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fun(*n*)求解过程中，直接调用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:

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

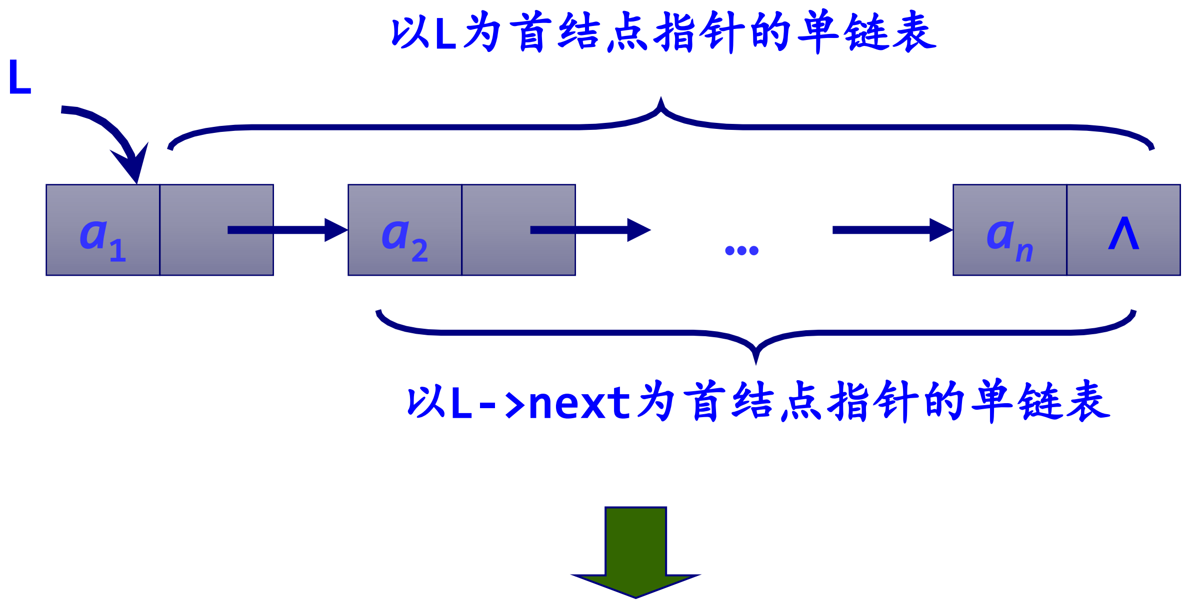
2．数据结构是递归的

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。

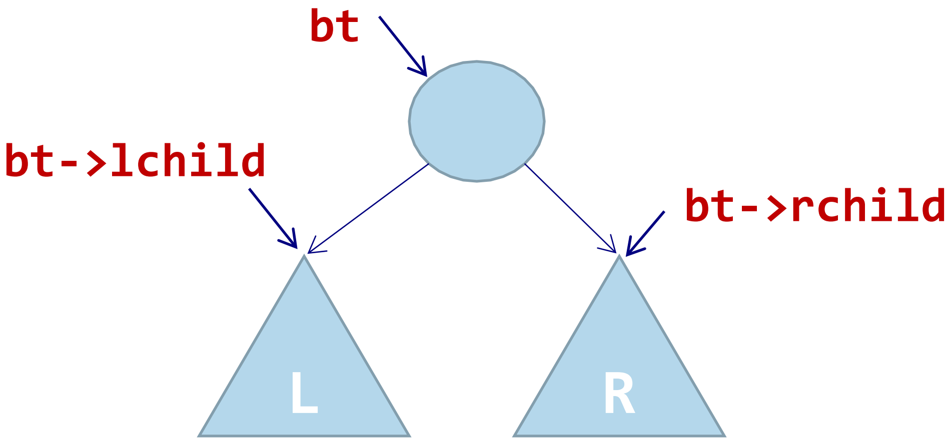
对于递归数据结构，采用递归的方法编写算法既方便又有效。例如，求一个不带头结点的单链表L的所有data域（假设为int型）之和的递归算法如下：

```
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;
} BTNode;                    //二叉链结点类型
```




```
int Sumbt(BTNode *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                        //否则返回左、右子树结点值之和加上根结点值
      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：

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：

```
fun(1)=1                              (1)
fun(n)=n*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fun(n)的值与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：

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

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：

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

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}、…、s_n为递归“小问题”，c_j、c_{j+1}、…、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

```
int fun(int n)
{  if (n==1)  return(1);

   else

       return(fun(n-1)*n);

}
```

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

退栈1次并求fun(4)值

4	6*4=24
5	fun(4)*5

退栈1次并求fun(5)值

5	24*5=120

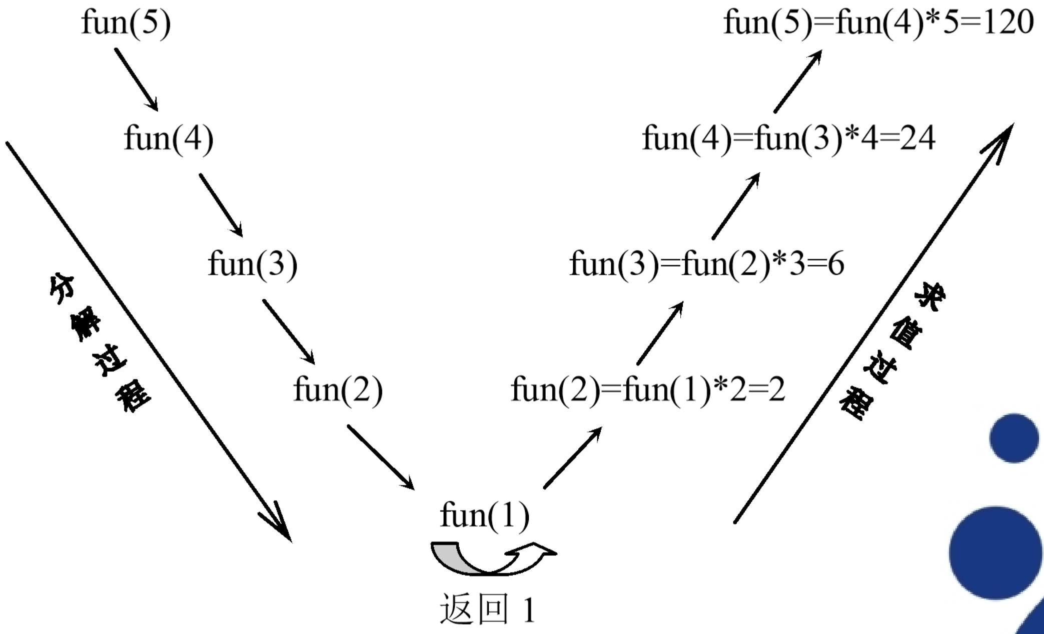
退栈1次并输出120

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。

- 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；
- 第二步的求值过程，即已知“小问题”，计算“大问题”。前面的fun(5)求解过程如下所示。



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。

Important !

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）；

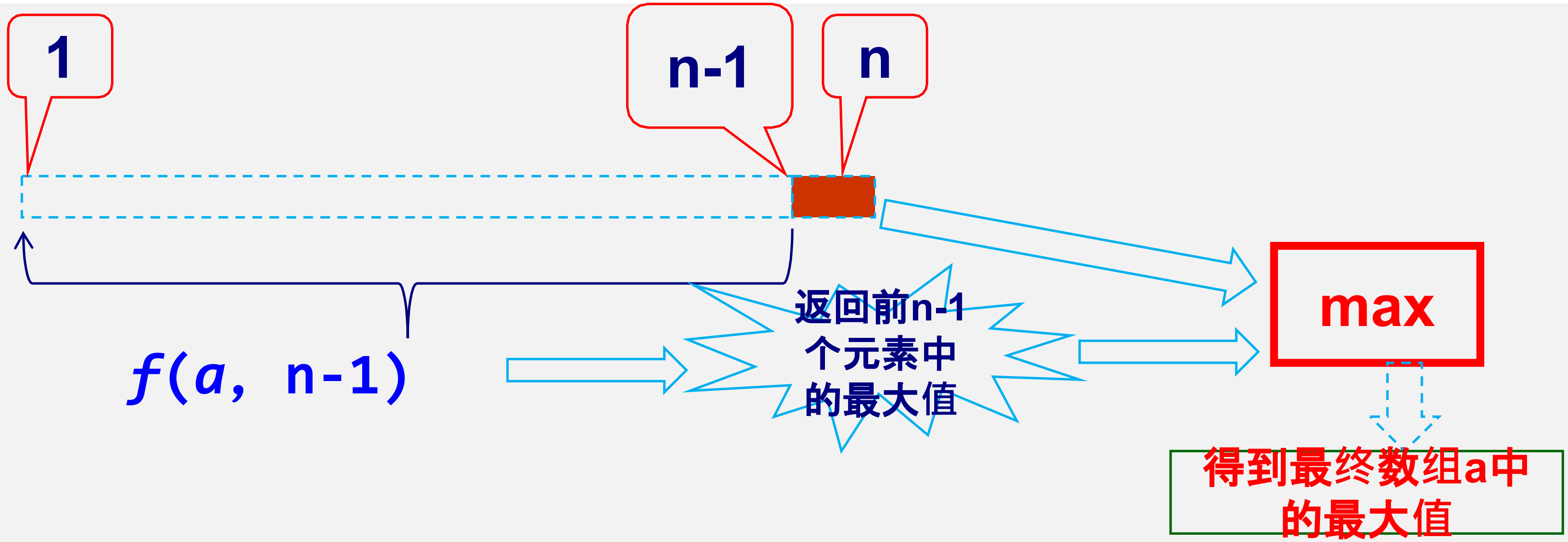
（2）假设$f(s_{n-1})$是可解的，在此基础上确定$f(s_n)$的解，即给出$f(s_n)$与$f(s_{n-1})$之间的关系（与数学归纳法中求证$n=k$时等式成立的过程相似）；

（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$a[1..n]$中的最大元素，则$f(a, n-1)$求解数组a中前$n-1$个元素即$a[1..n-1]$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$f(a, n)=MAX\{f(a, n-1), a[n-1]\}$。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$f(a, 1)=a[1]$。



由此得到递归模型如下：

$f(a, n)=a[1]$　　　　当$n=1$时

$f(a, n)=\mathrm{MAX}\{f(a, n-1), a[n]\}$　　　　当$n>1$时

对应的递归算法如下：

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

设计递归算法求一个数组A[1..n]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[1..mid]和A[mid+1..n],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[1..n]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

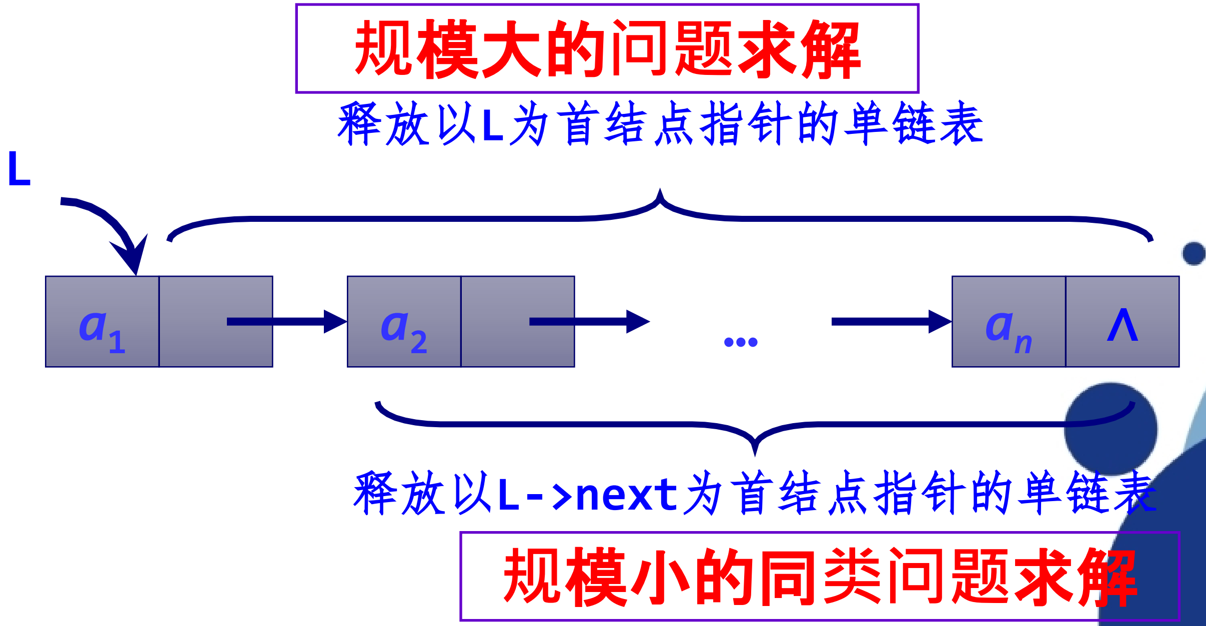
在设计不带头结点的单链表的递归算法时：

- 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，…，a_n}，f(L)的功能是释放a_1～a_n的所有结点，则f(L->next)的功能是释放a_2～a_n的所有结点，前者是“大问题”，后者是“小问题”。

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

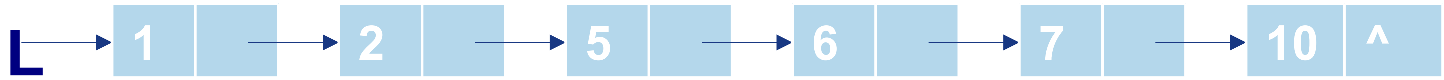


对应的递归模型如下：

f(L) ≡不做任何事件　　　　　　　　当L=NULL时
f(L) ≡ f(L->next); 释放L结点　　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、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。

【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a0相等）

先序序列：a_0 $\underbrace{a_1 \ \dots \ a_k}_{}$ $\underbrace{a_{k+1} \ \dots \ a_{n-1}}_{}$

中序序列：$\underbrace{b_0 \ b_1 \ \dots \ b_{k-1}}_{}$ b_k $\underbrace{b_{k+1} \ \dots \ b_{n-1}}_{}$

左子树先序序列，有k个结点

右子树先序序列，有n-k-1个结点

左子树中序序列，有k个结点

右子树中序序列，有n-k-1个结点

a_0

左子树

先序：a_1 … a_k

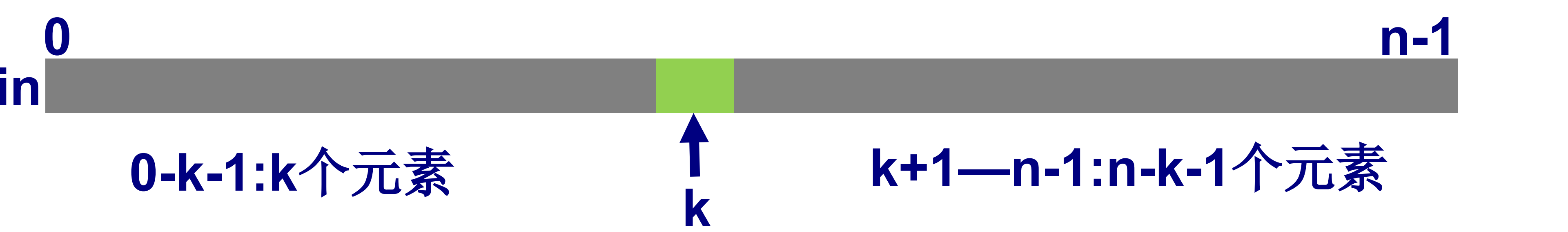
中序：b_0 …b_{k-1}

先序：a_{k+1} … a_{n-1}

中序：b_{k+1} … b_{n-1}

右子树

```c
BTNode *CreateBTree(ElemType pre[],ElemType in[],int n)
//由先序序列pre[0..n-1]和中序序列in[0..n-1]建立二叉链存储结构bt
{   int k;
    if (n<=0) return NULL;
    ElemType root=pre[0];                        //根结点值
    BTNode *bt=(BTNode *)malloc(sizeof(BTNode));
    bt->data=root;
    for (k=0;k<n;k++)     //在in中查找in[k]=root的根结点
       if (in[k]==root)
          break;
    bt->lchild=CreateBTree(pre+1,in,k);//递归创建左子树
    bt->rchild=CreateBTree(pre+k+1,in+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?

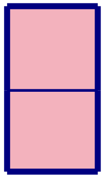
- 拓展
- 若条件改为：一只青蛙一次可以跳上1级台阶，也可以跳上2级，也可以跳上3级，。。。。也可以跳上n级。编写代码求青蛙跳上一个n级的台阶，总共有多少种跳法?
- Tips:考虑递归思想分析问题

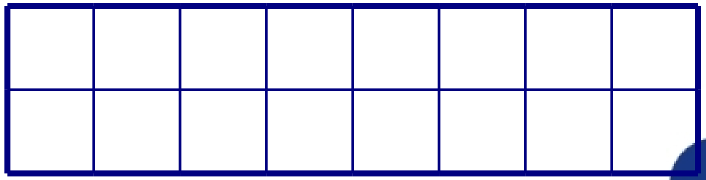
2.3递归算法设计示例

3、瓷砖覆盖问题：用一个2×1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2×1小矩形横着或竖着去覆盖2×8的大矩形，覆盖方法有多少种？
- 编写代码求用2×1小矩形横着或竖着去覆盖2×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 $R=\{r_1, r_2, \ldots, r_n\}$ 是要进行排列的 n 个元素，显然一共有 $n!$ 种排列。

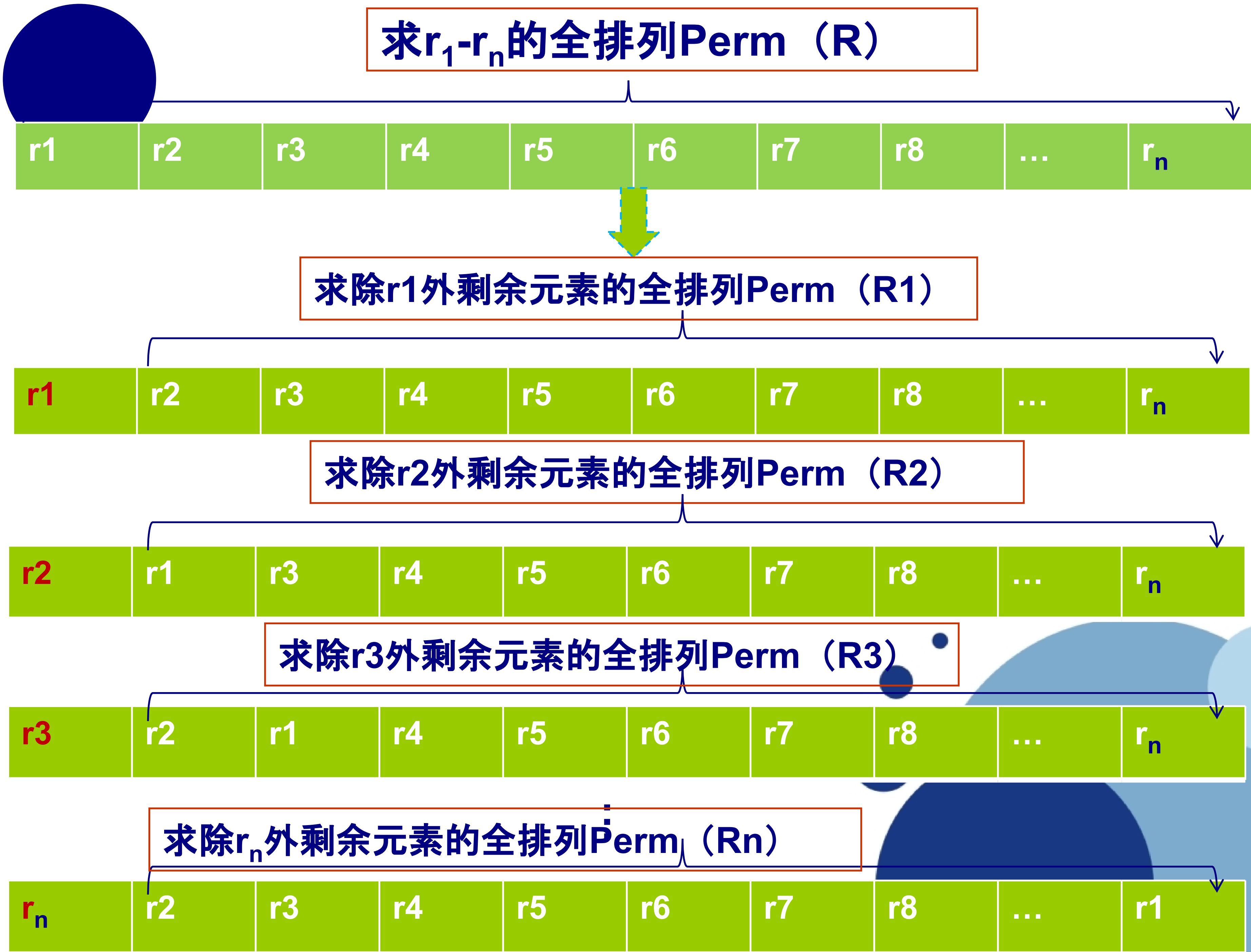
令 $R_i=R-\{r_i\}$。集合 X 中元素的全排列记为 $perm(X)$，则 $(r_i)perm(X)$ 表示在全排列 $perm(X)$ 的每一个排列前加上前缀 r_i 得到的排列。

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 $(r_1)perm(R_1)$，$(r_2)perm(R_2)$，…，$(r_n)perm(R_n)$ 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）
r1 r2 r3 r4 r5 r6 r7 r8 … r_n
求除r1外剩余元素的全排列Perm（R1）
r1 r2 r3 r4 r5 r6 r7 r8 … r_n
求除r2外剩余元素的全排列Perm（R2）
r2 r1 r3 r4 r5 r6 r7 r8 … r_n
求除r3外剩余元素的全排列Perm（R3）
r3 r2 r1 r4 r5 r6 r7 r8 … r_n
求除r_n外剩余元素的全排列Perm（Rn）
r_n r2 r3 r4 r5 r6 r7 r8 … r1

例：求1234的全排列

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～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～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]);
                }
}
```

k-1 k m

1 2 3 4 5

1 2 3 5 4

1 2 4 3 5

1 2 4 5 3

1 2 5 4 3

1 2 5 3 4

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：

  $n = n_1 + n_2 + \cdots + n_k$ （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$\{n_1, n_2, \cdots, n_i\}$中的最大加数 s 不超过m，即$s = \max(n_1, n_2, \cdots, n_i) \leqslant m$，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、$f(1,m)=1$，$m \geqslant 1$

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、$f(n,1)=1$，$n \geqslant 1$

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

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

3、$f(n,m)=f(n,n)$，$m \geqslant n$

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 $s \leqslant n-1$ 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 $s \leqslant m-1$ 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);
}
```

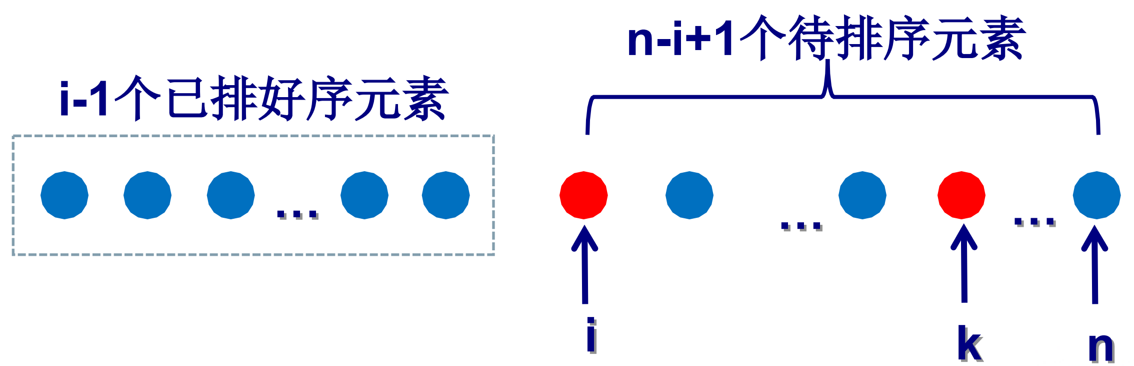
思考题

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，设计基于递归算法进行按元素值递增排序。

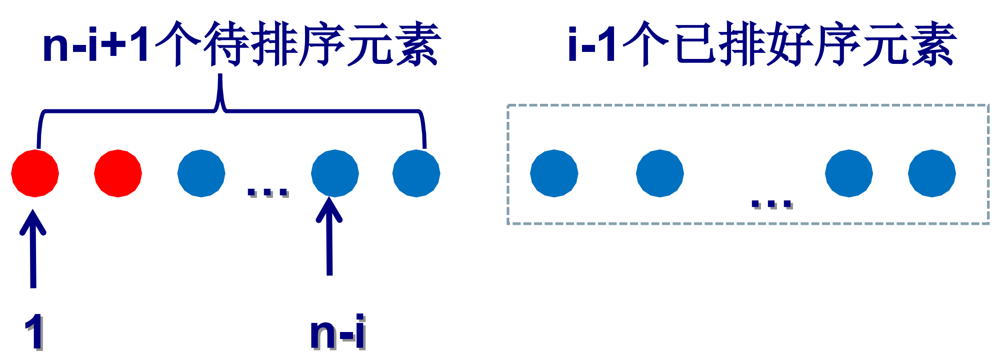
思考题

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，基于递归对其按元素值递增排序。

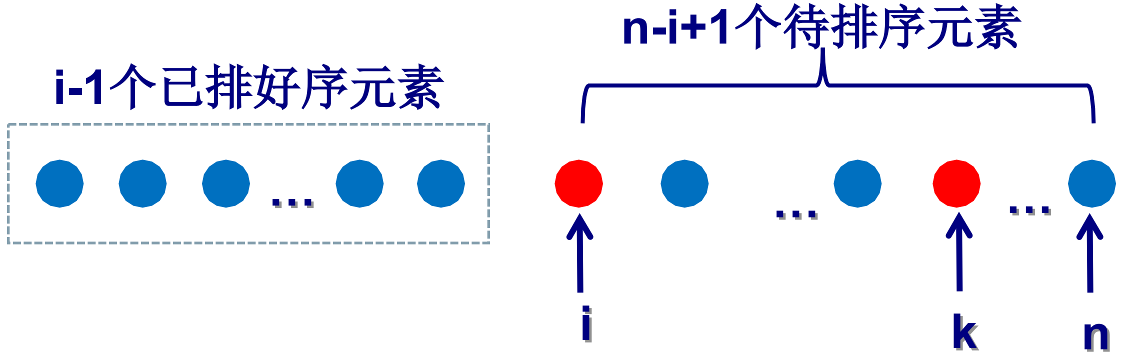
简单选择排序



冒泡排序



思考题 【递归的简单选择排序】

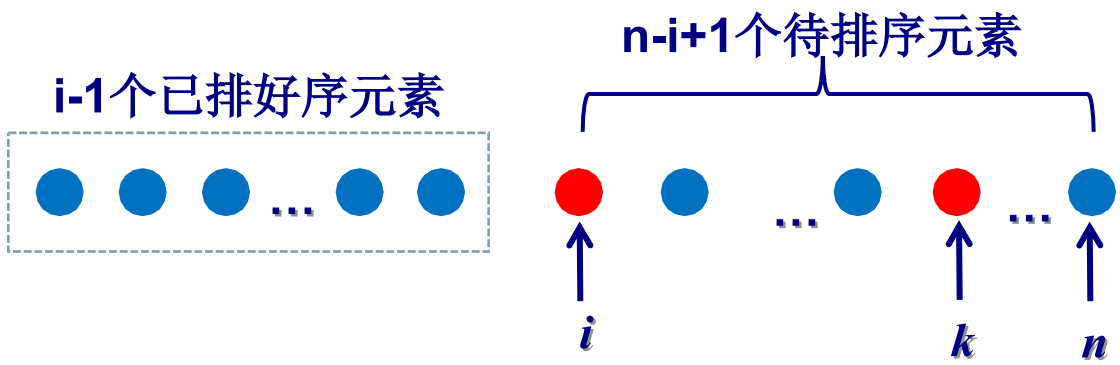


"大问题"：$f(a, i, n)$用于对$a[i..n]$元素序列（共$n-i+1$个元素）进行简单选择排序。

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

当$i=n$时所有元素有序，算法结束。

思考题 【递归的简单选择排序】



$f(a，i，n) \equiv$ 不做任何事情，算法结束　　　当$i=n$

$f(a，i，n) \equiv$ 通过简单比较挑选$a[i..n]$中　　　否则

的最小元素$a[k]$放在$a[i]$处;

然后，继续$f(a，i+1，n)$;

【递归的简单选择排序】

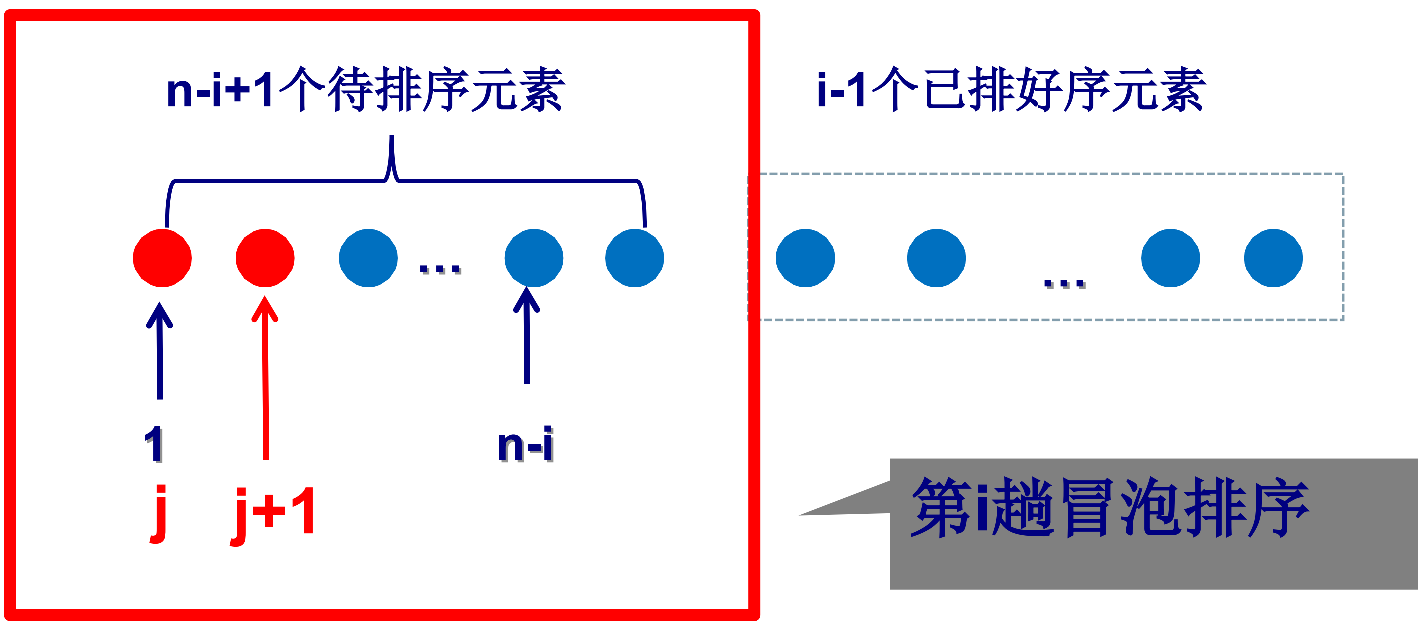
```
void SelectSort(int a[], int i, int n)  //首次调用时：i为1
{   int j, k;
    if (i==n) return;        //满足递归出口条件

    else
    {  k=i;                  //k记录a[i..n]中最小元素的下标
       for (j=i+1;j<=n;j++)  //在a[i..n]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, i+1, n);
    }
}
```

思考题 【递归的冒泡排序】

大问题：设*f*(*a*，*i*，*n*)用于对*a*[1..n-i+1]元素序列（共*n-i+1*个元素）进行冒泡排序（i表示排序的趟数，i=1~n-1）

小问题：*f*(*a*，*i*+1，*n*)用于对*a*[1..n-i]元素序列（共*n-i*个元素）进行冒泡排序。

当*i*=n时，所有元素有序，算法结束。



【递归的冒泡排序】

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

    else
    {   exchange=false;                   //置exchange为false
        for (j=1;j<=n-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, i+1, n);
    }
}
```

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。（例：阶乘）

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。（例：树、图的遍历）

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:

$$
\begin{aligned}
T(n) &= 2T(n-1)+1 \\
&= 2(2T(n-2)+1)+1 \\
&= 2^2T(n-2)+2+1 \\
&= 2^2(2T(n-3)+1)+2+1 \\
&\cdots\cdots \\
&= 2^{k-1}(2T(n-k)+1)+2^{k-2}+\cdots+2+1 \\
&= 2^{n-2}(2T(1)+1)+2^{n-2}+\cdots+2+1 \\
&= 2^{n-1}+\cdots+2+1 \\
&= 2^n-1
\end{aligned}
$$

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

二路归并排序算法的递归方程为：

$$T(n)=\begin{cases}C_1, & n=1, \quad \text{二次归并} \\ 2T\left(\frac{n}{2}\right)+C_2 n, & n>1\end{cases}$$

当n>1时，利用替换法，可得：

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

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

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

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

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

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

2.5递归算法分析

1.替换方法

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

（当n为奇数时，即 $n=2^k-1$ 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：

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

2.5递归算法分析

1.替换方法

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：

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

由 $\boxed{n = b^k}$ 可得到解一般形式为：

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

Chapter

2.5递归算法分析

1.替换方法

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

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

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

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

$$T(n)=2T(\frac{n}{2})+C_2n$$

$$=2[2T(\frac{n}{2^2})+C_2(\frac{n}{2})]+C_2n$$

$$=2^2T(\frac{n}{2^2})+2C_2n$$

$$=2^2(2T(\frac{n}{2^3})+C_2(\frac{n}{2^2}))+2C_2n$$

$$=2^3(\frac{n}{2^3})+3C_2n$$

$$=2^kT(\frac{n}{2^k})+kC_2n$$

2.5递归算法分析

1.替换方法

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

情况一：当 *d(n)=c* (常数) 时，有：

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

C

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

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

$$c\left(a^0 + a^1 + \ldots + a^{\log_b n}\right) = \frac{a . a^{\log_b n} - 1}{a - 1} < a . a^{\log_b n} - 1$$

2.5递归算法分析

1.替换方法

例：求a[1..n]最大值的递归算法，时间耗费的递归方程为：

$$T(n)=\begin{cases}1, & n=1,\\ 2T\left(\frac{n}{2}\right)+1, & n>1\end{cases}$$

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

此时：a=b=2,所以：该算法的时间复杂度为O（$n^{\log_2 2}$）即为O（n）

2.5递归算法分析

1.替换方法

情况二： 当$d(n)=cn$，$n\approx b^k$时，有：

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

$$=a^kT(1)+\sum_{j=0}^{k-1}a^j(cn/b^j)$$

$$=a^kT(1)+cn\sum_{j=0}^{\log_b n-1}(a/b)^j$$

即该递归方程的解为：

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

其中 $\boxed{r=\frac{a}{b}}$

2.5递归算法分析

1.替换方法

推论： 当d(n)=cn时

$$T(n)=\begin{cases} O(n), a<b & ① \\ O(n\log_b n), a=b & ② \\ O(n\log_b n), a>b & ③ \end{cases}$$

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

证明：

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

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

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

2.5递归算法分析

1.替换方法

推论： 当d(n)=cn时

$$T(n)=\begin{cases} O(n), a<b & ① \\ O(n\log_b n), a=b & ② \\ O(n\log_b n), a>b & ③ \end{cases}$$

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

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

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

2.5递归算法分析

1.替换方法

例：二路归并排序算法的递归方程为：

$$T(n)=\begin{cases} C_1, \quad n=1, & \text{二次归并} \\ 2T\left(\frac{n}{2}\right)+C_2 n, \quad n>1 \end{cases}$$

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

二路归并排序中，a=b=2,d(n)=n;T(n)=O（$n\log_2 n$）

2.5递归算法分析

1.替换方法

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

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

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

n=b^k

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

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

2.5递归算法分析

1.替换方法

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

情况一：当$d(n)=c$（常数）时，有： $T(n) = O(n^{\log_b a})$

情况二：当$d(n) = cn$，$n \approx b^k$时，有：

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

1.替换方法

替换法?

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

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：

$$\begin{cases} T(n) = a_1 T(n-1) + a_2 \mathrm{T}(n-2) + \ldots + a_k \mathrm{T}(n-k) \\ \mathrm{T}(i) = b_i \qquad\qquad 0 \le i \le k-1 \end{cases}$$

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T(n), 有：

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

两边分别除以x^{n-k}，得：

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

2.特征方程解递归方程

特征方程如下：

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

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

$$T(n) = c_1 {q_1}^n + c_2 {q_2}^n + \ldots + c_k {q_k}^n$$

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}, …, q_{i+r-1}，则递归方程的通解为

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

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$c_1, c_2, \ldots, c_k$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

$$\begin{cases} T(n) = 6T(n-1) - 11T(n-2) + 6T(n-3) \\ T(0) = 0 \\ T(1) = 2 \\ T(2) = 10 \end{cases}$$

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.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：

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

2.5递归算法分析

2.特征方程解递归方程

由初始条件得：

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

得到： $c_1=0, c_2=-2, c_3=2$

因此，递归方程的解为：

$$\mathrm{T}(n) = 2(3^n - 2^n)$$

2.5递归算法分析

2.特征方程解递归方程

【例】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T(n)：

T(n)=1　　　　　　当n=1或2时

T(n)=T(n-1)+T(n-2)　　　　　　当n>2时

为了简化解，可以引入额外项T(0)=0。其特征方程是$x^2-x-1=0$，求得根为：$r_1=\frac{1+\sqrt{5}}{2}$，$r_2=\frac{1-\sqrt{5}}{2}$

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

2.5递归算法分析

2.特征方程解递归方程

为求c_1和c_2，求解下面两个联立方程：

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

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

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

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

$$\begin{cases} T(n) = 5T(n-1) - 7T(n-2) + 3T(n-3) \\ T(0) = 0 \\ T(1) = 2 \\ T(2) = 7 \end{cases}$$

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：

$\boxed{q_1=1,\ q_2=1,}\ q_3=3$

重根

递归方程的通解为：

$$T(n) = \boxed{(c_1 + c_2 n) q_1{}^n} + c_3 q_3{}^n$$
$$= c_1 + c_2 n + c_3 3^n$$

代入初始条件：

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

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：

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

练习1

解下列递归方程：

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

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

其中，b_i为常数。

此时，特解$f^*(n)$也是n的m次多项式，形如：

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

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

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

g(n)

解： 对应的齐次方程的特征方程为

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

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：

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

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：

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

得到： c_1**=-41/3,** c_2**=43/24**

最后，非齐次递归方程通解为：

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

3.递归树方法求解递归方程

（3）递归树法

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。

3.递归树方法求解递归方程

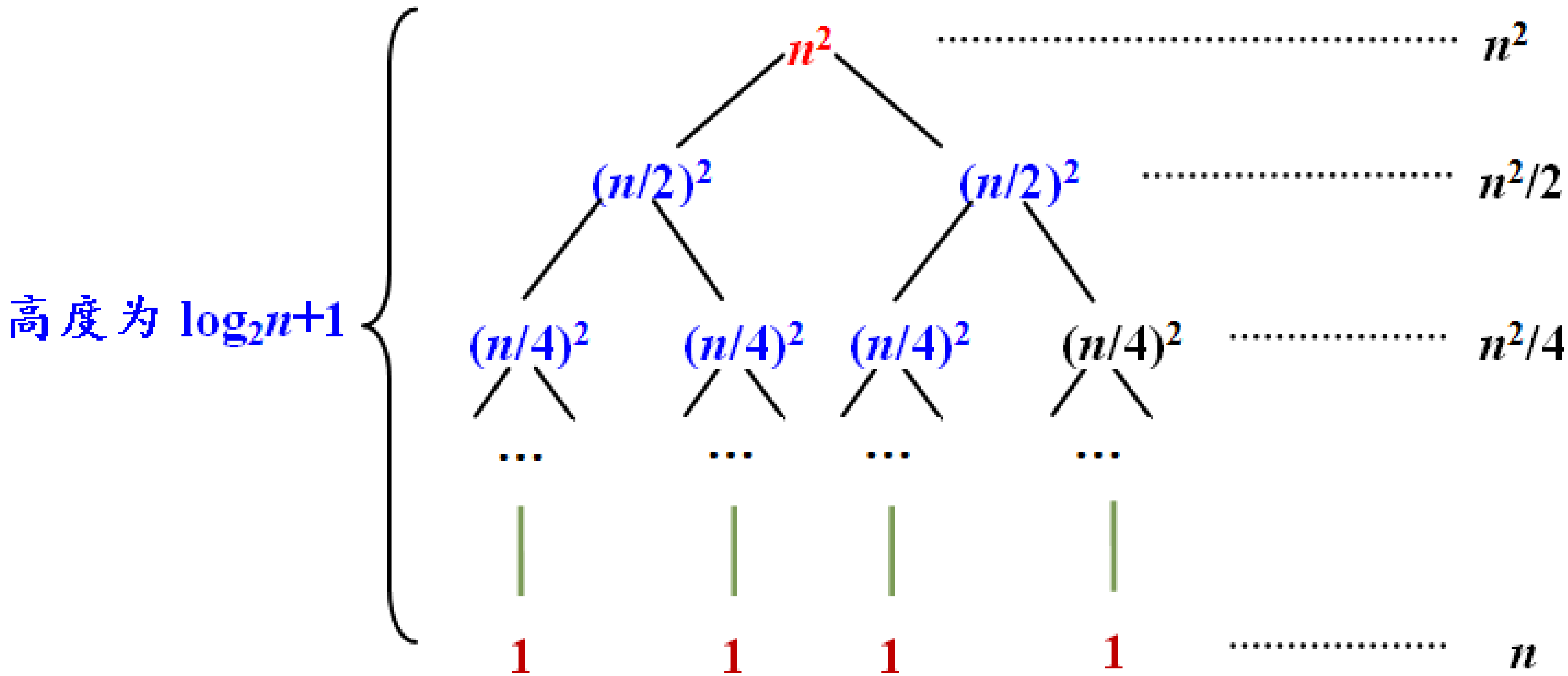
【例】用递归树法分析以下递归方程的时间复杂度：

$T(n)=1$	当$n=1$
$T(n)=2T(n/2)+n^2$	当$n>1$

2.5递归算法分析

3.递归树方法求解递归方程

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

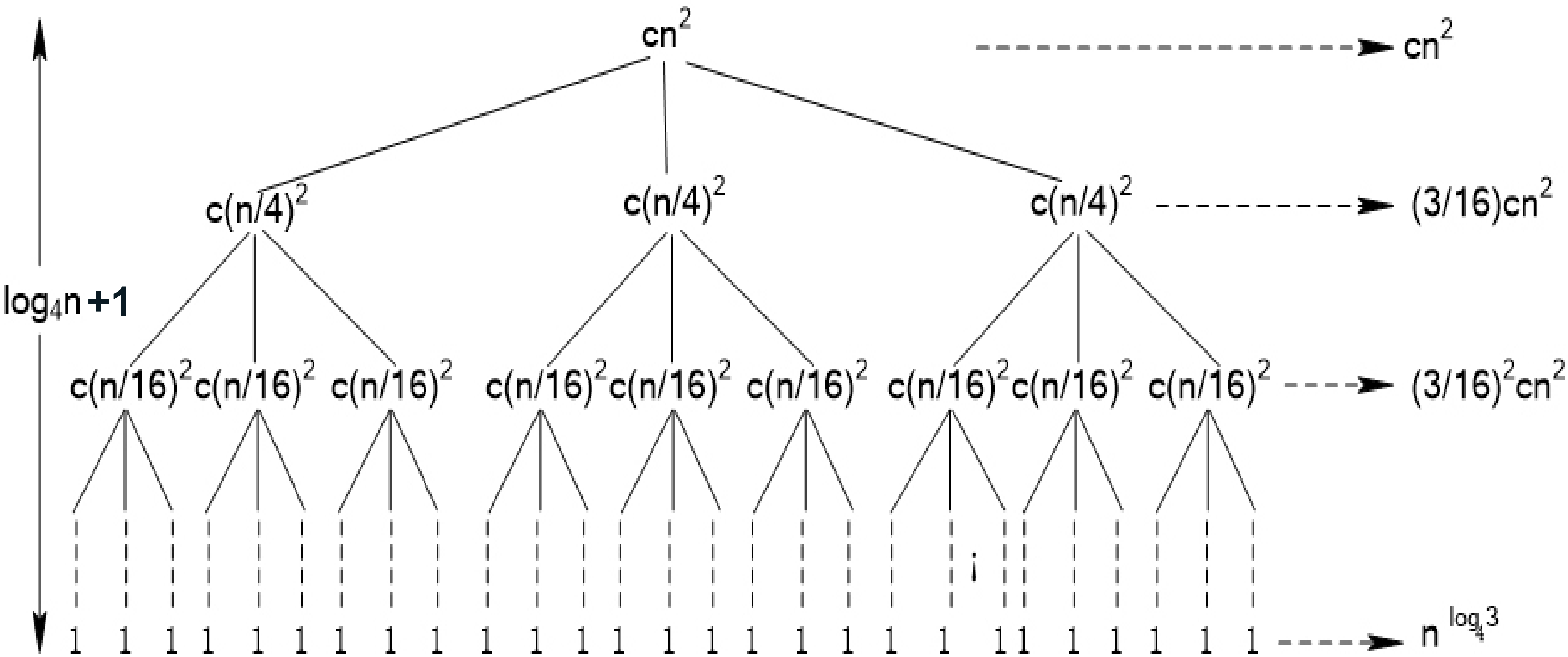


$$T(n)=n^2+n^2/2+\cdots+n^2/2^{k-1}+\cdots+n=0(n^2)。$$

2.5递归算法分析

3.递归树方法求解递归方程

【例】 $T(n)=3T(n/4)+cn^2$



$$T(n) = O(n^2)$$

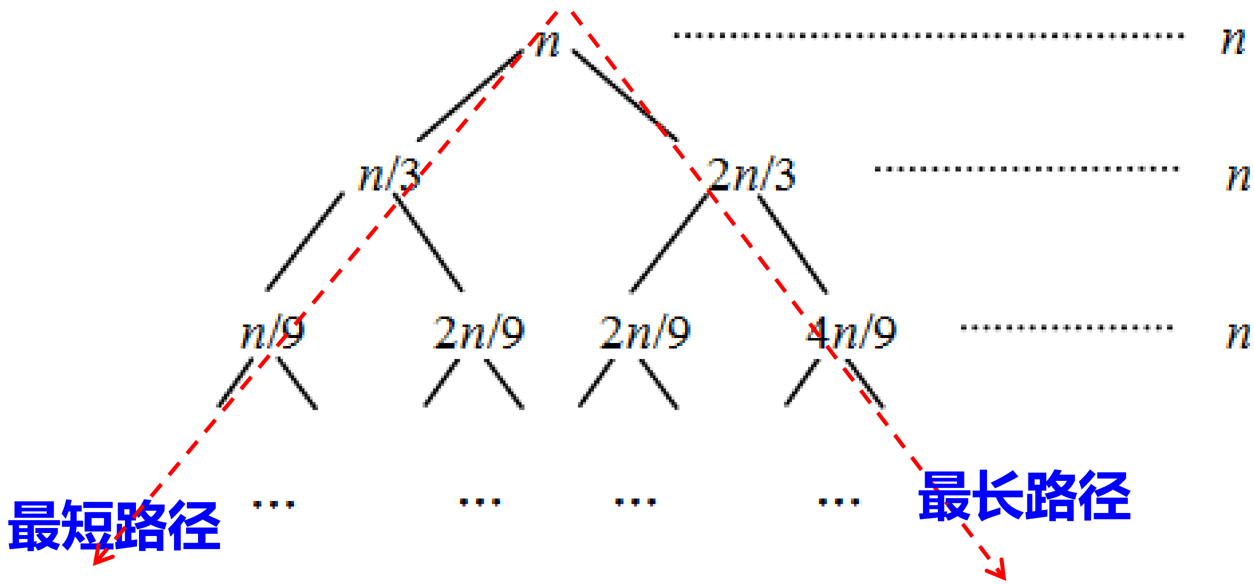
2.5递归算法分析

3.递归树方法求解递归方程

【例】用递归树法分析以下递归方程的时间复杂度：

$$T(n)=\begin{cases}1 & \text{当}n=1\\ T(n/3)+T(2n/3)+n & \text{当}n>1\end{cases}$$

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



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：

$$T(n)=O(n\log_{3/2}n)=O(n\log_2 n)。$$

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。

4．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：

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

其中a≥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递归算法分析

4．主方法求解递归方程

主定理： 设$a\geqslant 1$，$b>1$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ε>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ε>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$af(n/b)\leqslant cf(n)$，那么$T(n)=O(f(n))$。

4. 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：

$T(n)=1$	当$n=1$
$T(n)=4T(n/2)+n$	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$T(n)=O(n^{\log_b a})$

$=O(n^2)$。

【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。