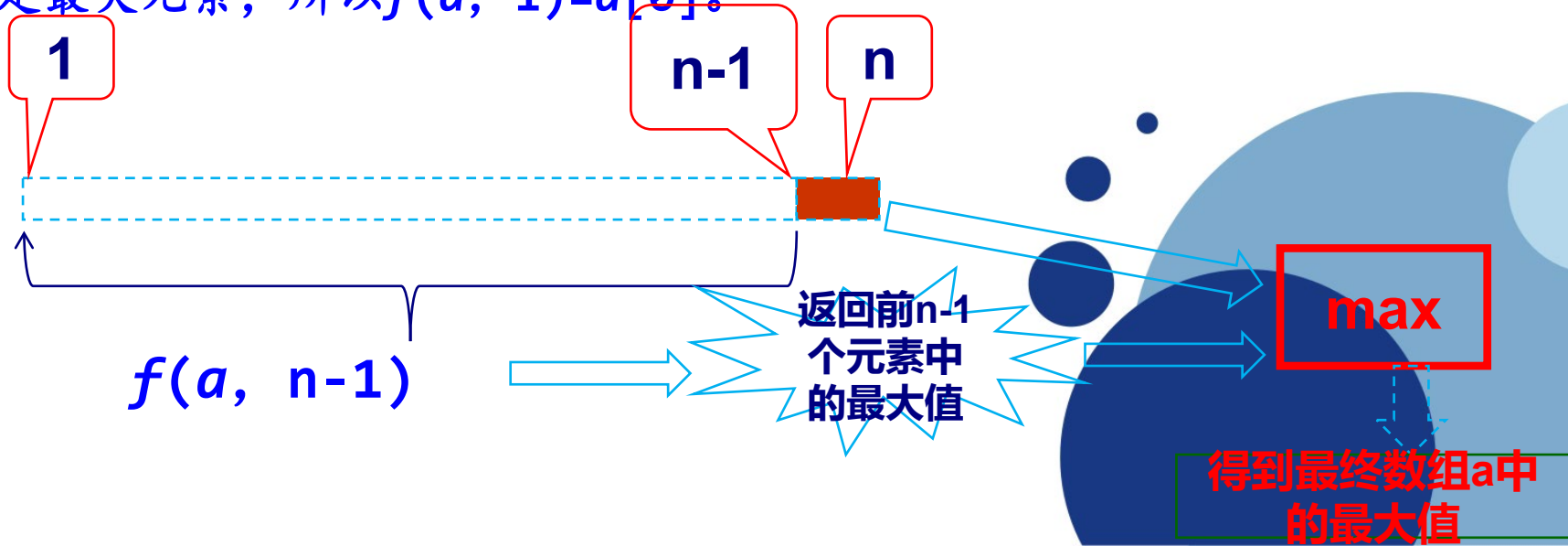
- (1) 对原问题 $f(s_n)$ 进行分析，抽象出合理的“小问题” $f(s_{n-1})$ （与数学归纳法中假设 $n=k-1$ 时等式成立相似）；
  - (2) 假设 $f(s_{n-1})$ 是可解的，在此基础上确定 $f(s_n)$ 的解，即给出 $f(s_n)$ 与 $f(s_{n-1})$ 之间的关系（与数学归纳法中求证 $n=k$ 时等式成立的过程相似）；
  - (3) 确定一个特定情况（如 $f(1)$ 或 $f(0)$ ）的解，由此作为递归出口（与数学归纳法中求证 $n=1$ 或 $n=0$ 时等式成立相似）。
- 



【例】用递归法求一个整数数组 $a$ 的最大元素。

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

假设 $f(a, n-1)$ 已求出，则有 $f(a, n) = \text{MAX}\{f(a, n-1), a[n-1]\}$ 。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：

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

当  $n=1$  时

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

当  $n>1$  时

对应的递归算法如下：

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



设计递归算法求一个数组A[0..n-1]中的最大值元素。

【算法设计思想】采用二分法将A数组分成A[0..mid]和A[mid+1..n-1],分别求出这两个部分的最大值max1和max2,之后max1和max2作比较,返回max(max1, max2)为整个数组A[0..n-1]的最大值。

【形式化描述】

Max(A,i,j)表示数组A中从A[i]~A[j]的最大值,递归模型如下:

Max(A, i, j) = A[i]      当i==j时即A中只有一个元素时;

Max(A, i, j) = max{Max(A,i,m),Max(A, m+1,j)}    其他情况,  
通常m取(i+j)/2

【算法描述】

```
int Max(int A[ ],int i,int j)
{ if(i==j) maxvalue=A[i];
  else{
      mid=(i+j)/2;
      max1=Max(A,i,mid);
      max2=Max(A,mid+1,j);
      maxvalue=(max1>max2?max1:max2);
  }
  rerutn maxvalue;
}
```

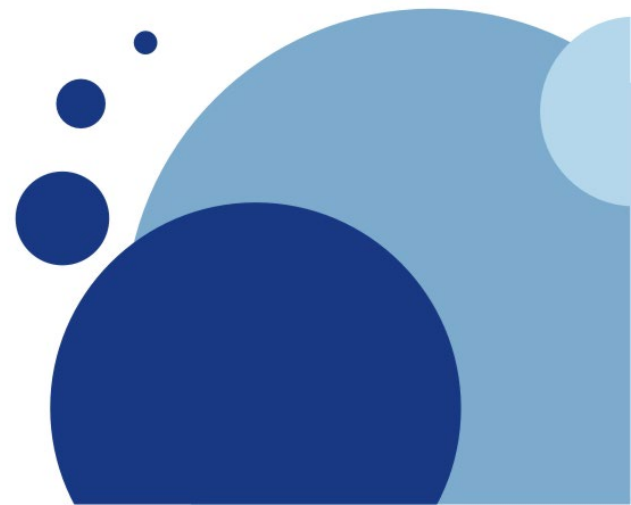


## 2.2递归算法设计

### 2.2.2 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**。






## 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。
- 



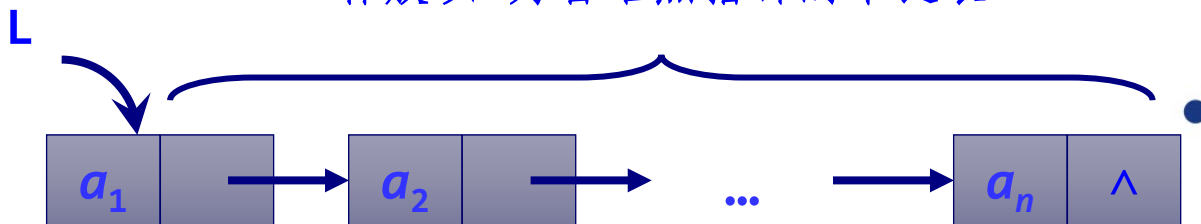
【例】有一个不带头结点的单链表L，设计一个算法释放其中所有结点

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

### 规模大的问题求解

释放以L为首结点指针的单链表



释放以L->next为首结点指针的单链表

### 规模小的同类问题求解

2、设L为不带头结点的单链表，实现从尾到头反向输出链表中每个结点的值。



思考：

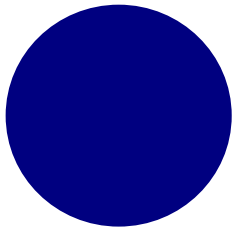
- 若L为NULL，则什么也不做。
- 若L非空，则一个长度为n的单链表的逆序输出可以转化为：
  - 将由L->next引领的长度为n-1的单链表逆序输出；
  - 输出L->data;
- 如上图：

因为：L非空，

- 则先逆序输出（L->next）开头的单链表（递归调用）即：

10,7, 6, 5, 2

- 输出L的data即1
- 最终得到整个单链表的逆序输出。



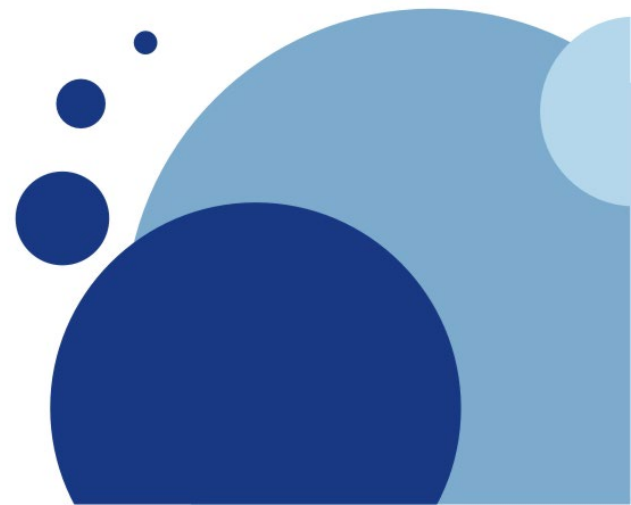
# 递归函数设计



逆序输出单链表中结点的值。

【算法描述】

```
void OutputFromTail(LinkList L)
{
    if(L!=NULL)
    {OutputFromTail(L->next);
      printf(L->data);
    }
}
```



## 2. 基于递归数据结构的递归算法设计

### 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”。转化为：

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**。



对应的递归模型如下：

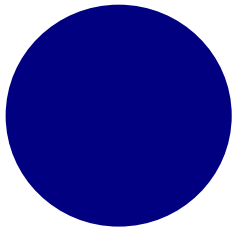
$f(L) \equiv$  不做什么事件

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

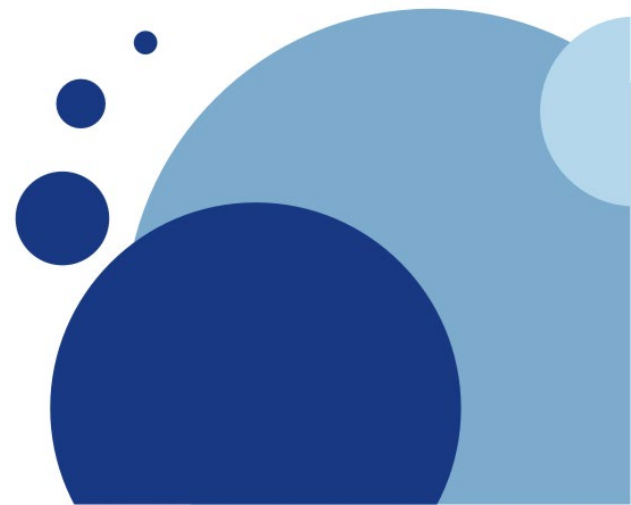
当  $L=NULL$  时  
其他情况



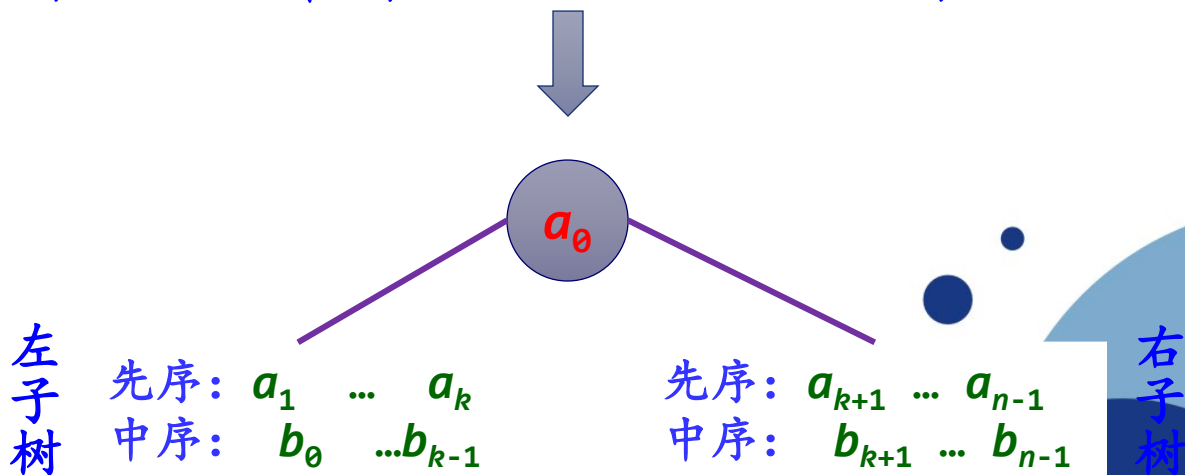
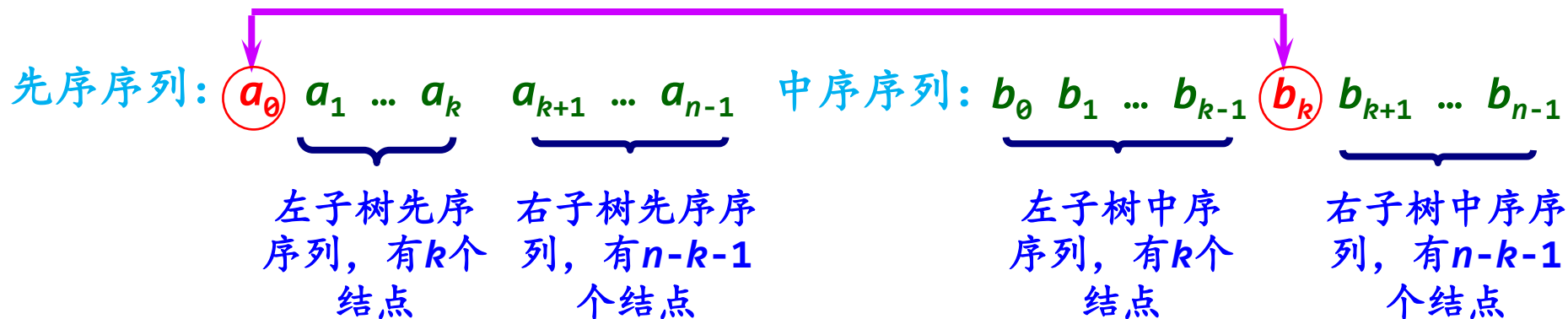
```
void DestroyList(LinkNode *&L)
//释放单链表L中所有结点
{  if (L!=NULL)
    {  DestroyList(L->next);
        free(L);
    }
}
```



**【例】** 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 $b_k$  (即与 $a_0$ 相等)



```
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;
}
```



## 2.3递归算法设计示例

青蛙跳台阶问题：一只青蛙一次可以跳上**1**级台阶，也可以跳上**2**级。编写代码求青蛙跳上一个**n**级的台阶，总共有多少种跳法？

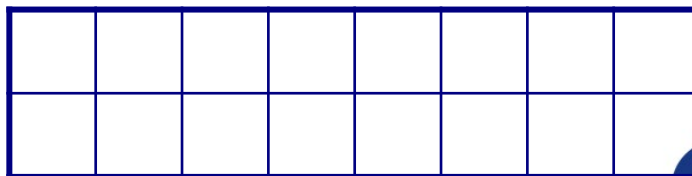
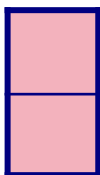
- 拓展
- 若条件改为：一只青蛙一次可以跳上**1**级台阶，也可以跳上**2**级，也可以跳上**3**级，。。。。也可以跳上**n**级。编写代码求青蛙跳上一个**n**级的台阶，总共有多少种跳法？
- **Tips:考虑递归思想分析问题**

## 2.3 递归算法设计示例

3、瓷砖覆盖问题：用一个 $2 \times 1$ 的小矩形横着或竖着去覆盖更大的矩形。如下图

- 具体：用8个 $2 \times 1$ 小矩形横着或竖着去覆盖 $2 \times 8$ 的大矩形，覆盖方法有多少种？
- 编写代码求用 $2 \times 1$ 小矩形横着或竖着去覆盖 $2 \times n$ 的大矩形。输出总共有多少种覆盖方法。

**Tips:**考虑递归思想分析问题



## 2.3递归算法设计示例

### ——集合的全排列问题

设  $R = \{r_1, r_2, \dots, r_n\}$  是要进行排列的  $n$  个元素，显然一共有  $n!$  种排列。

令  $R_i = R - \{r_i\}$ 。集合  $X$  中元素的全排列记为  $perm(X)$ ，则  $(r_i)perm(X)$  表示在全排列  $perm(X)$  的每一个排列前加上前缀  $r_i$  得到的排列。

$R$  的全排列可归纳定义如下：

当  $n=1$  时， $perm(R) = (r)$ ，其中  $r$  是集合  $R$  中唯一的元素；

当  $n>1$  时， $perm(R)$  由  $(r_1)perm(R_1)$ ， $(r_2)perm(R_2)$ ， $\dots$ ， $(r_n)perm(R_n)$  构成。

依此递归定义，可设计产生  $perm(R)$  的递归算法

求 $r_1-r_n$ 的全排列Perm (R)



求除 $r_1$ 外剩余元素的全排列Perm (R1)



求除 $r_2$ 外剩余元素的全排列Perm (R2)



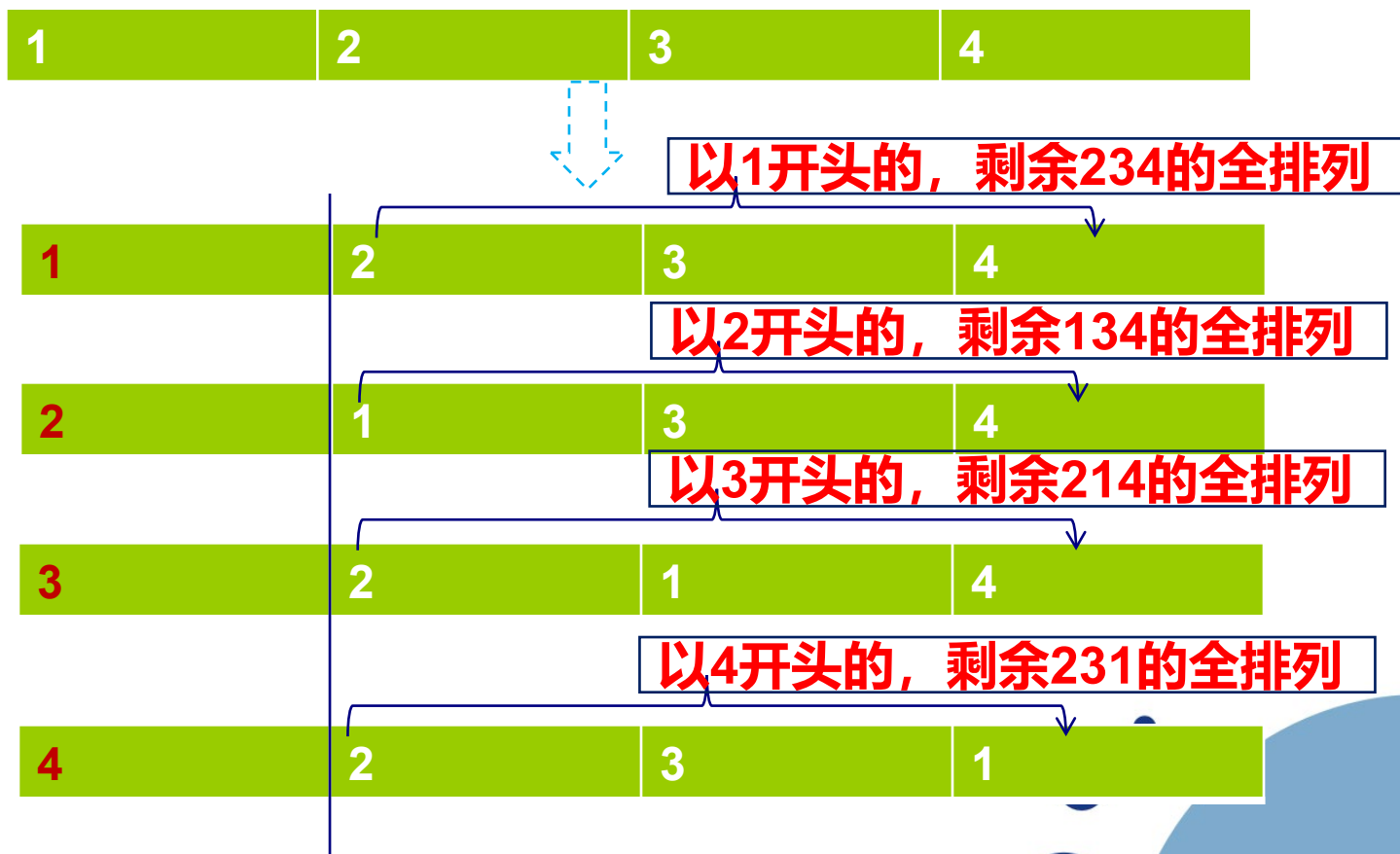
求除 $r_3$ 外剩余元素的全排列Perm (R3)



求除 $r_n$ 外剩余元素的全排列Perm (Rn)



例：求1234的全排列



循环完成

将求取1234（4个元素）全排列的问题转换为：分别取1234交换到第一个位置作为“开头”，求取剩余元素（3个）的全排列问题（递归体）  
当只有一个元素的时候其全排列就是该元素本身（递归出口）



例：求**1234**的全排列

r1=1的排列	r2=2的排列	r3=3的排列	r4=4的排列
1 2 3 4 1 2 4 3 1 3 2 4 1 3 4 2 1 4 3 2 1 4 2 3	2 1 3 4 2 1 4 3 2 3 1 4 2 3 4 1 2 4 3 1 2 4 1 3	3 2 1 4 3 2 4 1 3 1 2 4 3 1 4 2 3 4 1 2 3 4 2 1	4 2 3 1 4 2 1 3 4 3 2 1 4 3 1 2 4 1 3 2 4 1 2 3

# 全排列问题的递归算法

//产生从元素 $k \sim m$ 的全排列，作为前 $k-1$ 个元素的后缀。初始调用时  
 $k=0, m=n-1$

```
void Perm(int list[], int k, int m)
```

```
{
```

```
    if(k==m)           //构成了一次全排列，输出结果
```

```
    {
```

```
        for(int i=0;i<=m;i++)
```

```
            cout<<list[i]<<" ";
```

```
        cout<<endl;
```

```
    }
```

```
    else
```

```
        //在数组list中，产生从元素 $k \sim m$ 的全排列
```

```
        for(int j=k;j<=m;j++)
```

```
        {
```

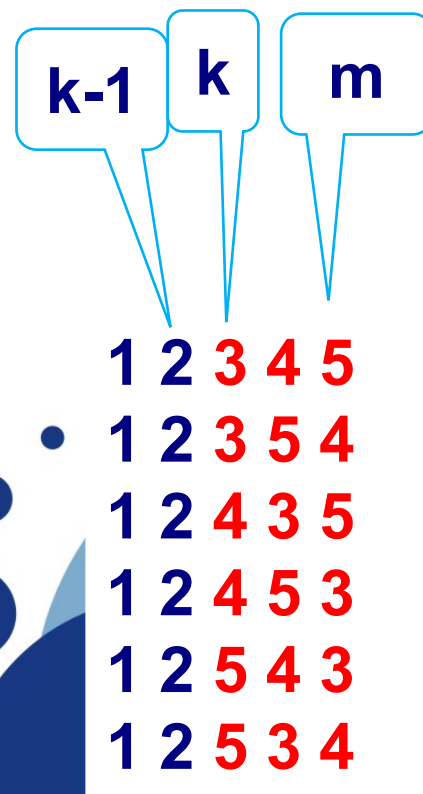
```
            swap(list[k],list[j]);
```

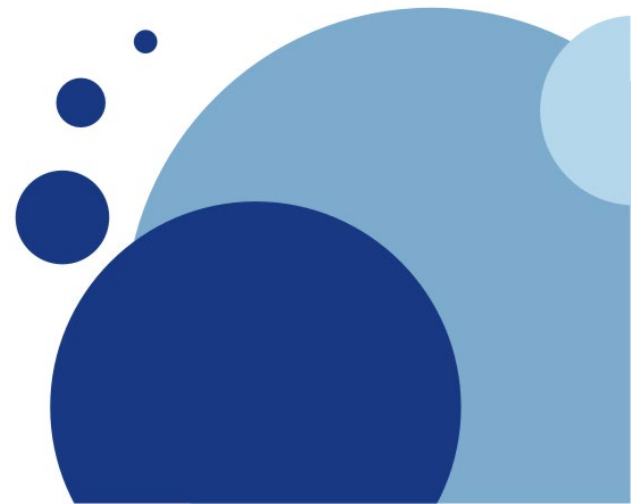
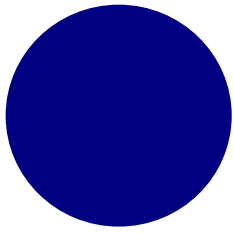
```
            Perm(list,k+1,m);
```

```
            swap(list[k],list[j]);
```

```
        }
```

```
}
```







# 递归算法设计示例

## —— 整数划分问题

- 整数划分问题是算法中的一个经典命题之一。把一个正整数 $n$ 表示成一系列正整数之和：

$$n = n_1 + n_2 + \cdots + n_k \quad (\text{其中, } n_1 \geq n_2 \geq \cdots \geq n_k \geq 1, k \geq 1)$$

- 正整数 $n$ 的这种表示称为正整数 $n$ 的划分。正整数 $n$ 的不同划分个数称为正整数 $n$ 的划分数，记作  $p(n)$ 。
  - 正整数6有如下11种不同的划分，所以  $p(6) = 11$ 。

6

5+1

4+2, 4+1+1

3+3, 3+2+1, 3+1+1+1

2+2+2, 2+2+1+1, 2+1+1+1+1

1+1+1+1+1+1

## 2.3 递归算法设计示例

### —— 整数划分问题

如果  $\{n_1, n_2, \dots, n_i\}$  中的最大加数  $s$  不超过  $m$ , 即  $s = \max(n_1, n_2, \dots, n_i) \leq m$ , 则称它属于  $n$  的一个  $m$  划分。我们记  $n$  的  $m$  划分的个数为  $f(n, m)$ 。该问题就转化为求  $n$  的所有划分个数  $f(n, n)$ 。我们可以建立  $f(n, m)$  的递归关系:

1、 $f(1, m) = 1, m \geq 1$

当  $n=1$  时, 不论  $m$  的值为多少 ( $m > 0$ ), 只有一种划分即 1 个 1。

2、 $f(n, 1) = 1, n \geq 1$

当  $m=1$  时, 不论  $n$  的值为多少 ( $n > 0$ ), 只有一种划分即  $n$  个 1:

$$n = \overbrace{1+1+\dots+1}^n$$

3、 $f(n, m) = f(n, n), m \geq n$

最大加数  $s$  实际上不能超过  $n$ 。例如,  $f(3, 5) = f(3, 3)$ 。

4、 $f(n, n) = 1 + f(n, n-1)$

正整数  $n$  的划分是由  $s=n$  的划分和  $s \leq n-1$  的划分构成。例如,  $f(6, 6) = 1 + f(6, 5)$ 。

5、 $f(n, m) = f(n, m-1) + f(n-m, m), n > m > 1$

正整数  $n$  的最大加数  $s$  不大于  $m$  的划分, 是由  $s=m$  的划分和  $s \leq m-1$  的划分组成。

## 2.3递归算法设计示例

### —— 整数划分问题

正整数 $n$ 的划分数 $p(n)=f(n,n)$

$$f(n, m) = \begin{cases} 1 & n = 1, m = 1 \\ f(n, n) & n < m \\ 1 + f(n, n - 1) & n = m \\ f(n, m - 1) + f(n - m, m) & n > m > 1 \end{cases}$$

$$f(6, 4) = f(6, 3) + f(2, 4) = f(6, 3) + f(2, 2)$$

$f(6,4) = 9$	$f(6,3) = 7$	$f(2,2) = 2$
4+2, 4+1+1 3+3, 3+2+1, 3+1+1+1 2+2+2, 2+2+1+1, 2+1+1+1+1 1+1+1+1+1+1	3+3, 3+2+1, 3+1+1+1 2+2+2, 2+2+1+1, 2+1+1+1+1 1+1+1+1+1+1	4+2, 4+1+1 (实际上是2的划分)

## 2.3递归算法设计示例

### —— 整数划分问题

正整数 $n$ 的划分数 $p(n)=f(n,n)$

$$f(n, m) = \begin{cases} 1 & n = 1, m = 1 \\ f(n, n) & n < m \\ 1 + f(n, n-1) & n = m \\ f(n, m-1) + f(n-m, m) & n > m > 1 \end{cases}$$

#### 算法3.4 正整数 $n$ 的划分算法

```
int split(int n,int m)
```

```
{
```

```
    if(n==1||m==1) return 1;
```

```
    else if (n<m) return split(n,n);
```

```
    else if(n==m) return split(n,n-1)+1;
```

```
    else return split(n,m-1)+split(n-m,m);
```

```
}
```

## 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\* 递归算法转化非递归算法

### 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。

## 2.4\* 递归算法转化非递归算法

### 2.4.1 用循环结构替代递归过程

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);
}
```

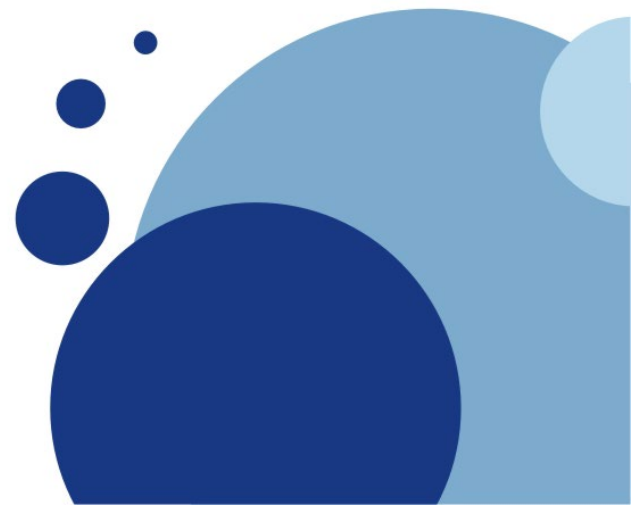


## 2.4\* 递归算法转化非递归算法

### 2.4.1 用循环结构替代递归过程

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。





## 2.4\* 递归算法转化非递归算法

### 2.4.1 用循环结构替代递归过程

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\* 递归算法转化非递归算法

### 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 $f(s)$ 。

## 2.5递归算法分析

当一个算法包含对自身的递归调用过程时，该算法的运行时间复杂度可用递归方程进行描述，求解该递归方程，可得到对该算法时间复杂度的函数度量。

递归方程的求解一般可采用如下方法：

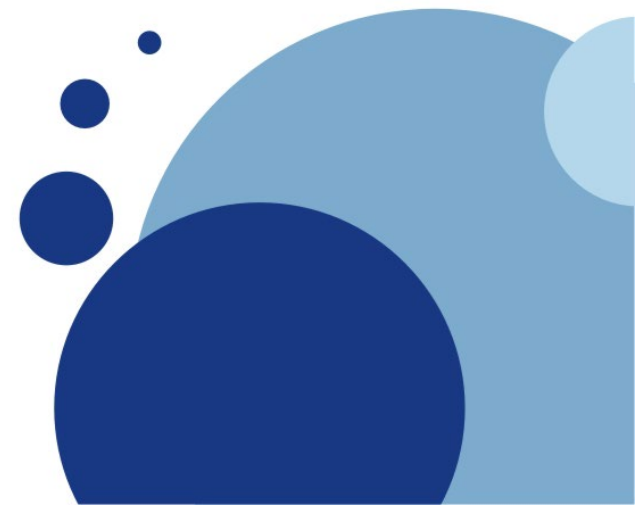
- (1) 替换法
- (2) 用特征方程求解递归方程
- (3) 递归树法
- (4) 主方法



## 2.5递归算法分析

### 1. 替换方法

替换方法的最简单方式为：根据递归规律，将递归公式通过方程展开、反复代换子问题的规模变量，通过多项式整理，如此类推，从而得到递归方程的解。





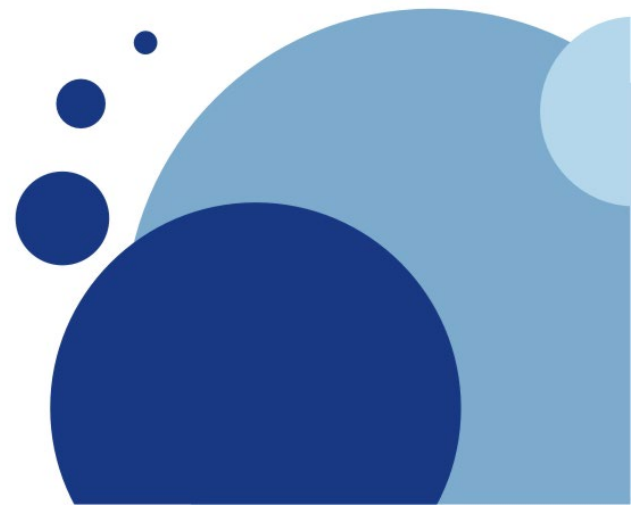
## 2.5递归算法分析

### 1. 替换方法

例:汉诺塔算法（见例2.5）的时间复杂度分析。

假设汉诺塔算法的时间复杂度为 $T(n)$ ，例2.5的算法递归方程为：

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



## 2.5 递归算法分析

### 1. 替换方法

利用替换法求解该方程：

$$T(n) = 2T(n-1) + 1$$

$$= 2(2T(n-2) + 1) + 1$$

$$= 2^2 T(n-2) + 2 + 1$$

$$= 2^2 (2T(n-3) + 1) + 2 + 1$$

.....

$$= 2^{k-1} (2T(n-k) + 1) + 2^{k-2} + \cdots + 2 + 1$$

$$= 2^{n-2} (2T(1) + 1) + 2^{n-2} + \cdots + 2 + 1$$

$$= 2^{n-1} + \cdots + 2 + 1$$

$$= 2^n - 1$$

得到该算法的时间复杂度  $T(n) = O(2^n)$

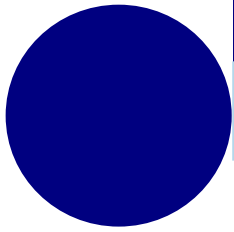
## 2.5递归算法分析

### 1. 替换方法

例2.7 2-路归并排序的递归算法分析。

假设初始序列含有 $n$ 个记录，首先将这 $n$ 个记录看成 $n$ 个有序的子序列，每个子序列的长度为1，然后两两归并，得到  $\lceil n/2 \rceil$  个长度为2（ $n$ 为奇数时，最后一个序列的长度为1）的有序子序列；在此基础上，再对长度为2的有序子序列进行两两归并，得到若干个长度为4的有序子序列；如此重复，直至得到一个长度为 $n$ 的有序序列为止。

这种方法被称作2-路归并排序。



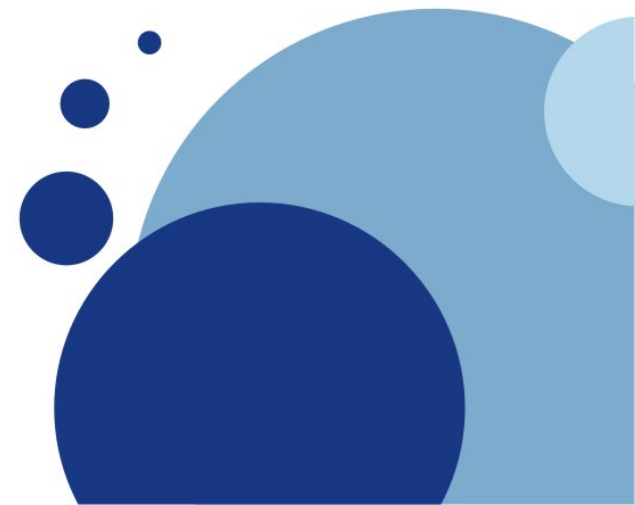
## 2.5递归算法分析

### 1. 替换方法

#### 两有序子序列合并为一个有序序列

2-路归并排序法的基本操作是将待排序列中相邻的两个有序子序列合并成一个有序序列, 算法如下:

```
void Merge ( RecordType r1[], int low, int mid, int high, RecordType SortrR2[])
/* 已知r1[low..mid]和r1[mid+1..high]分别按关键字有序排列, 将它们合并成一个有序
序列, 存放在r2[low..high] */
{
    i=low; j=mid+1; k=low;
    while ( (i<=mid)&&(j<=high) )
        {if ( r1[i].key<=r1[j].key )
            {SortrR2[k]=r1[i]; i++; k++; }
          else
            {SortrR2[k]=r1[j]; j++; k++; }
        }
    while( i<=mid )
        {SortrR2[k]=r1[i];k++;i++;}
    while( j<=high);
        { SortrR2[k]=r1[j];k++;j++;}
} /* Merge */
```





## 2.5递归算法分析

### 1. 替换方法

在合并过程中，两个有序的子表被遍历了一遍，表中的每一项均被复制了一次。因此，合并的代价与两个有序子表的长度之和成正比，该算法的时间复杂度为 $O(n)$ 。

2-□ □ □ □ □ □ □ □ □ □ □ □ □

□ □ □ □ □ □

□ **r1[ ]** □ □ □ □ □ □ □ □ □ □ □ □ □ □ **r3[ ]** □ □ □ □ □ □ □ □ □ □ □

□ □ □

□ □ **r1[ ]** □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ **r2[ ]** □ □ □ □ □ □ □ □

□ □ **r1[ ]** □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ **r2[ ]** □ □ □ □ □ □ □ □

□ □ **r2[ ]** □ □ □ □ □ □ □ □ □ □ □ □ □ □ **r3[ ]** □ □

## 2.5递归算法分析

### 1. 替换方法

2-路归并排序的完整算法如下：

```
void MSort (RecordType r1[], int low, int high, RecordType r3[])
/* r1[ low .... high ]排序后放在r3[ low .... high ]中， r2[ low .... high ]为辅助空间 */
{ RecordType *r2;
    r2 = ( RecordType * ) malloc(sizeof ( RecordType ) * ( high - low + 1 ));
    if ( low == high ) r3[ low ] = r1[ low ];
    else{
        mid = ( low + high ) / 2;
        MSort ( r1, low, mid, r2);
        MSort ( r1, mid+1, high, r2);
        Merge ( r2, low, mid, high, r3);
    }
    free(r2);
} /* MSort */
```

## 2.5递归算法分析

### 1. 替换方法

二路归并排序算法的递归方程为：

$$T(n) = \begin{cases} C_1, & n = 1, \quad \text{二次归并} \\ 2T\left(\frac{n}{2}\right) + C_2n, & n > 1 \end{cases}$$

当 $n > 1$ 时，利用替换法，可得：

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + C_2n \\ &= 2\left[2T\left(\frac{n}{2^2}\right) + C_2\left(\frac{n}{2}\right)\right] + C_2n \\ &= 2^2 T\left(\frac{n}{2^2}\right) + 2C_2n \\ &= 2^2 \left(2T\left(\frac{n}{2^3}\right) + C_2\left(\frac{n}{2^2}\right)\right) + 2C_2n \\ &= 2^3 \left(\frac{n}{2^3}\right) + 3C_2n \\ &= 2^k T\left(\frac{n}{2^k}\right) + kC_2n \end{aligned}$$

## 2.5递归算法分析

### 1. 替换方法

取  $n = 2^k$  则  $\forall n \quad 2^i \leq n \leq 2^{i+1} \quad T(n) = nC_1 + C_2 n \cdot \log_2 n$

(当  $n$  为奇数时, 即  $n = 2^k$  可用  $T(\frac{n+1}{2}) + T(\frac{n-1}{2})$

替代  $2T(\frac{n}{2})$  )

从而,  $T(n) = 2^k T(\frac{n}{2^k}) + kC_2 n = O(n \log_2 n)$

即二次归并排序的算法时间复杂度为

$$T(n) = O(n \log_2 n)$$

可将上述递归方程推广至一般形式, 可记为:

$$\begin{cases} T(n) = aT(\frac{n}{b}) + d(n) & n > 1 \\ T(1) = 1 & n = 1 \end{cases}$$

可将上述递归方程推广至一般形式,  
即不一定是2路归并, 可能是多 (a) 路归并:

$$\begin{cases} T(n) = aT(\frac{n}{b}) + d(n) & n > 1 \\ T(1) = 1 & n = 1 \end{cases}$$

## 2.5 递归算法分析

### 1. 替换方法

对该方程通过替换法求解：

$$\begin{aligned}T(n) &= aT\left(\frac{n}{b}\right) + d(n) \\&= a\left[aT\left(\frac{n}{b^2}\right) + d\left(\frac{n}{b}\right)\right] + d(n) \\&= a^2\left[aT\left(\frac{n}{b^3}\right) + d\left(\frac{n}{b^2}\right)\right] + ad\left[\frac{n}{b}\right] + d(n) \\&= a^3T\left(\frac{n}{b^3}\right) + a^2d\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) \\&\dots \quad \dots \quad \dots \\&= a^iT\left(\frac{n}{b^i}\right) + \sum_{j=0}^{i-1} a^j d\left(\frac{n}{b^j}\right)\end{aligned}$$

由  $n = b^k$  可得到解一般形式为：

$$T(n) = a^k T(1) + \sum_{j=0}^{k-1} a^j d(b^{k-j})$$

$$T(n) = a^k T\left(\frac{n}{b^k}\right) + \sum_{j=0}^{k-1} a^j d\left(\frac{n}{b^j}\right)$$

**一般设**  $n = b^k$ , 则  $k = \log_b n$  , 可得到解一般形式为:

$$T(n) = a^k T(1) + \sum_{j=0}^{k-1} a^j d(b^{k-j})$$

$$= a^{\log_b n} T(1) + \sum_{j=0}^{\log_b n - 1} a^j d(b^{\log_b n - j})$$

$$T(n) = 2T\left(\frac{n}{2}\right) + C_2 n$$

$$= 2[2T\left(\frac{n}{2^2}\right) + C_2\left(\frac{n}{2}\right)] + C_2 n$$

$$= 2^2 T\left(\frac{n}{2^2}\right) + 2C_2 n$$

$$= 2^2 (2T\left(\frac{n}{2^3}\right) + C_2\left(\frac{n}{2^2}\right)) + 2C_2 n$$

$$= 2^3 \left(\frac{n}{2^3}\right) + 3C_2 n$$

$$= 2^k T\left(\frac{n}{2^k}\right) + kC_2 n$$

当 $d(n)=c$ (常数)时, 有:

$$T(n) = a^k T(1) + \sum_{j=0}^{k-1} a^j d(b^{k-j})$$

$$= a^{\log_b n} T(1) + \sum_{j=0}^{\log_b n - 1} a^j d(b^{\log_b n - j})$$

$$T(n) = a^{\log_b n} T(1) + c \sum_{j=0}^{\log_b n} a^j = O(n^{\log_b a})$$



## 2.5递归算法分析

### 1. 替换方法

当 $d(n) = cn$ 时，有：

$$\begin{aligned}T(n) &= a^k T(1) + \sum_{j=0}^{k-1} a^j d(b^{k-j}) \\&= a^k T(1) + \sum_{j=0}^{k-1} a^j (cn / b^j) \\&= a^k T(1) + cn \sum_{j=0}^{\log_b n - 1} (a / b)^j\end{aligned}$$

即该递归方程的解为：

其中

$$r = \frac{a}{b}$$

$$T(n) = a^{\log_b n} T(1) + cn \sum_{j=0}^{\log_b n - 1} r^j$$

## 2.5 递归算法分析

### 1. 替换方法

推论

:

$$T(n) = \begin{cases} O(n), & a < b \\ O(n \log_b n), & a = b \\ O(n \log_b n), & a > b \end{cases}$$

$$T(n) = a^{\log_b n} T(1) + cn \sum_{j=0}^{\log_b n - 1} r^j$$

证明:

①当  $a < b$  时,  $r < 1$ ,  $\sum_{i=0}^{\infty} r^i$  收敛,  $cn \sum_{i=0}^{k-1} r^i = O(n)$

$$T(n) = n^{\log_b a} + O(n) = O(n)$$

$$a^{\log_b n} = a^{\frac{\log_a n}{\log_a b}} = a^{\log_a n \cdot \log_b a} = n^{\log_b a}$$

**推论**

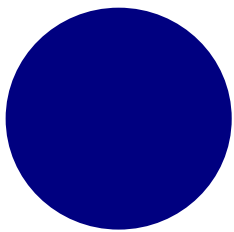
:

$$T(n) = \begin{cases} O(n), a < b \\ O(n \log_b n), a = b \\ O(n \log_b n), a > b \end{cases}$$

$$T(n) = a^{\log_b n} T(1) + cn \sum_{j=0}^{\log_b n - 1} r^j$$

②当  $a = b$  时, 有  $r = 1, cn \sum_{j=0}^{\log_b n - 1} r^j = cn \log_b n$

所以  $T(n) = n^{\log_b a} + cn \log_b n = O(n \log_b n)$



$$T(n) = a^{\log_b n} T(1) + cn \sum_{j=0}^{\log_b n - 1} r^j$$

③当  $a > b$  时, 则  $T(n) = O(n^{\log_b a})$

$$cn \sum_{j=0}^{\log_b n - 1} r^j = cn \frac{(a/b)^k}{a/b - 1} = c \frac{a^k}{a/b - 1} = O(a^k) = O(n^{\log_b a})$$

$n = b^k$

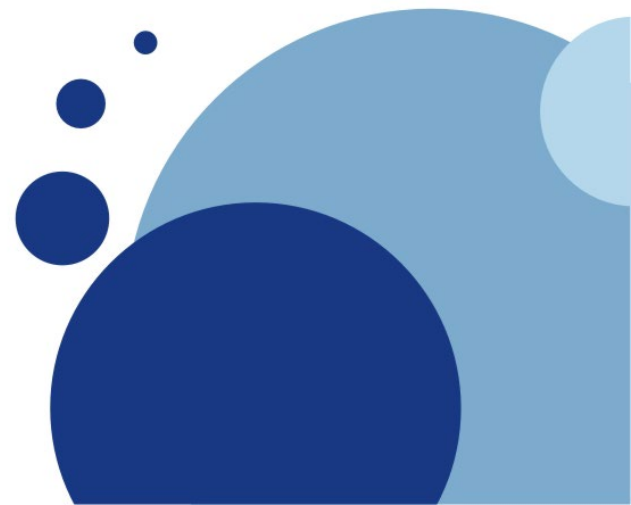
$$= cb^k \frac{1 - (\frac{a}{b})^k}{1 - \frac{a}{b}} = c \frac{b^k - a^k}{1 - \frac{a}{b}} = c \frac{a^k - b^k}{\frac{a}{b} - 1}$$

$$T(n) = a^{\log_b n} T(1) + O(a^k) = n^{\log_b a} + O(n^{\log_b a}) = O(n^{\log_b a})$$

替换法?

$$T(n) = T(n-1) + T(n-2) + 1$$

真让人头大



## 2.5递归算法分析

### 2. 特征方程解递归方程

- K阶常系数线性齐次递归方程
- K阶常系数线性非齐次递归方程



# K阶常系数线性齐次递归方程

K阶常系数线性齐次递归方程形如：

$$\begin{cases} f(n) = a_1 f(n-1) + a_2 f(n-2) + \dots + a_k f(n-k) \\ f(i) = b_i \end{cases} \quad 0 \leq i \leq k-1$$

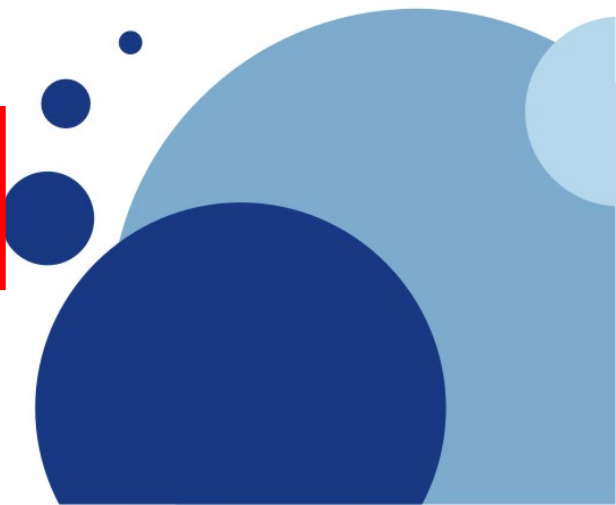
其中， $b_i$ 为常数，第2项为方程初始条件。

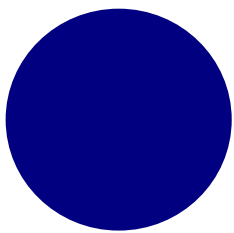
在上式中，用 $x^n$ 取代 $f(n)$ ，有：

$$x^n = a_1 x^{n-1} + a_2 x^{n-2} + \dots + a_k x^{n-k}$$

两边分别除以 $x^{n-k}$ ，得：

$$x^k = a_1 x^{k-1} + a_2 x^{k-2} + \dots + a_k$$





特征方程如下：

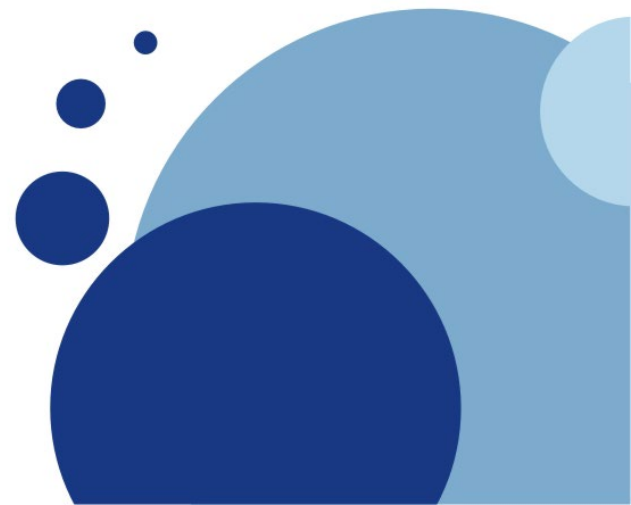
$$x^k - a_1 x^{k-1} - a_2 x^{k-2} - \dots - a_k = 0$$

解题原理：

- 1) 求解上述特征方程的根，得到递归方程的通解
- 2) 利用递归方程初始条件，确定通解中待定系数，得到递归方程的解

考虑2种情况：

- 1) 特征方程的k个根不相同
- 2) 特征方程有相重的根





## 2.5递归算法分析

### 2.特征方程解递归方程

特征方程的k个根不相同:

假设:  $q_1, q_2, \dots, q_k$  是k个不同的根, 则递归方程的通解为

$$f(n) = c_1 q_1^n + c_2 q_2^n + \dots + c_k q_k^n$$

## 2.5递归算法分析

### 2.特征方程解递归方程

特征方程的k个根有重根:

假设: r个重根 $q_i, q_{i+1}, \dots, q_{i+r-1}$ , 则递归方程的通解为

$$f(n) = c_1 q_1^n + \dots + c_{i-1} q_{i-1}^n + (c_i + c_{i+1}n + \dots + c_{i+r-1}n^{r-1})q_i^n \\ + \dots + c_k q^k$$

## 2.5递归算法分析

### 2. 特征方程解递归方程

前面2种情况下的 $c_1, c_2, \dots, c_k$ 均为待定系数；  
将初始条件代入，建立联立方程，确定各个系数具体值，  
得到通解 $f(n)$ 。

例：3阶常系数线性齐次递归方程如下

$$f(n) = 6f(n-1) - 11f(n-2) + 6f(n-3)$$

$$f(0) = 0$$

$$f(1) = 2$$

$$f(2) = 10$$

根据初始条件  
(已知)，确定  
通解中的系数。

解：特征方程为

$$x^3 - 6x^2 + 11x - 6 = 0$$

得到特征根，  
构造通解

## 2.5递归算法分析

### 2.特征方程解递归方程

改写方程为：

$$x^3 - 3x^2 - 3x^2 + 9x + 2x - 6 = 0$$

因式分解：

$$(x-1)(x-2)(x-3)=0$$

得到特征根：

$$q_1=1, q_2=2, q_3=3$$

递归方程的通解为：

$$\begin{aligned} f(n) &= c_1 q_1^n + c_2 q_2^n + c_3 q_3^n \\ &= c_1 + c_2 2^n + c_3 3^n \end{aligned}$$

## 2.5递归算法分析

### 2.特征方程解递归方程

由初始条件得：

$$\begin{cases} f(0) = c_1 + c_2 + c_3 = 0 \\ f(1) = c_1 + 2c_2 + 3c_3 = 2 \\ f(2) = c_1 + 4c_2 + 9c_3 = 10 \end{cases}$$

得到：  $c_1=0, c_2=-2, c_3=2$

因此，递归方程的解为：

$$f(n) = 2(3^n - 2^n)$$

## 2.5递归算法分析

### 2.特征方程解递归方程

例：3阶常系数线性齐次递归方程如下

$$\left\{ \begin{array}{l} f(n) = 5f(n-1) - 7f(n-2) + 3f(n-3) \\ f(0) = 0 \\ f(1) = 2 \\ f(2) = 7 \end{array} \right.$$

解：特征方程为

$$x^3 - 5x^2 + 7x - 3 = 0$$

改写为：  $x^3 - 5x^2 + 6x + x - 3 = 0$

因式分解：

$$(x-3)(x^2 - 2x+1)=0$$

$$(x-3)(x-1)(x-1)=0$$

## 2.5 递归算法分析

### 2. 特征方程解递归方程

得到特征根：

$$q_1=1, q_2=1, q_3=3$$

重根

递归方程的通解为：

$$\begin{aligned} f(n) &= (c_1 + c_2 n) q_1^n + c_3 q_3^n \\ &= c_1 + c_2 n + c_3 3^n \end{aligned}$$

代入初始条件：

$$\begin{cases} f(0) = c_1 + c_3 = 1 \\ f(1) = c_1 + c_2 + 3c_3 = 2 \\ f(2) = c_1 + 2c_2 + 9c_3 = 7 \end{cases}$$

## 2.5.2 递归算法分析: 特征方程解递归方程

得到:  $c_1=0, c_2=-1, c_3=1$

因此, 递归方程的解为:

$$\begin{aligned} f(n) &= (c_1 + c_2 n)q_1^n + c_3 q_3^n \\ &= 3^n - n \end{aligned}$$



# 练习1

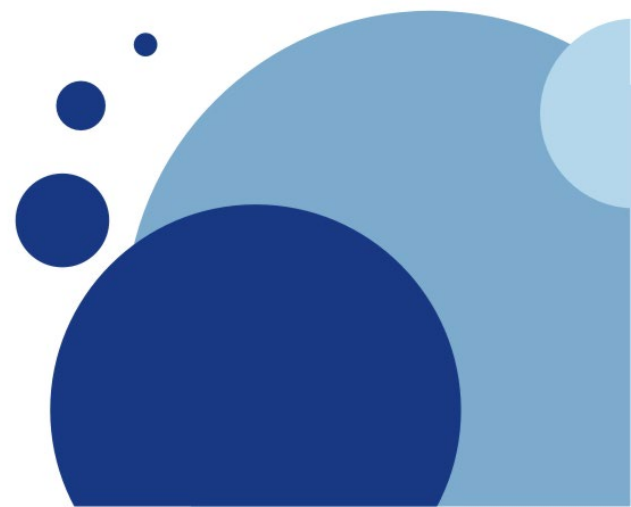
解下列递归方程：

1.  $f(n)=3f(n-1), f(0)=5$

2.  $f(n)=2f(n-1) \quad f(0)=2$

3.  $f(n)=5f(n-1) - 6f(n-2), f(0)=1, f(1)=1$

4.  $f(n)= -6f(n-1) - 9f(n-2), f(0)=3, f(1)=-3$



## 2.5递归算法分析

### 2.特征方程解递归方程

解题原理：

1. 一般没有寻找特解的有效方法
2. 先根据 $g(n)$ 具体形式，确定特解；再将特解代入递归方程，用待定系数法，求解特解的系数

3.  $g(n)$ 分为以下几种情况：

$g(n)$ 是 $n$ 的 $m$ 次的多项式

$g(n)$ 是 $n$ 的指数函数

## 2.5递归算法分析

### 2. 特征方程解递归方程

case1:  $g(n)$  是  $n$  的  $m$  次的多项式  
 $g(n)$  形如:

$$g(n) = b_1 n^m + b_2 n^{m-1} + \dots + b_m n + b_{m+1}$$

其中,  $b_i$  为常数。

此时, 特解  $f^*(n)$  也是  $n$  的  $m$  次多项式, 形如:


$$f^*(n) = A_1 n^m + A_2 n^{m-1} + \dots + A_m n + A_{m+1}$$

各个系数  $A_i$  待定

## 2.5递归算法分析

### 2.特征方程解递归方程

例： 2阶常系数线性非齐次递归方程如下

$$\begin{cases} f(n) = 7f(n-1) - 10f(n-2) + 4n^2 \\ f(0) = 1 \\ f(1) = 2 \end{cases}$$


解： 对应的齐次方程的特征方程为

$$x^2 - 7x + 10 = 0$$

因式分解：  $(x - 2)(x - 5) = 0$

特征根：  $q_1 = 2, q_2 = 5$

对应齐次方程通解：

$$\overline{f(n)} = c_1 2^n + c_2 5^n$$

## 2.5递归算法分析

### 2.特征方程解递归方程

令非齐次递归方程的特解为：

$$f^*(n) = A_1 n^2 + A_2 n + A_3$$

代入原递归方程得：

$$\begin{aligned} & \{A_1 n^2 + A_2 n + A_3\} - 7\{A_1 (n-1)^2 + A_2 (n-1) + A_3\} \\ & + 10\{A_1 (n-2)^2 + A_2 (n-2) + A_3\} \\ & = 4n^2 \end{aligned}$$



化简后得到:

$$4A_1n^2 + (-26A_1 + 4A_2)n + (33A_1 - 13A_2 + 4A_3) \\ = 4n^2 = 4n^2 + 0 * n + 0$$

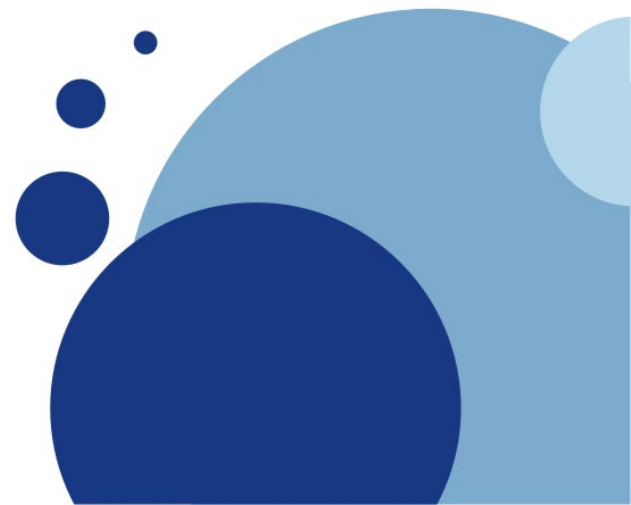
!!!!!!由此得到联立方程:

$$\begin{cases} 4A_1 = 4 \\ -26A_1 + 4A_2 = 0 \\ 33A_1 - 13A_2 + 4A_3 = 0 \end{cases}$$

解得:  $A_1=1$ ,  $A_2=13/2$ ,  $A_3=103/8$

非齐次递归方程的通解为:

$$f(n) = c_1 2^n + c_2 5^n + \boxed{n^2 + 13n/2 + 103/8}$$





初始条件代入有：

$$\begin{cases} f(0) = c_1 + c_2 + \frac{8}{103} = 1 \\ f(1) = 2c_1 + 5c_2 + \frac{163}{8} = 2 \end{cases}$$

得到：  $c_1 = -41/3, c_2 = 43/24$

最后，非齐次递归方程通解为：

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


## 2.5递归算法分析

### 2.特征方程解递归方程

## 二、K阶常系数线性非齐次递归方程

K阶常系数线性非齐次递归方程形如：

其中， $b_i$ 为常数，第2项为方程初始条件。

$$\begin{cases} f(n) = a_1 f(n-1) + a_2 f(n-2) + \dots + a_k f(n-k) + g(n) \\ f(i) = b_i \quad 0 \leq i \leq k-1 \end{cases}$$

它的通解形式为：

$$f(n) = \overline{f(n)} + f^*(n)$$

其中，

1)  $\overline{f(n)}$  为对应齐次递归方程的通解

2)  $f^*(n)$  为原非齐次递归方程的特解



## 2.5递归算法分析

### 2.特征方程解递归方程

case2: $g(n)$ 是 $n$ 的指数函数  
 $g(n)$ 形如:

$$g(n) = (b_1 n^m + b_2 n^{m-1} + \dots + b_m n + b_{m+1}) a^n$$

其中,  $a$ 和 $b_i$ 为常数。

1) 如果 $a$ 不是特征方程的重根, 特解 $f^*(n)$ 形如:

$$f^*(n) = (A_1 n^m + A_2 n^{m-1} + \dots + A_m n + A_{m+1}) a^n$$

各个系数 $A_i$ 待定

## 2.5.2递归算法分析:特征方程解递归方程

2) 如果 $a$ 是特征方程的 $r$ 重特征根, 特解 $f^*(n)$ 形如:

$$f^*(n) = (A_1 n^m + A_2 n^{m-1} + \dots + A_m n + A_{m+1}) n^r a^n$$

各个系数 $A_i$ 待定

## 2.5.2递归算法分析:特征方程解递归方程

例 2阶常系数线性非齐次递归方程如下

$$\begin{cases} f(n) = 7f(n-1) - 12f(n-2) + n2^n \\ f(0) = 1 \\ f(1) = 2 \end{cases}$$

$g(n)$

解: 对应的齐次方程的特征方程为

$$x^2 - 7x + 12 = 0$$

因式分解:  $(x - 3)(x - 4) = 0$

特征根:  $q_1=3, q_2=4$

对应齐次方程通解:

$$f(n) = c_1 3^n + c_2 4^n$$

## 2.5.2递归算法分析:特征方程解递归方程

$a=2$ 不是特征方程的重根, 故令非齐次递归方程的特解为:

$$f^*(n) = (A_1n + A_2)2^n$$

代入原递归方程得:

$$\begin{aligned} & (A_1n^2 + A_2)2^n - 7\{A_1(n-1) + A_2\}2^{n-1} \\ & + 12\{A_1(n-2) + A_2\}2^{n-2} \\ & = n2^n \end{aligned}$$

化简后得到:  $2A_1n + 2A_2 - 10A_1 = 4n$

## 2.5递归算法分析

### 2.特征方程解递归方程

由此得到联立方程：

$$\begin{cases} 2A_1 = 4 \\ 2A_2 - 10A_1 = 0 \end{cases}$$

解得：  $A_1=2$ ,  $A_2=10$

非齐次递归方程的通解为：

$$f(n) = c_1 3^n + c_2 4^n + (2n + 10)2^n$$

## 2.5递归算法分析

### 2.特征方程解递归方程

初始条件代入有：

$$\begin{cases} f(0) = c_1 + c_2 + 10 = 1 \\ f(1) = 3c_1 + 4c_2 + 24 = 2 \end{cases}$$

得到：  $c_1 = -14, c_2 = 5$

最后，非齐次递归方程通解为：

$$\begin{aligned} f(n) &= -14 \times 3^n + 5 \times 4^n + (2n + 10)2^n \\ &= -14 \times 3^n + 5 \times 4^n + (n + 5)2^{n+1} \end{aligned}$$

## 练习2

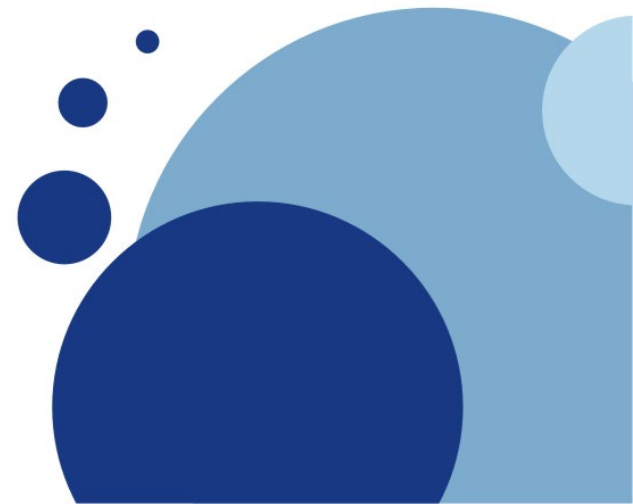
解下列递归方程：

1.  $f(n)=f(n-1) + n^2, f(0)=0$

2.  $f(n)=2f(n-1) +n, f(0)=1$

3.  $f(n)=3f(n-1) + 2^n, f(0)=3$

4.  $f(n)= - 2f(n-1) + 2^n -n^2, f(0)=1$



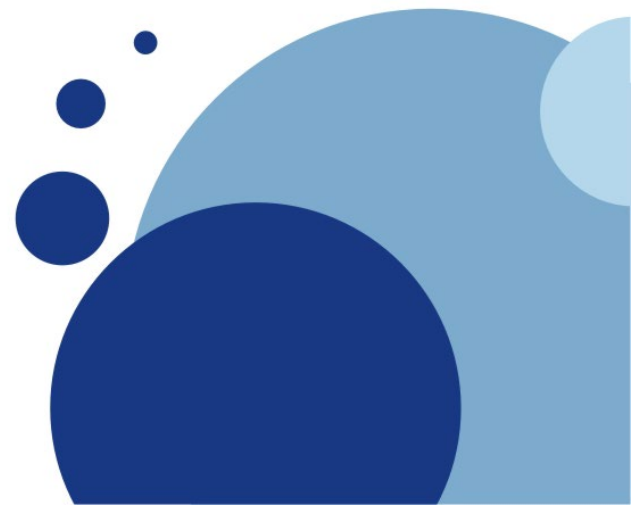


## 2.5递归算法分析

### 3. 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。





## 递归树方法

递归树的方法可以更好地猜测一个问题的解，并用替换方法证明这个猜测。

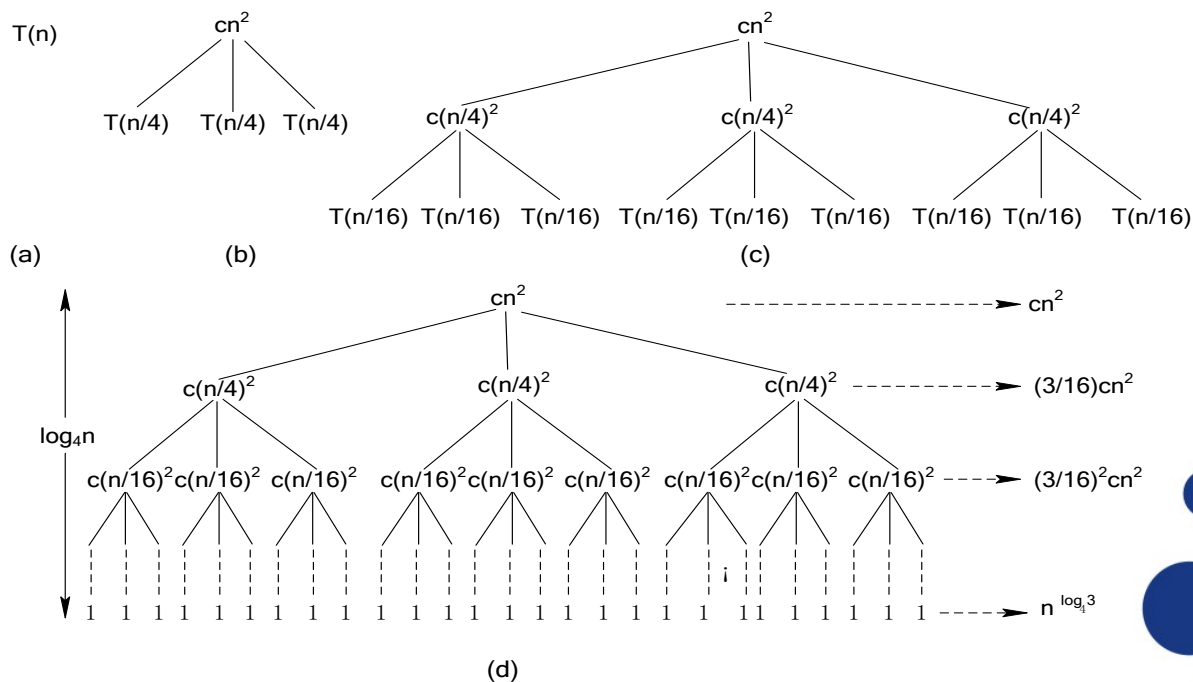
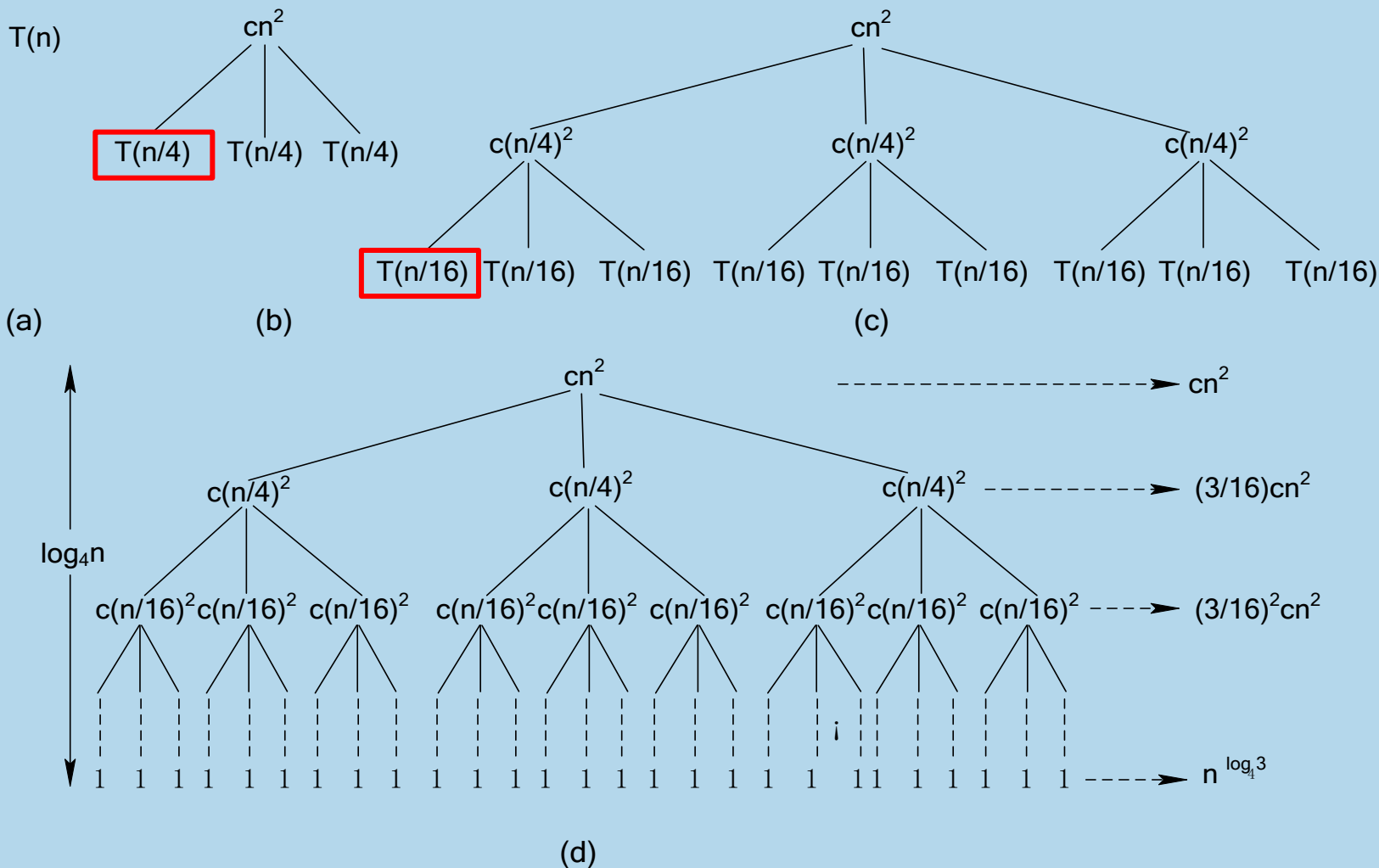


图 2-4 递归树的构造过程

图2-4表明了如何求解方程  $T(n) = 3T(n/4) + cn^2$  构造递归树的方法，其中n假设为4的幂。



## 2.5 递归算法分析

### 3. 递归树方法求解递归方程

深度为 $i$ 的结点，其子问题的规模为 $n/4^i$ ，当 $n/4^i=1$ 时，子问题规模为1，这时位于树的最后一层（即 $i=\log_4 n$ ）。在图2-4的递归树中，层数是从0开始算起的，第一层层数为0，最后一层的层数为 $\log_4 n$ ，共有 $\log_4 n+1$ 层。深度是从0开始算起的，第一层深度为0，最后一层深度为 $\log_4 n$ 。

第 $i$ 层的结点数为 $3^i$ 。（每一层的结点数是上一层结点数的三倍）

层数为  $i (i = 0, 1, \dots, \log_4 n - 1)$  的每个结点的开销为

$$c(n/4^i)^2$$

(每一层子问题规模为上一层的1/4)

## 2.5 递归算法分析

### 3. 递归树方法求解递归方程

第*i*层上结点的总开销为

$$3^i c \left( n / 4^i \right)^2 = (3 / 16)^i c n^2 \quad i = 0, 1, \dots, \log_4 n - 1$$

层数为 $\log_4 n$ 的最后一层有 $3^{\log_4 n} = n^{\log_4 3}$ 个结点, 每个结点的开销为 $\pi(1)$ , 该层总开销 $n^{\log_4 3} T(1)$ 即 $\theta(n^{\log_4 3})$

将所有层的开销相加得到整棵树的开销:

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \dots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \theta(n^{\log_4 3}) \\ &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \theta(n^{\log_4 3}) \\ &\leq \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \theta(n^{\log_4 3}) \\ &= \frac{1}{1 - 3/16} cn^2 + \theta(n^{\log_4 3}) \\ &= \frac{16}{13} cn^2 + \theta(n^{\log_4 3}) \\ &= O(n^2) \end{aligned}$$

## 2.5.3 递归算法分析: 递归树方法求解递归方程

现在利用替换方法证明我们的猜测是正确的。假设这个界限对于 $n/4$ 成立, 即存在某个常数 $d$ ,  $T(n/4) \leq d(n/4)^2$

代入递归方程可得  $T(n) = 3T(n/4) + cn^2$

$$\leq 3d(n/4)^2 + cn^2$$

$$= (3/16)dn^2 + cn^2$$

根据上面式子, 当  $c \leq (13/16)d$  时, 有:

$$T(n) \leq (3/16)dn^2 + (13/16)dn^2 = dn^2$$

## 2.5.4递归算法分析:主方法求解递归方程

**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:  $a=3$ ,  $b=4$ ,  $f(n) = n^2$ 。

## 2.5.4递归算法分析:主方法求解递归方程

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

## 2.5.4 递归算法分析：主方法求解递归方程

应用该定理的过程是，首先把函数  $f(n)$  与函数  $n^{\log_b a}$  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  $T(n) = O(f(n))$ 。



## 2.5.4递归算法分析:主方法方法求解递归方程

【例】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】采用主方法求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,  $n^{\log_b a} = n$ , 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。

