第2章：递归与分治策略

知识要点

- **递归的概念和典型的递归问题**
  - 阶乘、Fibonacci数列、hanoi塔等问题
- **分治法的基本思想**
- **分治法的典型例子**
  - 二分搜索、矩阵乘法、归并排序、快速排序
  - 大整数的乘法、最接近点对问题

2.1 递归的概念

递归(recursion)

- **什么是递归?**
  - **程序调用自身的编程技巧称为递归**
  - **基本思路**
    - 将一个大型的复杂问题转化为
    - 一些与原问题相似的规模较小的问题来求解

递归(recursive)

- 如果函数调用它本身，那么此函数就是递归的
- 例如n!的定义就是递归的：n! = n × (n – 1)!
- 考察下面的函数：

```
int fact(int n)
{
    if (n <= 1) //初值，1！=1
        return 1;
    else
        return n * fact(n - 1);
}
```

递归终止条件

递归表达式

- 为了解递归的工作原理，我们来跟踪 **fact(4)** 的执行

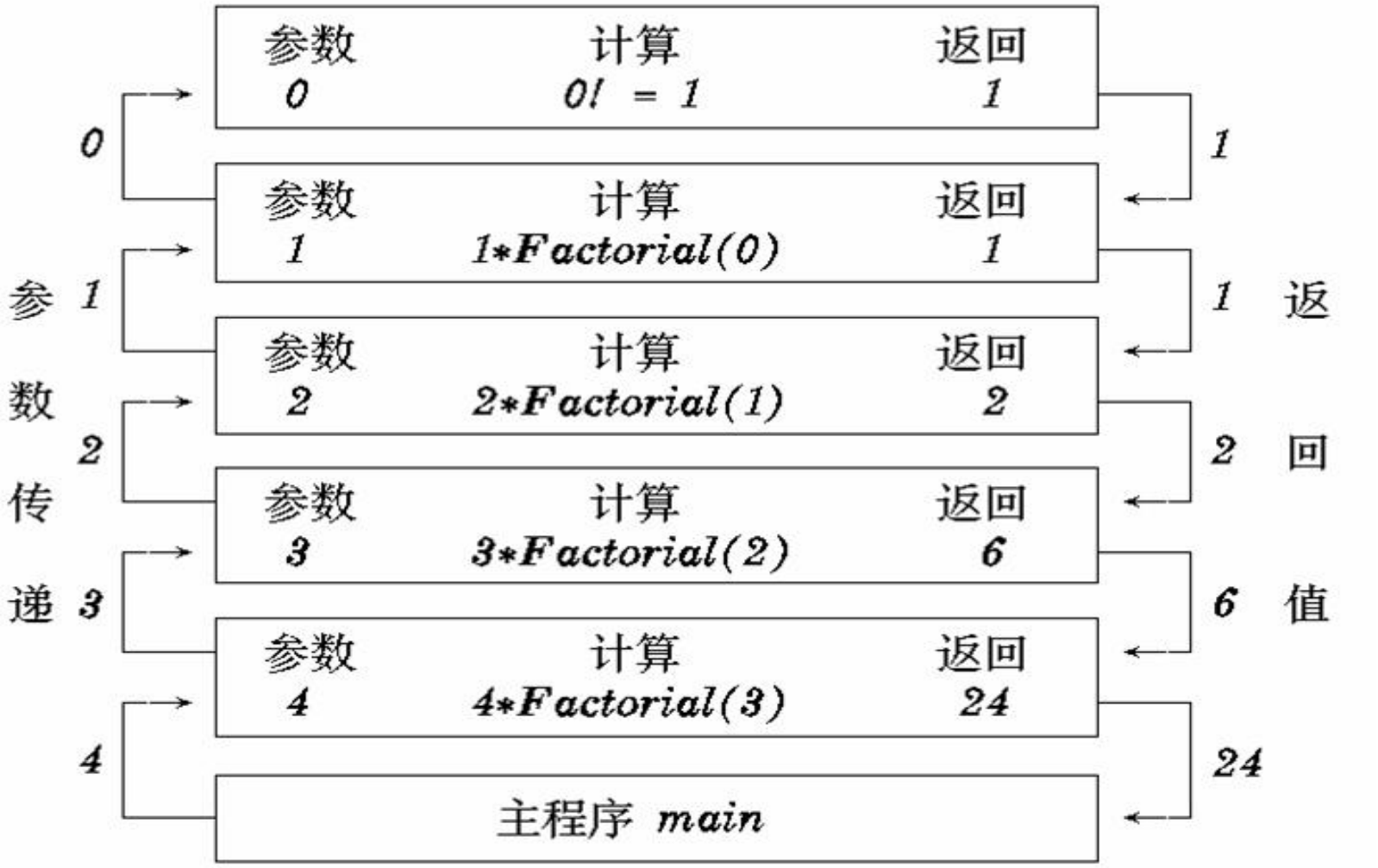
调用函数时系统的工作

- **在调用函数时系统需要完成3件事：**
  - 将所有实参（指针），返回地址传递给被调用的函数
  - 为被调用函数的局部变量分配存储区
  - 将控制转移到被调用函数的入口
- **从被调用函数返回时系统也要做3件事：**
  - 保存被调用算法的计算结果（返回值）
  - 释放分配给被调用算法的存储空间
  - 依照被调算法保存的返回地址将控制转移回到调用算法

递归过程与递归工作栈

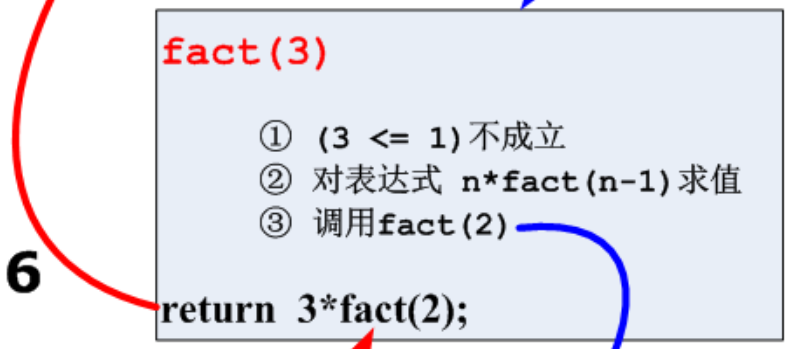
- **递归过程执行时需多次调用自身。多个(相同)函数嵌套调用，信息传递和控制转移通过栈实现**
  - 每一次递归调用时，需要为过程中所使用的参数、局部变量等另外分配存储空间
  - 层层向下递归，退出时次序正好相反
  - 每层递归调用需分配的空间形成递归工作记录，用栈按照后进先出规则管理这些信息

阶乘的递归调用过程

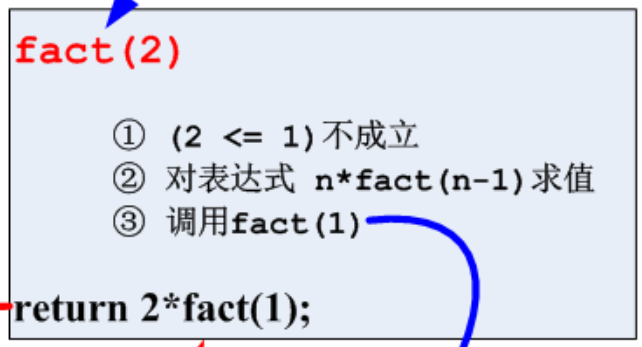

参数 计算 返回
0 0! = 1 1
参数 计算 返回
1 1*Factorial(0) 1
参数 计算 返回
2 2*Factorial(1) 2
参数 计算 返回
3 3*Factorial(2) 6
参数 计算 返回
4 4*Factorial(3) 24
主程序 main
参数传递
返回值
0
1
2
3
4
1
1
2
6
24

```
int main(void){
    int a = fact(3);
    return 0;
}
```

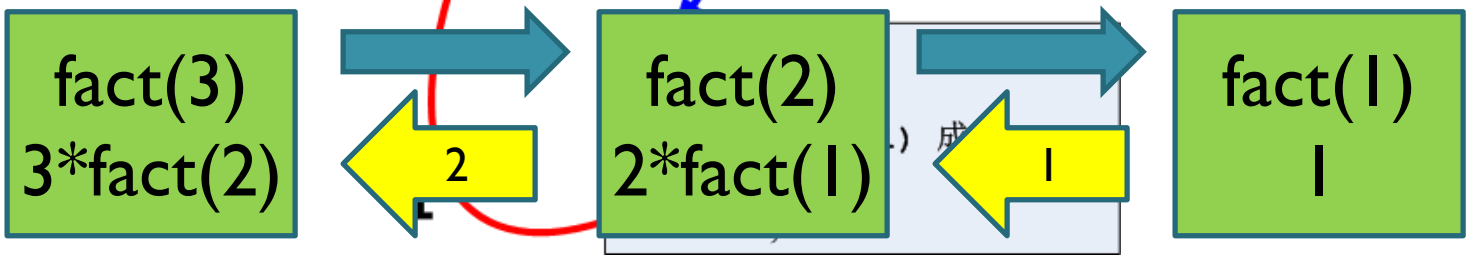
```
int fact(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * fact(n - 1);
}
```



6



2



递归的应用场景

- 遇到如下三种情况，可以考虑使用递归

1. 问题定义是递归的
2. 解决问题时采用的数据结构是递归定义的
3. 问题的求解过程是递归的

递归的应用场景（1）

问题定义是递归的

示例：递归法求阶乘

- 利用公式 $x^n = x \times x^{n-1}$计算出x^n的值

```
int power(int x, int n)
{
  if ( n == 0 )
    return 1;
  else
    return x * power( x, n - 1 );
}
```

- **可化简为：**

```
int power(int x, int n)
{
  return n == 0 ? 1 : x * power(x, n - 1);
}
```

斐波纳契数列（Fibonacci Sequence）

- **斐波纳契数列：1、1、2、3、5、8、13、21、……**
  - F0=1，F1=1
  - Fn = F(n-1) + F(n-2)，（n>=2，n∈N）

问题定义是递归的!

斐波纳契数列

- 递归定义式

$$F(n)=\begin{cases}1 & n=0,1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

- 非递归定义式：

$$F(n)=\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}-\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right)$$

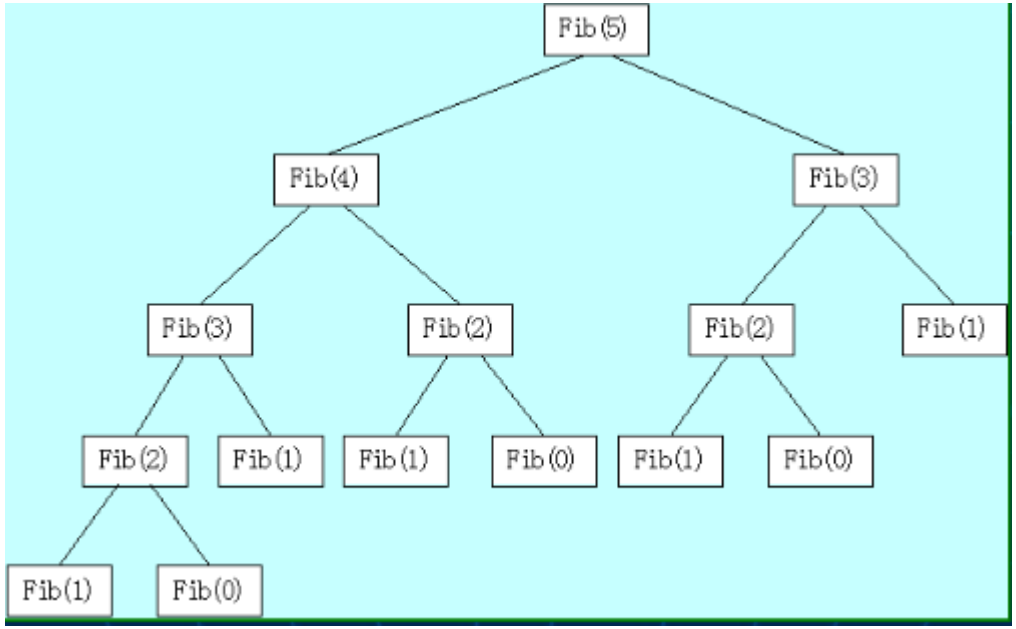
斐波纳契数列解法1：递归

```
long fib1(int n){
    if (n <= 1) {
        return 1;
    }
   else{
        return fib1(n - 1) + fib1(n - 2);
    }
}
```

斐波那契数列的递归求解过程

- 调用次数 NumCall(k)=$2^{k-1}-1$
- fibonacci(5)的递归求解过程：

算法复杂度：$O(2^n)$



斐波纳契数列解法2：递推

递归解法的问题在于：重复求解子问题

观察Fib(n)的定义：F(n) = F(n-1) + F(n-2)；(n>=2)

F(n) 具有“无后效性”：只需“记住”前两个状态的结果即可

```
long fib2(int n){
    long f1 = 1, f2 = 1, fu;
    for(int i = 2; i <= n; ++i){
        fu = f1 + f2;
        f1 = f2; f2 = fu;  // 记忆
    }
    return fu;
}
```

算法复杂度：O(n)

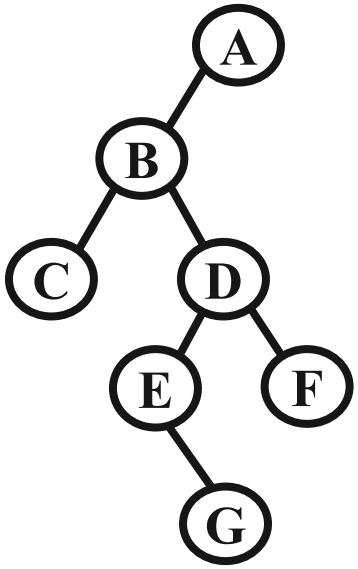
递归的应用场景（2）

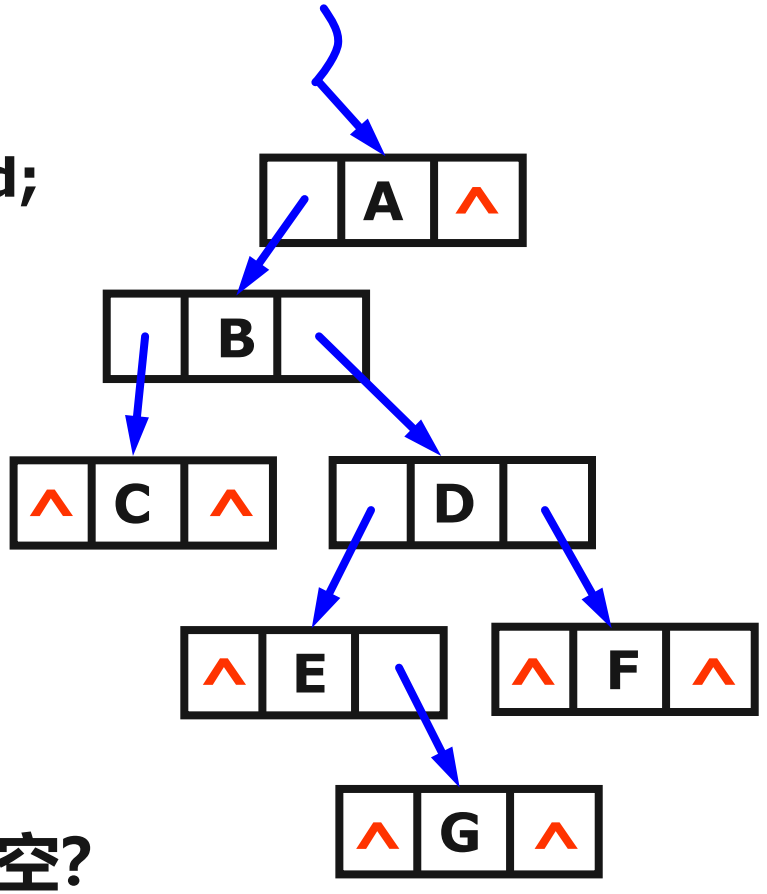
问题采用的数据结构是递归的

递归算法示例2：数据结构是递归的

二叉树链式存储结构：二叉链表


```
typedef   struct  node{
      Entrytype    data;
      struct  node  *lchild,  *rchild;
} BTNode, *BTPtr;
```





有多少指针为空?

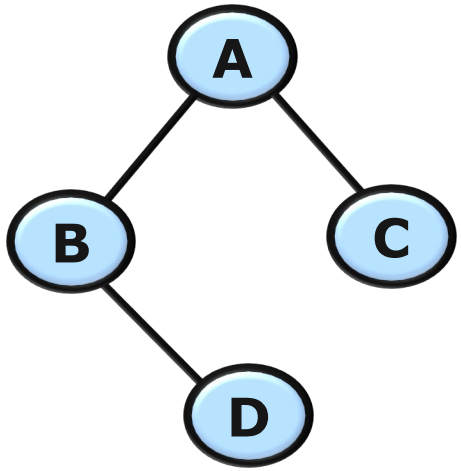
在n个结点的二叉链表中，有n+1个空指针域

☙ 中序遍历递归算法

```
void inorder ( BTPtr bt) {
    if(bt != NULL){
        inorder ( bt->lchild );
        printf ("%c\t", bt->data);
        inorder ( bt->rchild );
    }
    return;
}
```

中序序列：B D A C

后序序列：D B C A



☙ 后序遍历递归算法

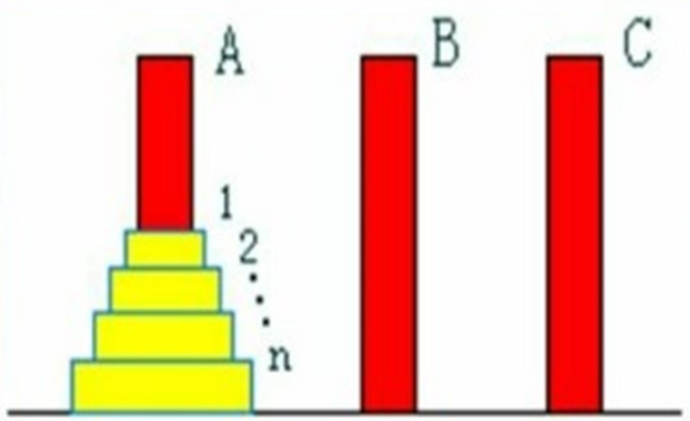
```
void postorder( BTPtr bt) {
    if(bt != NULL){
        postorder( bt->lchild );
        postorder( bt->rchild );
        printf ("%c\t", bt->data);
    }
    return;
}
```

递归的应用场景（3）

问题的求解过程是递归的

示例1：Hanoi Tower（汉诺塔）问题

- **问题描述：设有A、B、C3个塔座**



  - 在塔座A上有一叠共n个圆盘
    - ☑ 自上而下，由小到大地叠在一起
    - ☑ 自上而下依次编号为1,2,…,n
  - 问题：要求将塔座A上的圆盘全部移到塔座C上，仍按同样顺序叠置。在移动圆盘时遵守以下规则：
    - ☑ 每次只允许移动1个圆盘
    - ☑ 任何时刻都不允许将较大的圆盘压在较小的圆盘之上
    - ☑ 在规则1和2的前提下，可将圆盘移至任何一塔座上

用递归技术求解汉诺塔问题

- 将3个盘子从A移至C，以B为辅助（7步完成）

(1) 1#A->C

(2) 2#A->B

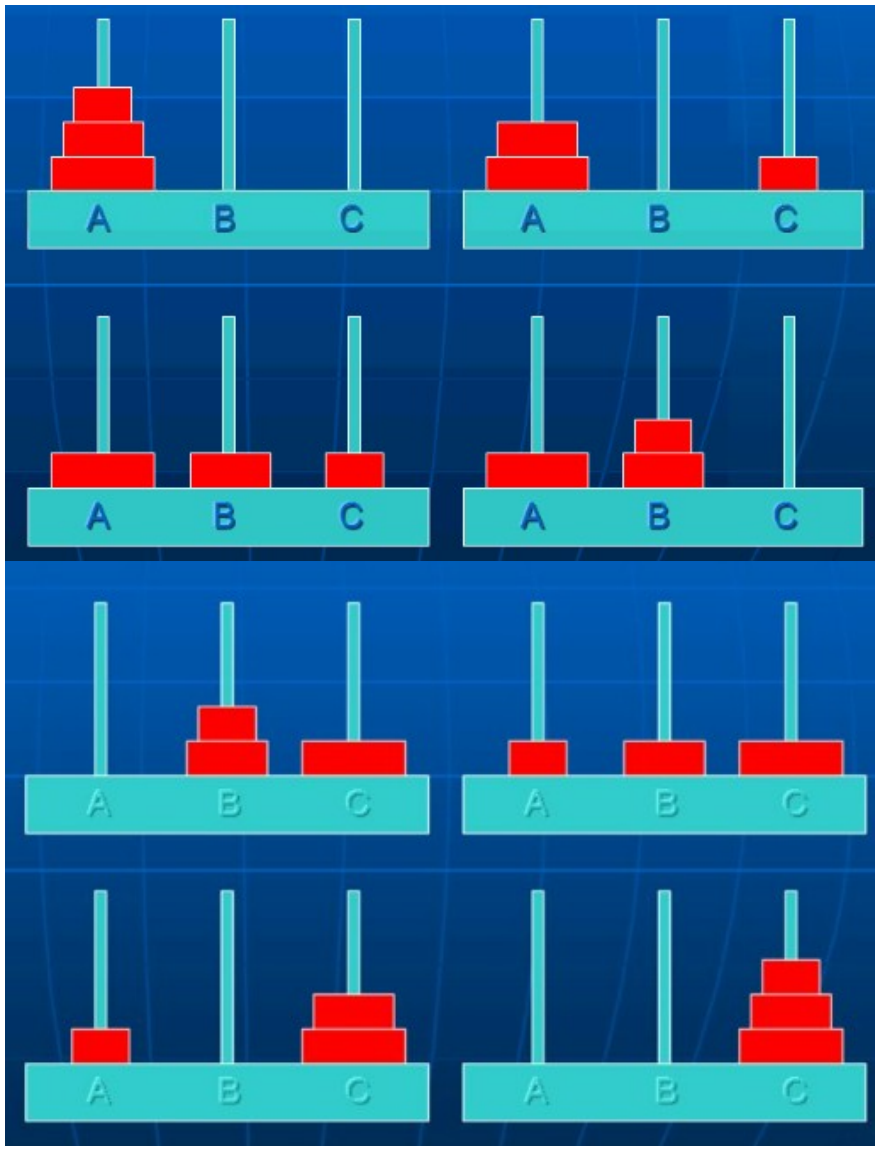
(3) 1#C->B

(4) 3#A->C

(5) 1#B->A

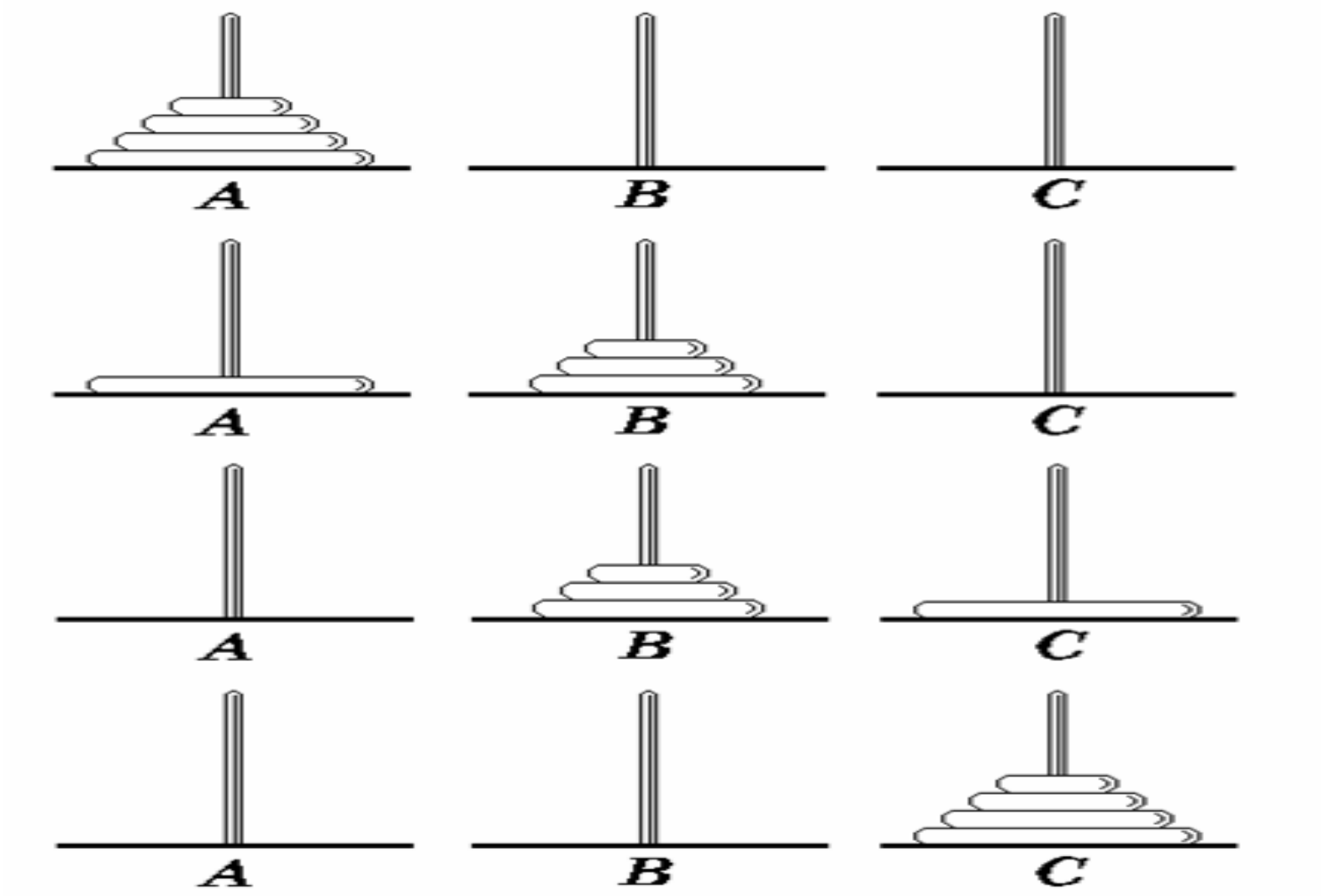
(6) 2#B->C

(7) 1#A->C



用递归技术求解汉诺塔问题

- 将4个盘子从A移至C，以B为辅助



汉诺塔问题的规模

- **一般的Hanoi塔玩具不超过8片**
  - 如果n=8，需移动255次
  - 如果n=10，需移动**1023**次
  - 如果n=15，需移动**32767**次
  - 如果n=20，需移动**1048575**次（超过一百万次）
  - 如果n=64，需移动$\mathbf{2^{64}-1}$次（超过一百万次）
    - ☑ 如果每秒移动一块圆盘，需5845.54亿年
    - ☑ 按照宇宙大爆炸理论推测，宇宙的年龄也仅为137亿年

用递归技术求解汉诺塔问题

☙ 算法设计思路

- 当n=1时，问题可以直接求解，一步完成
- 当n>1时，分三步完成：
  - ➢ 将n-1个较小盘子设法移动到辅助塔座
    - – 构造出一个比原问题规模小1的问题
  - ➢ 将最大的盘子从原塔座一步移至目标塔座
  - ➢ 将n-1个较小的盘子设法从辅助塔座移至目标塔座
    - – 仍然是比原问题规模小1的问题

汉诺塔问题的递归算法

```
void hanoi (int  n, int src, int tar, int aux){
    if(n>0){
        hanoi(n-1,  src,  aux, tar);
        move(src, tar);
        hanoi(n-1,  aux, tar, src);
    }
}
```

其中：hanoi(int n, int src, int tar, int aux) 表示将塔座src上的n个盘子移动到塔座tar，以塔座aux为辅助（auxiliary）

示例2：排列问题

- **设计一个递归算法：生成n个元素{r_1,r_2,…,r_n}的全排列**
  - 设**R={r_1,r_2,…,r_n}**是要进行排列的n个元素，**R_i=R-{r_i}**
  - 集合**X**中元素的全排列记为：**perm(X)**
  - **(r_i)perm(X)**：在perm(X)的每个排列前加上前缀得到的排列
  - **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)的构成情况如下：
    - (r_1)perm(R_1)，(r_2)perm(R_2)，…，(r_n)perm(R_n)

```
template<class Type>
void Perm(Type list[ ], int k, int m)
{ //产生list[k:m]的所有排列
    if (k==m)
    { //只剩下一个元素
        for (int i=0; i<=m; i++) cout <<list[i];
        cout<<endl;
    }
    else //还有多个元素待排列，递归产生排列
      for(int i=k; i<=m; i++)
      { Swap(list[k], list[i]);
        Perm(list, k+1, m);
        Swap(list[k], list[i]);
      }
}
```

```
template<class Type>
inline void Swap(Type &a, Type &b)
{  Type temp=a; a=b; b=temp;
}
```

排列问题：产生list[k:m]的所有排列

```
void perm(type R[], int k, int m) {
    if (k==m) {    // 只剩下一个元素
        for (int i=0; i<=m; i++)
            cout << R[i];
        cout<<endl;
    }
    else {          // 还有多个元素待排列，递归产生排列
        for(int i=k; i<=m; i++) {
            int tmp = R[k]; R[k] = R[i]; R[i] = tmp;;
            perm(R, k+1, m);
            tmp = R[k]; R[k] = R[i]; R[i] = tmp;
        }
    }
}
```

整数划分问题

将正整数n表示成一系列正整数之和：$n=n_1+n_2+\ldots+n_k$，其中$n_1\geq n_2\geq\ldots\geq n_k\geq 1$，$k\geq 1$。
正整数n的这种表示称为正整数n的划分。求正整数n的不同划分个数。

例如正整数6有如下11种不同的划分：

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。

整数划分问题

前面的几个例子中，问题本身都具有比较明显的递归关系，因而容易用递归函数直接求解。

在本例中，如果设p(n)为正整数n的划分数，则难以找到递归关系，因此考虑增加一个自变量：将最大加数n_1不大于m的划分个数记作q(n,m)。可以建立q(n,m)的如下递归关系。

(3) q(n,n)=1+q(n,n-1);

正整数n的划分由$n_1=n$的划分和$n_1 \le n-1$的划分组成。

(4) q(n,m)=q(n,m-1)+q(n-m,m),n>m>1;

正整数n的最大加数n_1不大于m的划分由$n_1 \le m-1$ 的划分和$n_1=m$的划分组成。

整数划分问题

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

正整数n的划分数p(n)=q(n,n)。

整数划分问题

伪代码：

```
int q(int n,int m) {
    if (n<1 || m<1)  return 0;
    if (n==1 || m==1)  return 1;
    if (n<m)  return q(n,n);
    if (n==m)  return q(n,n-1)+1;
     return  q(n,m-1)+q(n-m,m); }
```

递归小结

- **递归算法（recursive algorithm)**
  - 直接或间接的调用自身的算法。
- **递归函数（recursive function）**
  - 用函数自身给出定义的函数。
- **递归式：描述递归函数的数学公式**
  - 它是一组等式或者不等式。
  - 它的第一式给出了函数的初始值，称为边界条件。
  - 他的第二式是用较小自变量的函数值来描述大自变量的函数值，称为递归方程。
  - 边界条件和递归方程是递归的两个基本要素。

递归小结

使用递归的注意事项

- 原始问题可以**分解**为相似的子问题
- 子问题的**规模小于**原始问题
  - **思考：只要问题变小就适合用递归吗?**
- 递归函数必须有某些类型的**终止条件**

递归的应用

- 问题的定义是递归的，如阶乘问题
- 问题的求解过程是递归的，如汉诺塔问题
- 问题采用的数据结构是递归的，如链表中搜索元素

递归小结

递归算法的优点

- 结构清晰，易于理解，而且容易用数学归纳法来证明算法的正确性，因此用递归技术来设计算法很方便。

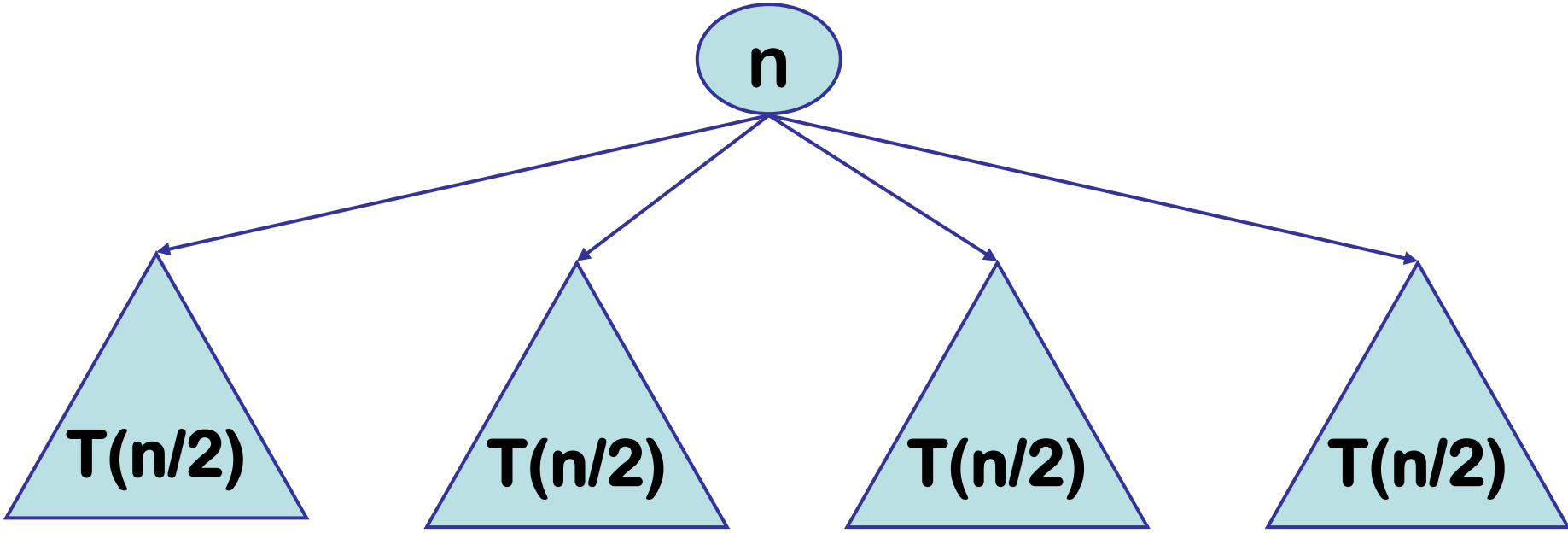
递归算法的缺点

- 在执行时要多次调用自身，运行效率较低，无论是计算时间还是占用存储空间都要比非递归算法要多。
- 一些运算步骤可能重复运算，会进一步降低效率。

2.2 分治法的思想

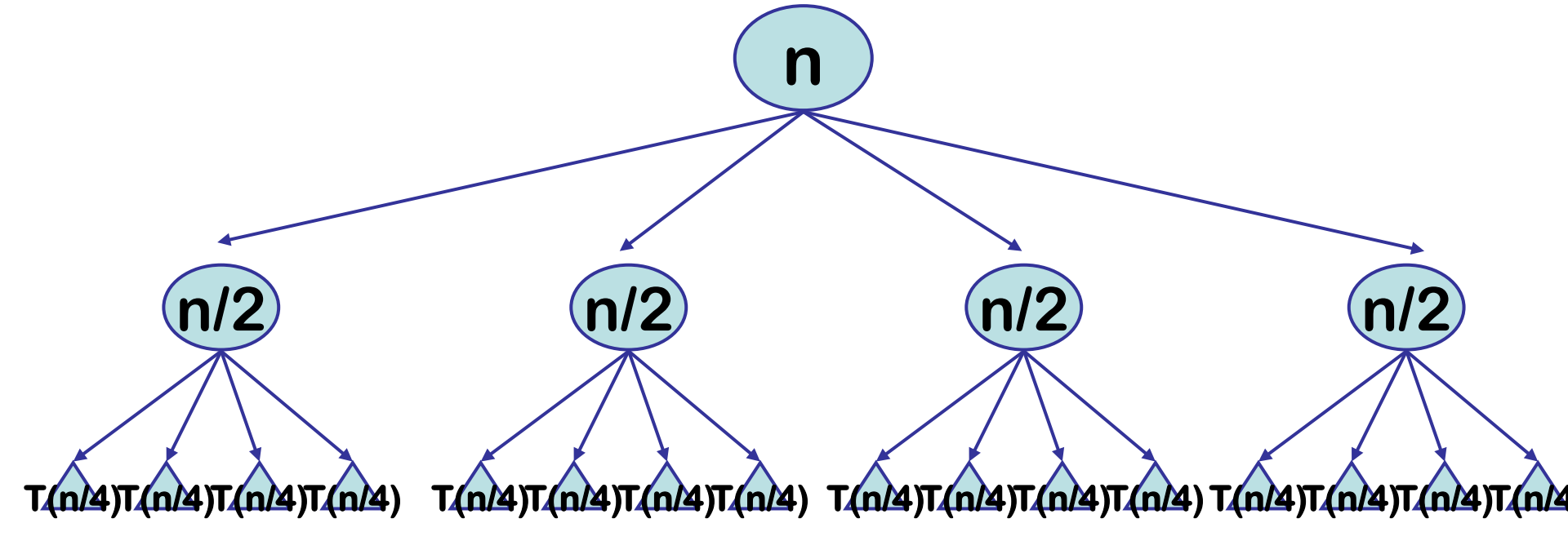
算法总体思想

- 对这k个子问题分别求解。如果子问题的规模仍然不够
- 小，则再划分为k个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止。



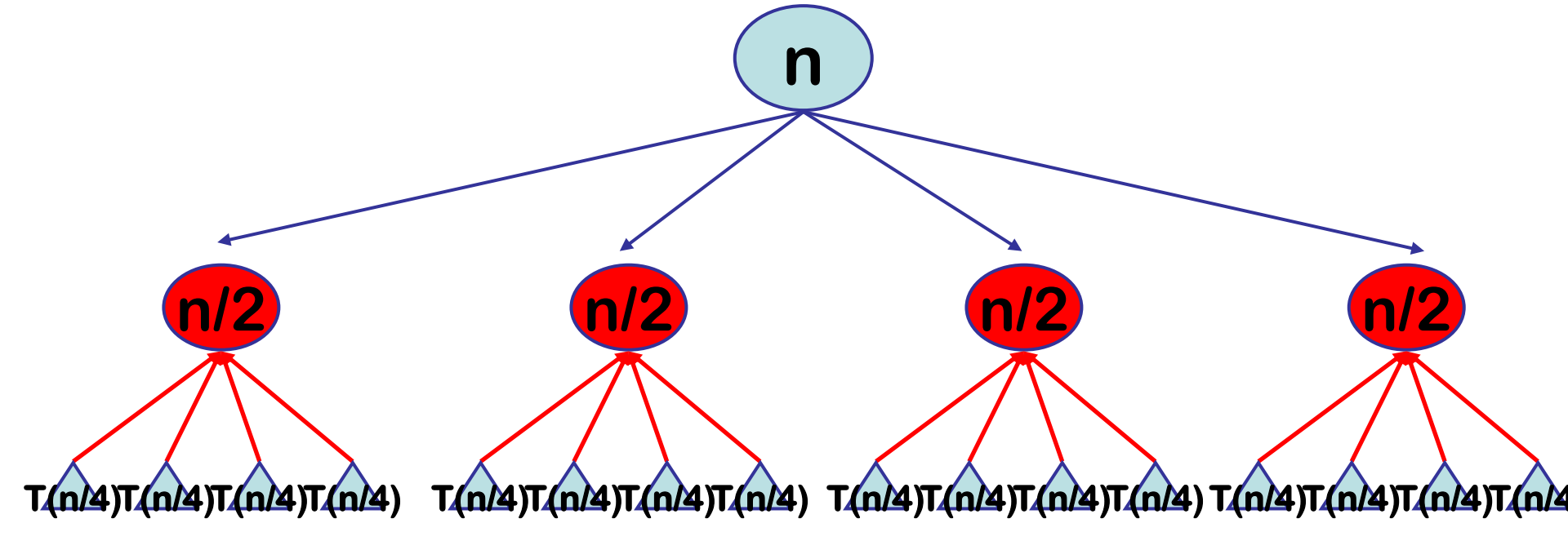
算法总体思想

- 将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。



算法总体思想

- 将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。



算法总体思想

- 将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。

分治法的设计思想是，将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

分治法的基本步骤

```
divide-and-conquer(P) {
    if ( | P | <= n0)  solve(P);   // 解决小规模的问题
    divide P into smaller subinstances P1,P2,...,Pk;   // 分解
    for ( i=1, i<=k, i ++)
        yi=divide-and-conquer(Pi);   // 递归地解各子问题
    return merge(y1,...,yk);  // 合并子问题的解为原问题的解
}
```

经验：实践表明，在用分治法设计算法时，最好使子问题的规模大致相同，即：将一个问题分成大小相等的k个子问题。

这种使子问题规模大致相等的做法是出自一种平衡(balancing)子问题的思想，它几乎总是比子问题规模不等的做法要好。

分治法（Divide-and-Conquer）

- **分治模式在每一层递归上都有三个步骤**
  - **分解（Divide)；求解（Conquer）；合并（Combine)**
- **分治法的设计思想：**
  - **Divide**：将一个难以直接解决的大问题，分割成一些规模较小的子问题，这些子问题互相独立，且与原问题相同
  - **Conquer**：递归求解子问题，若问题足够小则直接求解
  - **Combine**：将各子问题的解合并得到原问题的解

求二叉树深度

```
int get_depth(BTPtr  pbt){
    int dL = 0, dR = 0
    if( pbt == NULL ) {
        return 0;
    }
    if((!pbt->lchild) && (!pbt->rchild)){
        return 1;
    }
    dL = get_depth(pbt->lchild);
    dR = get_depth(pbt->rchild);
    // 二叉树的深度为根结点左、右子树的深度值较大者加一
    return 1 + ((dL > dR) ? dL : dR);
}
```

Divide

Conquer

Combine

分治法的适用条件

- **分治法所能解决的问题一般具有以下四个特征：**

1. 该问题的规模缩小到一定的程度就可以容易地解决

& 因为问题的计算复杂性一般是随着问题规模的增加而增加，因此大部分问题满足这个特征

2. 该问题可以分解为若干个规模较小的相同问题

& 即：该问题具有**最优子结构性质**

& 这条特征是应用分治法的前提，它也是大多数问题可以满足的，此特征反映的就是递归算法的设计思想

分治法的适用条件

- **分治法所能解决的问题一般具有以下四个特征：**

3. 利用子问题的解可以合并得到原始问题的解

& 能否利用分治法完全取决于问题是否具有这条特征

& 如果具备了前两条特征，而不具备第三条特征，则可以考虑贪心算法或动态规划

4. 该问题所分解出的各个子问题是相互独立的

& 即：子问题之间不包含公共的子问题

& 这条特征涉及到分治法的效率，如果各子问题是不独立的，则采用分治法需要重复地求解公共的子问题，此时虽然也可用分治法，但一般用动态规划较好

分治法示例1：二分搜索技术

- **给定：已按升序排好序的n个元素a[0:n-1]**
- **问题：在这n个元素中找出一特定元素x。**
- **分析1：该问题的规模缩小到一定的程度就可以容易地解决**
  - 如果n=1，则通过一次比较就可以解决问题
- **分析2：该问题可以分解为若干个规模较小的相同问题**
  - 取中间元素a[mid]对序列进行划分
  - 在a[mid]前面或后面查找x，其方法都和在a中查找x一样
- **分析3：分解出的子问题的解可以合并为原问题的解**
- **分析4：分解出的各个子问题是相互独立的**
  - 在a[i]的前面或后面查找x是独立的子问题

分治法示例1：二分搜索算法

```
int BinarySearch(int a[], int x, int left, int right) {
    while (right >= left){
        int mid = (left+right)/2;
        if (x == a[m]) return mid;
        if (x < a[m]) right = mid-1;
        else left = mid+1;
    }
    return -1;
}
```

算法复杂度分析：每执行一次算法的while循环， 待搜索数组的大小减少一半。因此，在最坏情况下，while循环被执行了O(logn) 次。循环体内运算需要O(1) 时间，因此整个算法在最坏情况下的计算时间复杂性为O(logn) 。

Master定理（递归复杂度判定定理）

设常数a >= 1，b > 1，f(n)为函数，

T(n)为非负整数，$T(n)=aT(n/b)+f(n)$ ，则有：

1.若 $f(n)=\mathrm{O}(n^{\log_b a-\varepsilon}),\varepsilon>0$，则 $\mathrm{T}(n)=\Theta(n^{\log_b a})$ ；

2.若 $f(n)=\Theta(n^{\log_b a})$ ，则 $T(n)=\Theta(n^{\log_b a}\log n)$ ；

3.若 $f(n)=\Omega(n^{\log_b a+\varepsilon}),\varepsilon>0$ ，并且对于某个常数c<1

和充分大的n有 $af(n/b)\le cf(n)$ ，那么 $T(n)=\Theta(f(n))$ 。

分治法示例2：大整数的乘法

- 问题：设计一个可以进行两个n位大整数乘法运算的算法
- 逐位相乘、错位相加的传统方法：$O(n^2)$ → 效率太低
- 分治法：将该问题分解为若干个规模较小的相同问题
  - X = | a | b |
  - Y = | c | d |
  - $X = a\,2^{n/2} + b \qquad Y = c\,2^{n/2} + d$
  - $XY = ac\,2^{n} + (ad+bc)\,2^{n/2} + bd$
- 复杂度分析

$$T(n) = \begin{cases} O(1) & n = 1 \\ 4T(n/2) + O(n) & n > 1 \end{cases}$$

$T(n)=O(n^2)$ ✘没有改进

分治法示例2：大整数的乘法

- 思考：怎样改进?
  - XY = ac 2^n + (ad+bc) $2^{n/2}$ + bd
  - 为了降低时间复杂度，必须减少乘法的次数
  - XY = ac 2^n + ((a-b)(d-c)+ac+bd) $2^{n/2}$ + bd
  - XY = ac 2^n + ((a+b)(c+d)-ac-bd) $2^{n/2}$ + bd
- 复杂度分析

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

$T(n)=O(n^{\log 3}) = O(n^{1.59})$ ✓ **较大地改进**

细节问题：两个XY的复杂度都是$O(n^{\log 3})$，但考虑到a+c,b+d可能得到(n/2)+1位的结果，使问题的规模变大，故不选择第2种方案。

分治法示例3：Strassen矩阵乘法

- **求矩阵乘积的传统算法**

```
// 以二维数组存储矩阵元素，C为 A 和 B的乘积
void multi(int A[], int B[], int C[]){
    for (int i=1; i<=n; ++i){
        for (int j=1; j<=n; ++j){
            C[i,j] = 0;
            for (int k=1; k<=n; ++k) {
                C[i,j] += A[i,k]*B[k,j];
            }
        }
    }
}
```

时间复杂度： $O(n^3)$

分治法示例3：Strassen矩阵乘法

- 分治法
  - 与整数乘法类似，可以将矩阵A，B和C中每一矩阵都分块成4个大小相等的子矩阵，由此可将方程C=AB重写为：

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

- 由此可得：

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

分治法示例3：Strassen矩阵乘法

- 算法复杂度分析

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

$$T(n) = \begin{cases} O(1) & n = 2 \\ 8T(n/2) + O(n^2) & n > 2 \end{cases}$$

$T(n)=O(n^3)$

分治法示例3：Strassen矩阵乘法

- 思考：怎样改进?
  - 为了降低时间复杂度，必须减少乘法的次数

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$M_1 = A_{11}(B_{12} - B_{22})$$
$$M_2 = (A_{11} + A_{12})B_{22}$$
$$M_3 = (A_{21} + A_{22})B_{11}$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_6 = (A_{12} - A_{22})(B_{21} + B_{22})$$
$$M_7 = (A_{11} - A_{21})(B_{11} + B_{12})$$

$$C_{11} = M_5 + M_4 - M_2 + M_6$$
$$C_{12} = M_1 + M_2$$
$$C_{21} = M_3 + M_4$$
$$C_{22} = M_5 + M_1 - M_3 - M_7$$

分治法示例3：Strassen矩阵乘法

- 算法复杂度分析

$$
\begin{aligned}
M_1 &= A_{11}(B_{12} - B_{22}) \\
M_2 &= (A_{11} + A_{12})B_{22} \\
M_3 &= (A_{21} + A_{22})B_{11} \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_6 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\
M_7 &= (A_{11} - A_{21})(B_{11} + B_{12})
\end{aligned}
$$

$$
\begin{aligned}
C_{11} &= M_5 + M_4 - M_2 + M_6 \\
C_{12} &= M_1 + M_2 \\
C_{21} &= M_3 + M_4 \\
C_{22} &= M_5 + M_1 - M_3 - M_7
\end{aligned}
$$

$$
T(n) = \begin{cases} O(1) & n = 2 \\ 7T(n/2) + O(n^2) & n > 2 \end{cases}
$$

$T(n)=O(n^{\log 7}) = O(n^{2.81})$ ✓**明显改进**

$O(n^3)$ vs. $O(n^{2.81})$

- 设 n = 100,000
- 则： n^3 = 1000000000000000
- 则： $n^{2.81}$ = 112201845430196
- 则： $n^{2.376}$ = 758577575029
- 设CPU主频2.4GHz，并设一个时钟周期内执行一条指令
- 则：每秒可执行浮点运算次数为： 2576980378
- 由此：完成一次矩阵相乘运算所需时间估计为
  - 普通算法：388051秒 (108 小时)
  - Strassen算法：43540秒 (12 小时)
  - 当前最优算法：294秒（ 4.9 分钟)

Strassen矩阵乘法

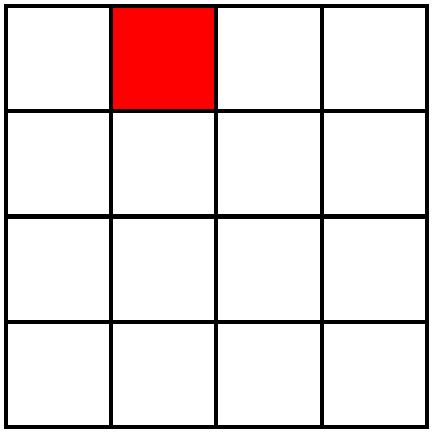
- **更快的方法?**
  - **已经证明：计算2个 2 × 2 矩阵的乘积，7次乘法是必要的**
  - 因此，要想进一步改进矩阵乘法的时间复杂性，或许应当研究基于 3 × 3 或 5 × 5 划分的更好算法
  - 在Strassen之后又有许多算法改进了矩阵乘法的计算时间复杂性。目前最好的算法计算时间上界是 $O(n^{2.376})$
  - **问题：是否能找到$O(n^2)$的算法?**

参考文献： John E.Hopcroft, and Leslie R. Kerr. "On minimizing the number of multiplications necessary for matrix multiplication." SIAM Journal on Applied Mathematics, 1971, 20(1):30-36.

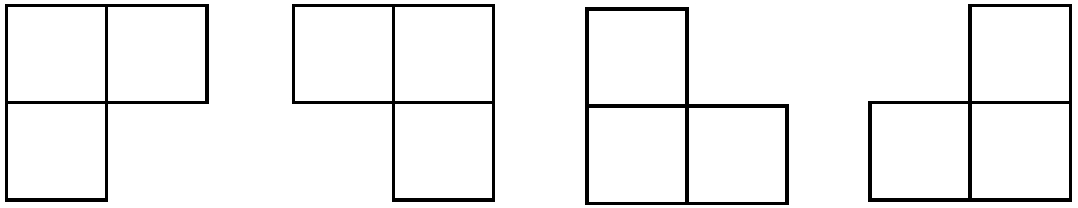
分治法示例4：棋盘覆盖问题

问题描述

- 在一个$2^k \times 2^k$ 个方格组成的棋盘中，有一个方格与其它不同，称该方格为**特殊方格**，且称该棋盘为一特殊棋盘
- 棋盘覆盖问题：
  - 要求用图(b)所示的4种L形态骨牌覆盖给定的特殊棋盘
  - 限制条件：覆盖给定特殊棋盘上除特殊方格以外的所有方格
  - 限制条件：任何2个L型骨牌不得重叠覆盖



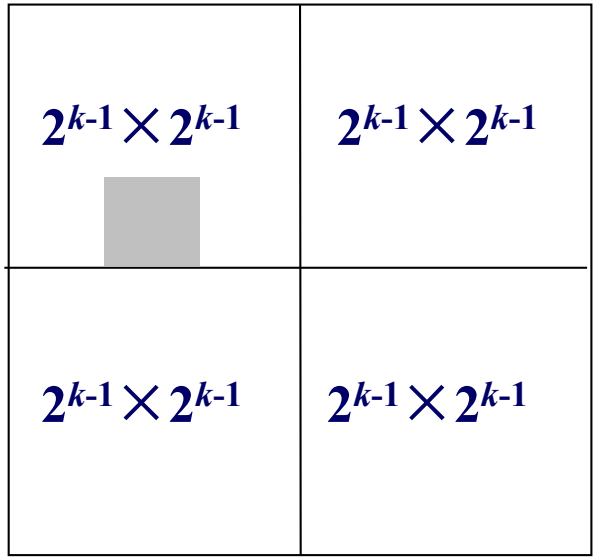
(a) k=2时的一种棋盘



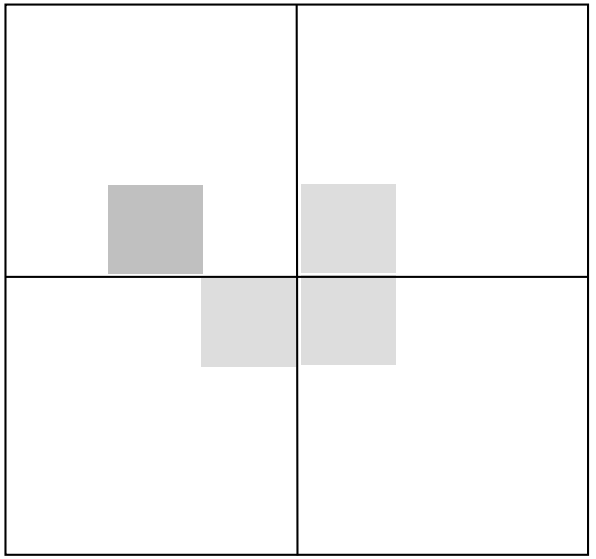
(b) 4种不同形状的L型骨牌

分治策略

- 技巧在于划分棋盘，使每个子棋盘均包含一个特殊方格，从而将原问题分解为规模较小的棋盘覆盖问题



(a)棋盘分割



(b) 构造相同子问题

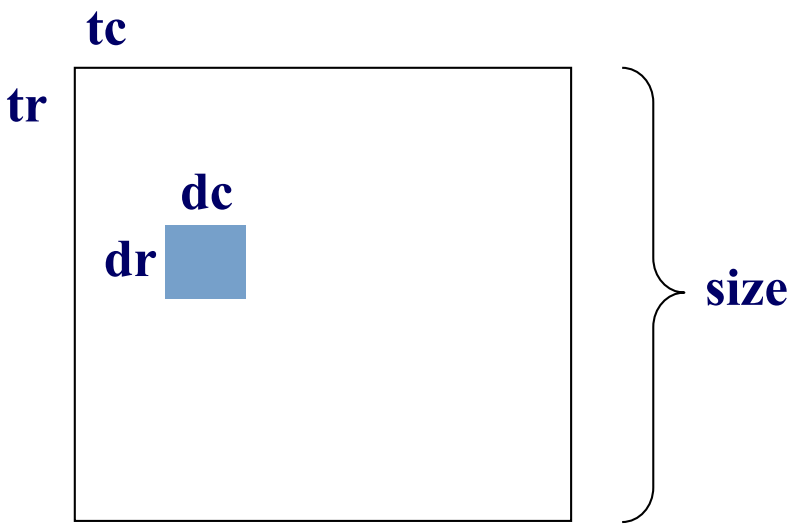
下面介绍棋盘覆盖问题中数据结构的设计：

（1）棋盘：用二维数组board[size][size]表示一个棋盘，其中，size=2^k。为了在递归处理的过程中使用同一个棋盘，将数组board设为全局变量；

（2）子棋盘：在棋盘数组board[size][size]中，由子棋盘左上角的下标tr、tc和棋盘边长s表示；

（3）特殊方格：用board[dr][dc]表示，dr和dc是该特殊方格在棋盘数组board中的下标;

（4） L型骨牌：一个$2^k \times 2^k$的棋盘中有一个特殊方格，所以，用到L型骨牌的个数为$(4^k-1)/3$，将所有L型骨牌从1开始连续编号，用一个全局变量t表示

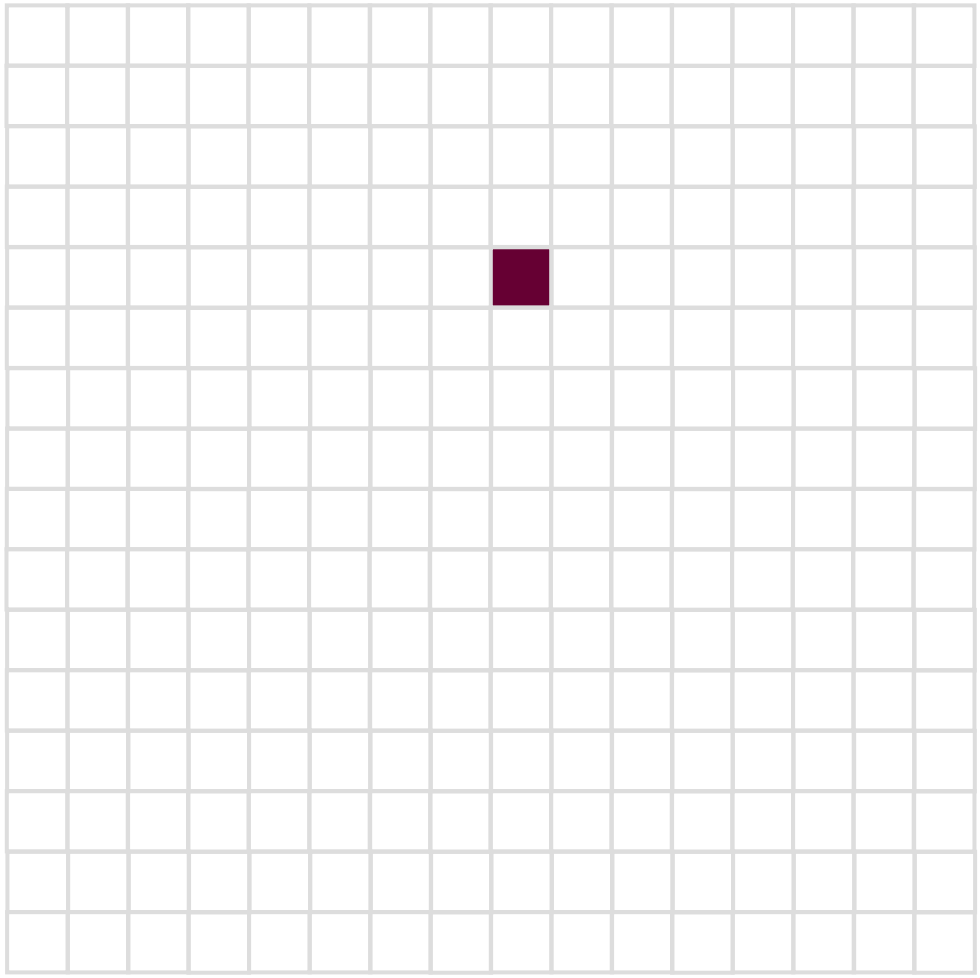


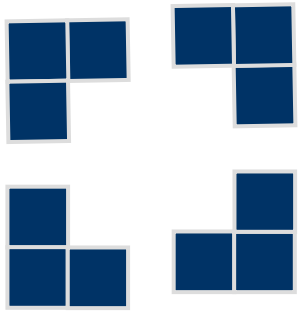
棋盘覆盖问题中的数据结构

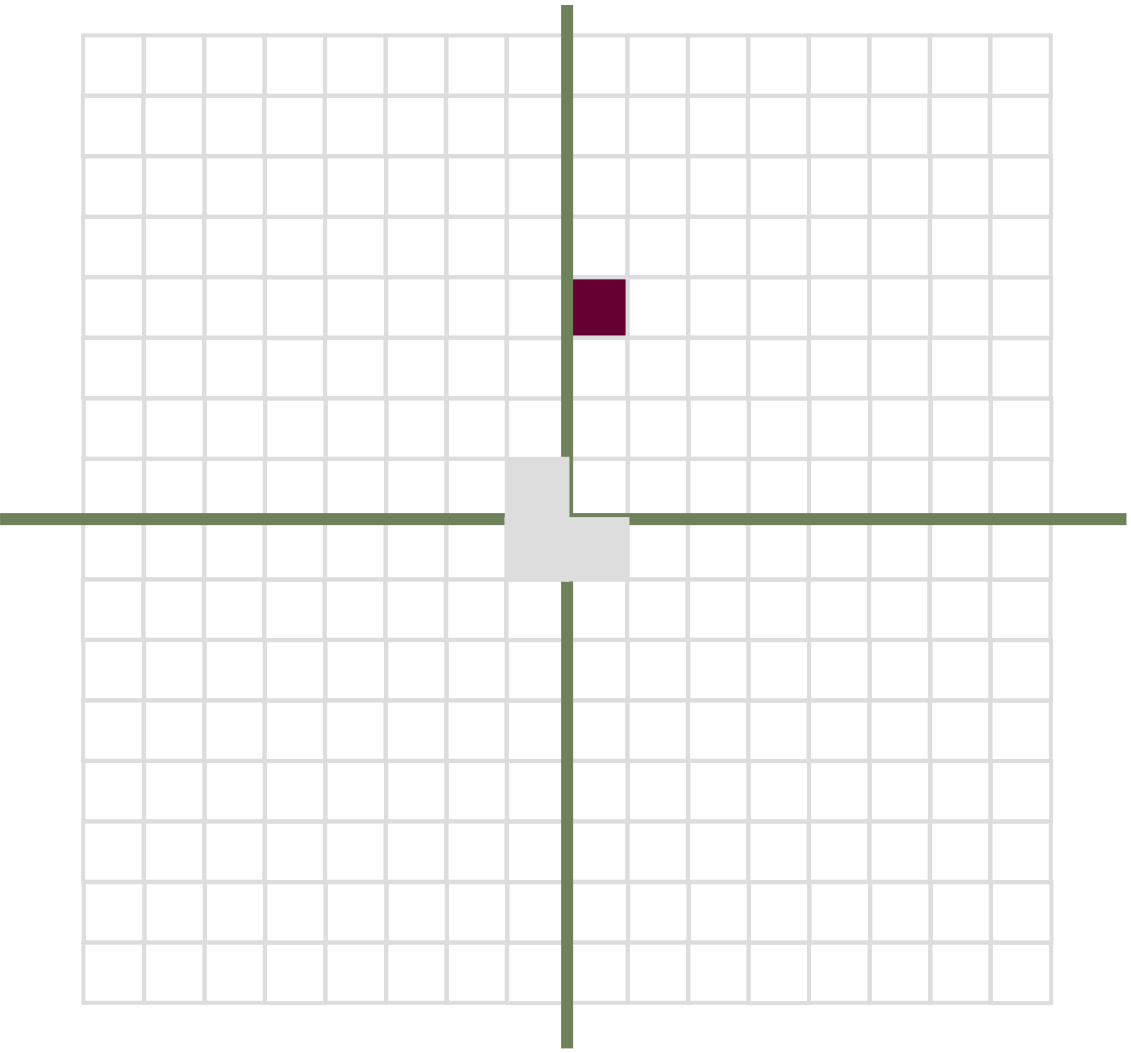
算法——棋盘覆盖

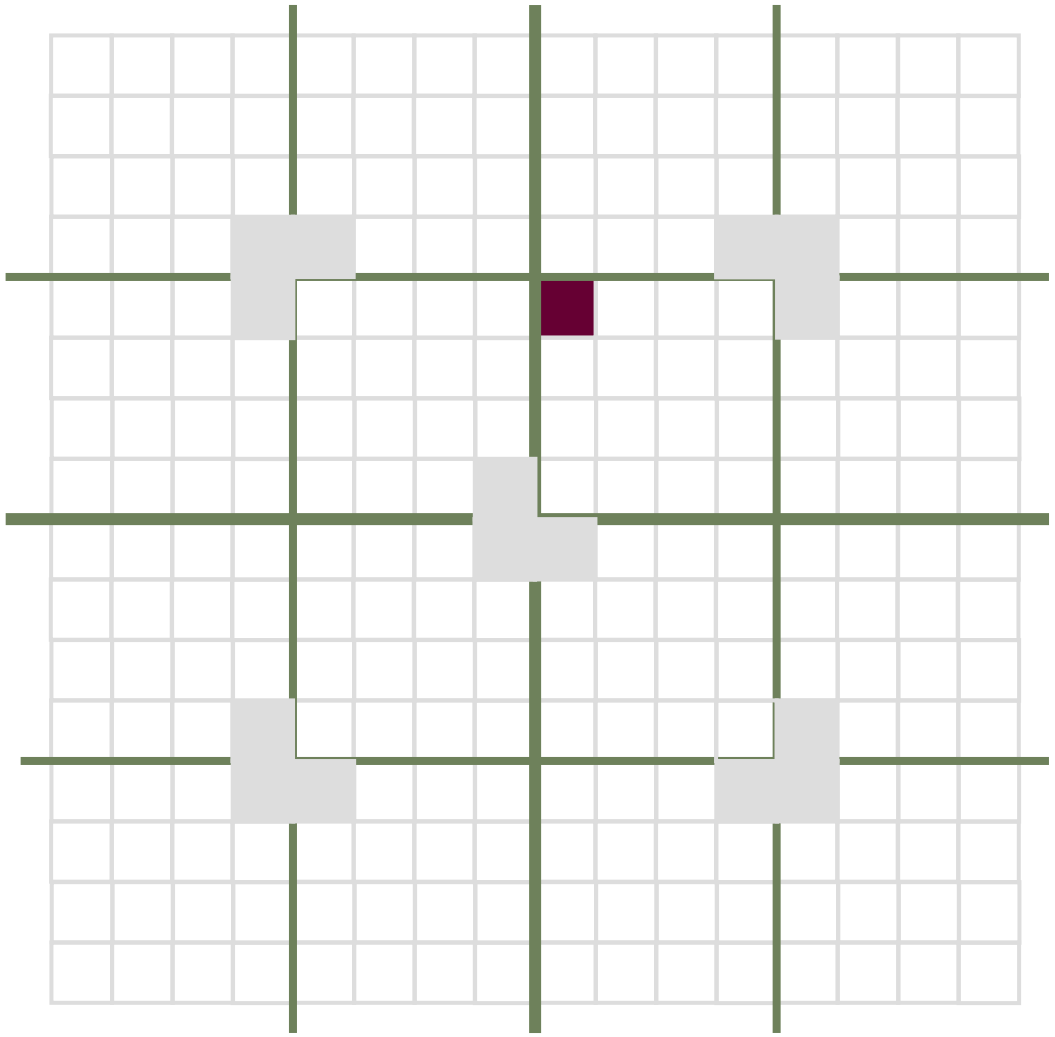
```
void  ChessBoard(int tr, int tc, int dr, int dc, int size)
// tr和tc是棋盘左上角的下标，dr和dc是特殊方格的下标，
// size是棋盘的大小，t已初始化为0
{
    if (size = = 1) return;  //棋盘只有一个方格且是特殊方格
    t++;  // L型骨牌号
    s = size/2;  // 划分棋盘
    // 覆盖左上角子棋盘
    if (dr < tr + s && dc < tc + s)   // 特殊方格在左上角子棋盘中
      ChessBoard(tr, tc, dr, dc, s);          //递归处理子棋盘
    else{      // 用 t 号L型骨牌覆盖右下角，再递归处理子棋盘
     board[tr + s - 1][tc + s - 1] = t;
     ChessBoard(tr, tc, tr+s-1, tc+s-1, s);
    }
```

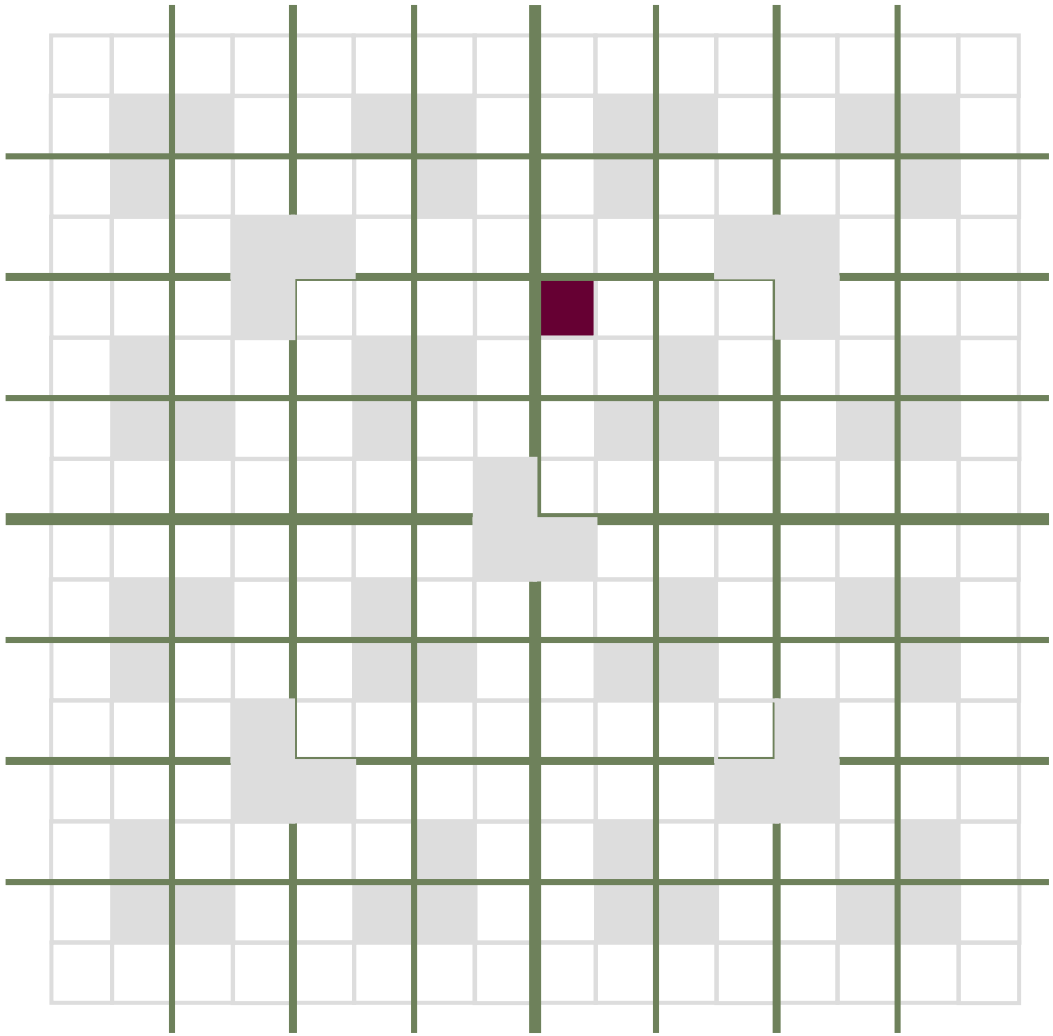
```
    // 覆盖右上角子棋盘
    if (dr < tr + s && dc >= tc + s)    // 特殊方格在右上角子棋盘中
      ChessBoard(tr, tc+s, dr, dc, s);          //递归处理子棋盘
    else {       // 用 t 号L型骨牌覆盖左下角，再递归处理子棋盘
      board[tr + s - 1][tc + s] = t;
      ChessBoard(tr, tc+s, tr+s-1, tc+s, s); }
    // 覆盖左下角子棋盘
    if (dr >= tr + s && dc < tc + s)   // 特殊方格在左下角子棋盘中
      ChessBoard(tr+s, tc, dr, dc, s);         //递归处理子棋盘
    else {       // 用 t 号L型骨牌覆盖右上角，再递归处理子棋盘
      board[tr + s][tc + s - 1] = t;
      ChessBoard(tr+s, tc, tr+s, tc+s-1, s); }
    // 覆盖右下角子棋盘
    if (dr >= tr + s && dc >= tc + s)  // 特殊方格在右下角子棋盘中
      ChessBoard(tr+s, tc+s, dr, dc, s);       //递归处理子棋盘
    else {       // 用 t 号L型骨牌覆盖左上角，再递归处理子棋盘
      board[tr + s][tc + s] = t;
      ChessBoard(tr+s, tc+s, tr+s, tc+s, s); }
}
```

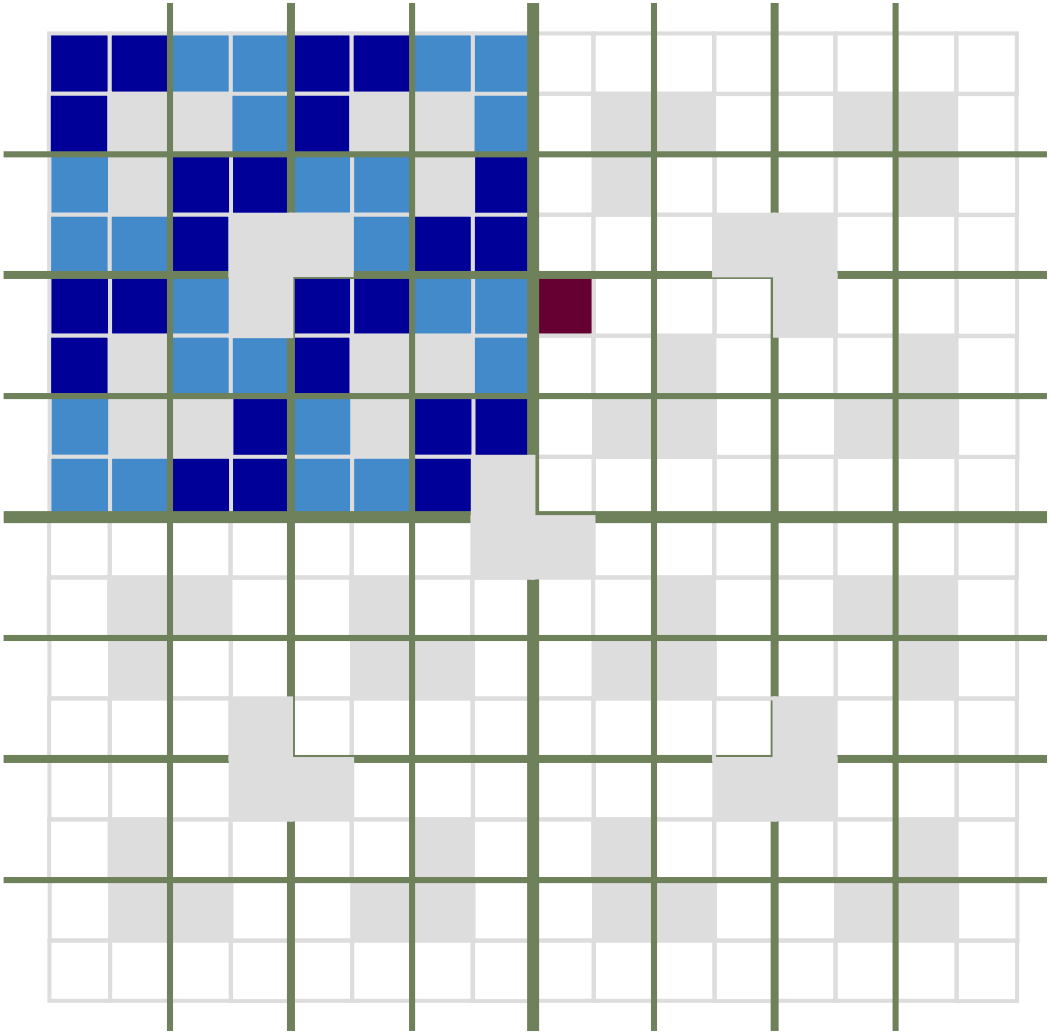


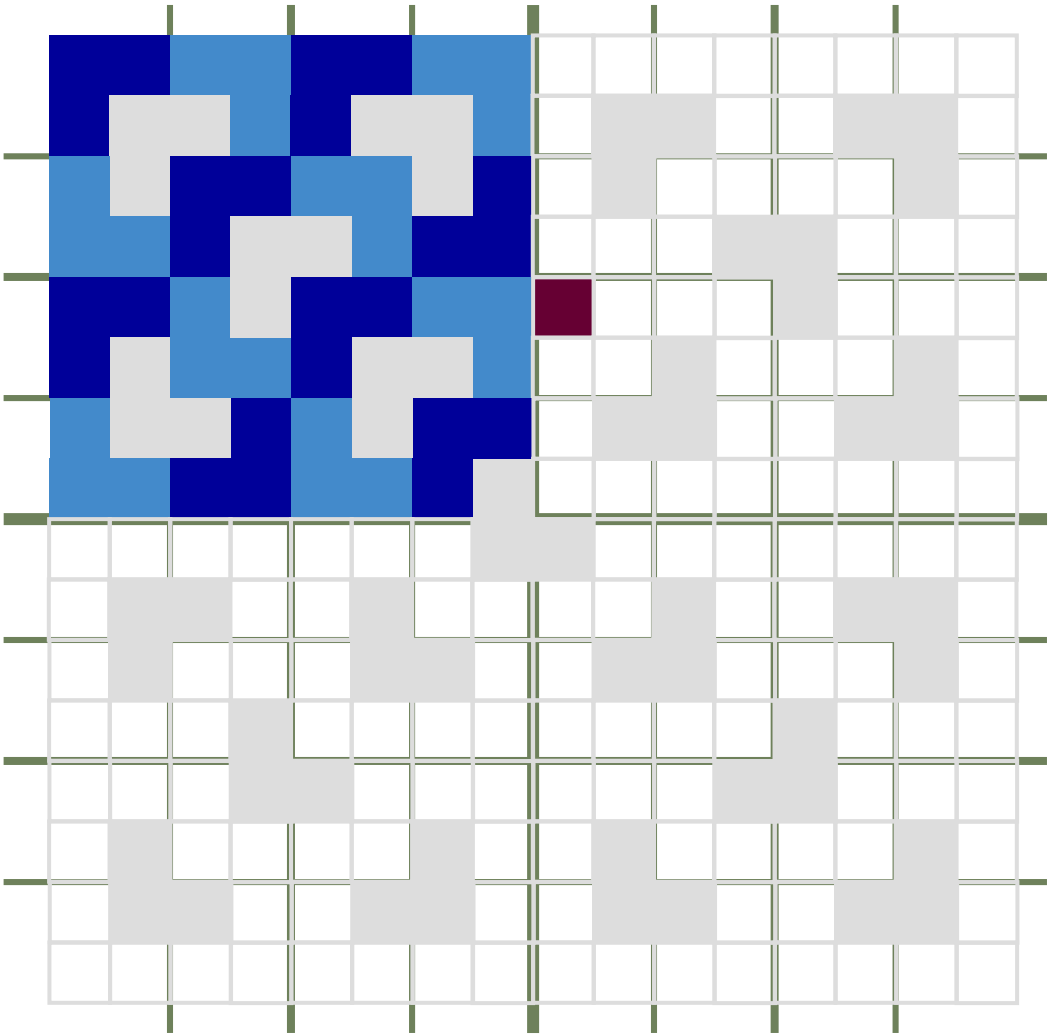












问题描述：在一个2^k×2^k 个方格组成的棋盘中，恰有一个方格与其它方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。

算法思想：

用分治策略，可以设计出解棋盘覆盖问题的简洁算法。

(1)当k>0时，将2的k次幂乘以2的k次幂棋盘分割为4个2的k-1次幂乘以2的k-1次幂子棋盘。

(2)特殊方格必位于4个较小棋盘之一中，其余3个子棋盘中无特殊方格。

(3)为了将这3个无特殊方格的子棋盘转化为特殊棋盘，可以用一个L型骨牌覆盖这3个较小棋盘的会合处，这3个子棋盘上被L型骨牌覆盖的方格就成为该棋盘上的特殊方格，从而将原问题转化为4个较小规模的棋盘覆盖问题。递归地使用这种分割，直至棋盘简化为1*1棋盘。

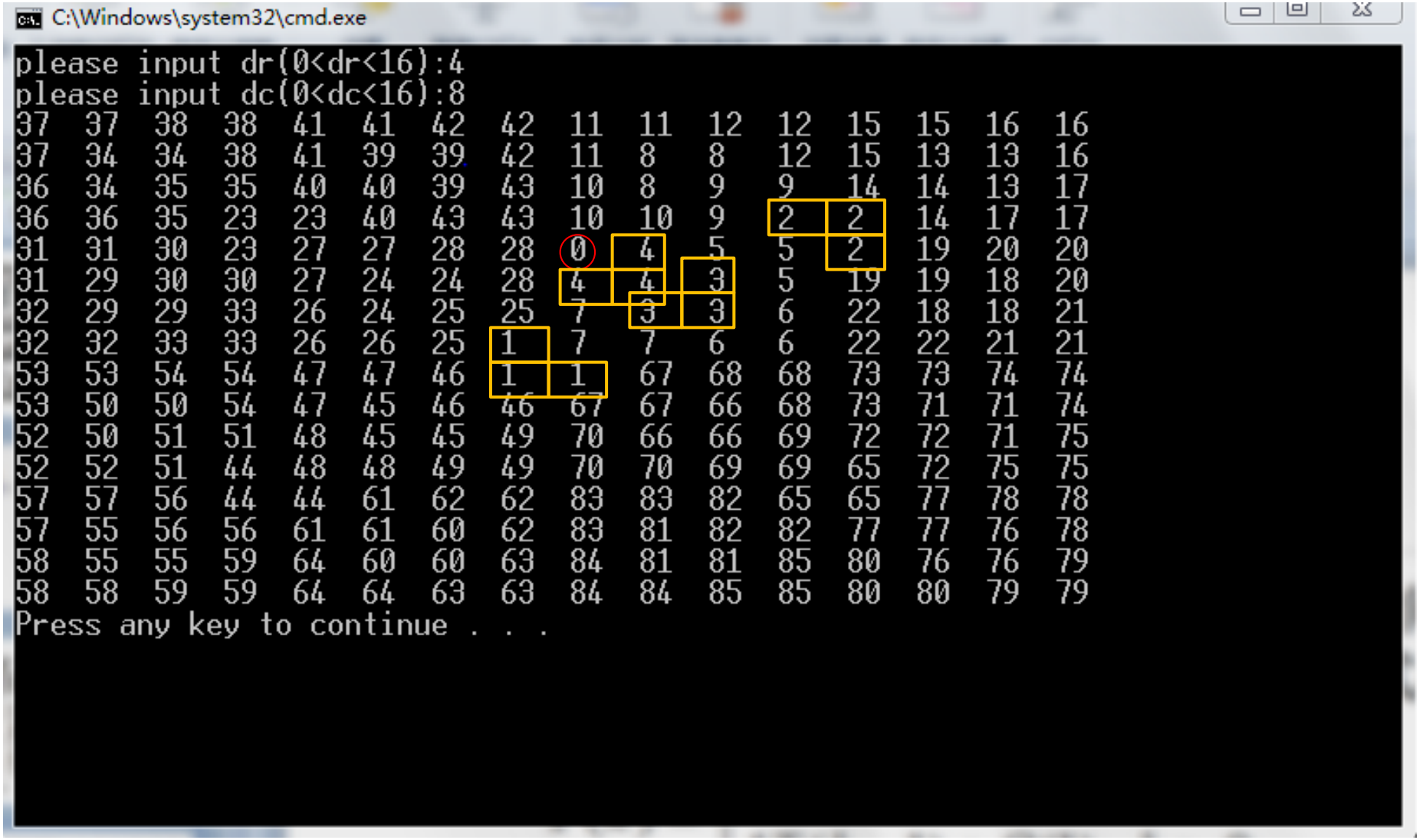
tr:棋盘左上角方格的行号
tc:棋盘左上角方格的列号
dr:特殊方格所在的行号
dc:特殊方格所在的列号
size:方形棋盘的边长

```
#include <stdio.h>
#include <stdlib.h>
#define N 16
int a[100][100];
int t=1;
void Tromino(int (*a)[N],int dr,int dc,int tr,int tc,int size)
{
  int s;
  if(size==1) return;
  if(size>1)
  {
     s=size/2;
     if(dr<=(tr+s-1)&&dc<=(tc+s-1))        /*特殊方块在左上部分*/
     {
         a[tr+s-1][tc+s]=t;
         a[tr+s][tc+s]=t;
         a[tr+s][tc+s-1]=t;
         t++;
         Tromino(a,dr,dc,tr,tc,s);//左上角子棋盘递归处理
         Tromino(a,tr+s-1,tc+s,tr,tc+s,s);//右上角子棋盘递归处理
         Tromino(a,tr+s,tc+s,tr+s,tc+s,s);//右下角子棋盘递归处理
         Tromino(a,tr+s,tc+s-1,tr+s,tc,s);//左下角子棋盘递归处理
     }
```

```
if(dr<=(tr+s-1)&&dc>(tc+s-1))        /*特殊方块在右上部分*/
    {
       a[tr+s-1][tc+s-1]=t;
       a[tr+s][tc+s-1]=t;
       a[tr+s][tc+s]=t;
       t++;
       Tromino(a,dr,dc,tr,tc+s,s); //右上角子棋盘递归处理
       Tromino(a,tr+s-1,tc+s-1,tr,tc,s); //左上角子棋盘递归处理
       Tromino(a,tr+s,tc+s-1,tr+s,tc,s); //左下角子棋盘递归处理
       Tromino(a,tr+s,tc+s,tr+s,tc+s,s); //右下角子棋盘递归处理
     }
    if(dr>(tr+s-1)&&dc<=(tc+s-1))  /*特殊方块在左下部分*/
    {
       a[tr+s-1][tc+s-1]=t;
       a[tr+s-1][tc+s]=t;
       a[tr+s][tc+s]=t;
       t++;
       Tromino(a,dr,dc,tr+s,tc,s);
       Tromino(a,tr+s-1,tc+s-1,tr,tc,s);
       Tromino(a,tr+s-1,tc+s,tr,tc+s,s);
       Tromino(a,tr+s,tc+s,tr+s,tc+s,s);
    }
```

```
if(dr>(tr+s-1)&&dc>(tc+s-1))        /*特殊方块在右下部分*/
     {
        a[tr+s][tc+s-1]=t;
        a[tr+s-1][tc+s-1]=t;
        a[tr+s-1][tc+s]=t;
        t++;
        Tromino(a,dr,dc,tr+s,tc+s,s);
        Tromino(a,tr+s,tc+s-1,tr+s,tc,s);
        Tromino(a,tr+s-1,tc+s-1,tr,tc,s);
        Tromino(a,tr+s-1,tc+s,tr,tc+s,s);
     }
  }
}
```

```
int main()
{
  int i,j,dr,dc, a[N][N];
  printf("please input dr(0<dr<%d):",N);
  scanf("%d",&dr);
  printf("please input dc(0<dc<%d):",N);
  scanf("%d",&dc);
  a[dr][dc]=0;
  Tromino(a,dr,dc,0,0,N);
  for(i=0;i<N;i++)
  {
      for(j=0;j<N;j++)
        printf("%-4d",a[i][j]);
      printf("\n");
  }
  system("pause");
  return 0;
}
```


C:\Windows\system32\cmd.exe
please input dr(0<dr<16):4
please input dc(0<dc<16):8
37 37 38 38 41 41 42 42 11 11 12 12 15 15 16 16
37 34 34 38 41 39 39 42 11 8 8 12 15 13 13 16
36 34 35 35 40 40 39 43 10 8 9 9 14 14 13 17
36 36 35 23 23 40 43 43 10 10 9 2 2 14 17 17
31 31 30 23 27 27 28 28 0 4 5 5 2 19 20 20
31 29 30 30 27 24 24 28 4 4 3 5 19 19 18 20
32 29 29 33 26 24 25 25 7 3 3 6 22 18 18 21
32 32 33 33 26 26 25 1 7 7 6 6 22 22 21 21
53 53 54 54 47 47 46 1 1 67 68 68 73 73 74 74
53 50 50 54 47 45 46 46 67 67 66 68 73 71 71 74
52 50 51 51 48 45 45 49 70 66 66 69 72 72 71 75
52 52 51 44 48 48 49 49 70 70 69 69 65 72 75 75
57 57 56 44 44 61 62 62 83 83 82 65 65 77 78 78
57 55 56 56 61 61 60 62 83 81 82 82 77 77 76 78
58 55 55 59 64 60 60 63 84 81 81 85 80 76 76 79
58 58 59 59 64 64 63 63 84 84 85 85 80 80 79 79
Press any key to continue . . .

设$T(k)$是覆盖一个$2k\times 2k$棋盘所需时间，从算法的划分策略可知，$T(k)$满足如下递推式：

$$T(k)=\begin{cases} O(1) & k=0 \\ 4T(k-1)+O(1) & k>0 \end{cases}$$

解：$T(k)=4T(k-1)+O(1)=\cdots$

$$=4^kT(0)+O(1)\sum_{i=0}^{k-1}4^i=4^kO(1)+O(1)(4^k-1)/3$$

$$T(k)=O(4^k)$$

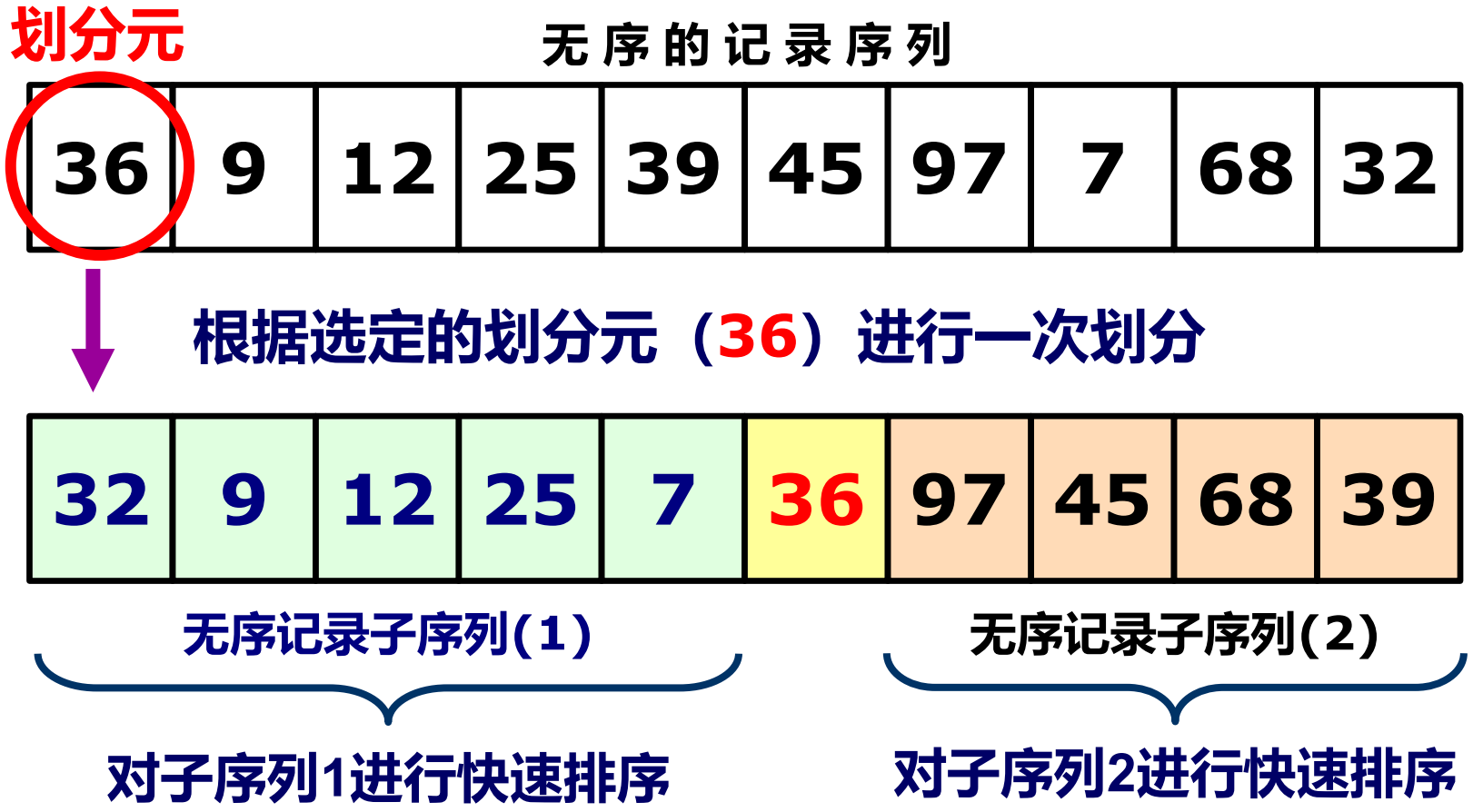
分治法示例5：快速排序（Quick Sort）

算法基本思想

- 在数组中确定一个记录（的关键字）作为“划分元”
- 将数组中关键字小于划分元的记录均移动至该记录之前
- 将数组中关键字大于划分元的记录均移动至该记录之后
- 由此：一趟排序之后，序列R[s...t]将分割成两部分
  - R[s ... i-1] 和 R[i+1 ... t]
  - 且满足：R[s ... i-1]≤ R[i]≤ R[i+1...t]
  - 其中：R[i] 为选定的“划分元”
- 对各部分重复上述过程，直到每一部分仅剩一个记录为止

快速排序（Quick Sort）

- 首先对无序的记录序列进行一次划分
- 之后分别对分割所得两个子序列“递归”进行快速排序



快速排序算法流程

- 首先：设 R[s]=36 为划分元，将其暂存到R[0]
- 比较 R[high] 和划分元的大小，要求：R[high] ≥ 划分元
- 比较 R[low] 和划分元的大小，要求：R[low] ≤ 划分元
- 若条件不满足，则交换元素，并在low-high之间进行切换
- 一轮划分后得到：（32,9,12,25,7）36（97,45,68,39）

快速排序

递归的快速排序算法

```
 template<class Type>
void QuickSort (Type a[], int p, int r)
{
      if (p<r) {
        int q=Partition(a,p,r);
        QuickSort (a,p,q-1); //对左半段排序
        QuickSort (a,q+1,r); //对右半段排序
        }
}
```

在快速排序中，记录的比较和交换是从两端向中间进行的，关键字较大的记录一次就能交换到后面单元，关键字较小的记录一次就能交换到前面单元，记录每次移动的距离较大，因而总的比较和移动次数较少。

快速排序

一趟快速排序算法的实现：

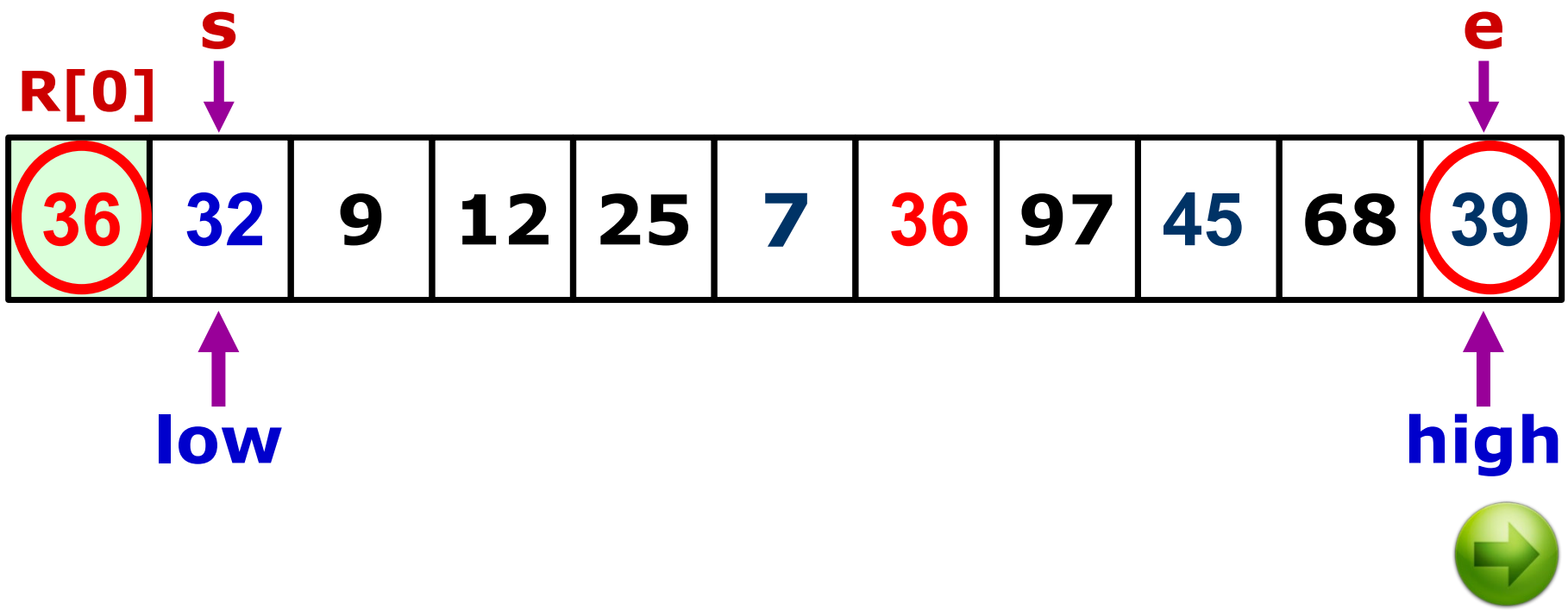
```
template<class Type>
int Partition (Type a[], int p, int r)
{
      int i = p, j = r + 1;
      Type x=a[p];
      // 将< x的元素交换到左边区域
      // 将> x的元素交换到右边区域
      while (true) {
        while (a[- -j] >=x) ;
        Swap(a[i], a[j]); //交换
        while (a[++i] <=x && i<r );
        if (i >= j) break;
        Swap(a[i], a[j]); //交换
        }
       return j;
}
```

快速排序

一趟快速排序算法的实现（改进）

```
PROC qkpass(VAR r:listtype; s,t:integer;  VAR i:integer);
{    对r[s..t]进行一趟快速排序,i将r[s..t]分为两部分;
      r[s..i-1]的关键字不大于r[i].key, r[i+1..t]的关键字不小于r[i].key  }
   i:=s; j:=t; rp:=r[s]; x:=r[s].key;
   WHILE  i<j  DO
   [  WHILE  (i<j) AND (r[j].key≥x) DO j:=j-1;
      r[i]:=r[j];   { 赋值 }
      WHILE  (i<j) AND (r[i].key≤x) DO i:=i+1;
      r[j]:=r[i]    { 赋值 }
   ];
    r[i]:=rp
ENDP;  {qkpass}
```

一趟快速排序算法的实现（改进）



快速排序算法特点

- 时间复杂度
  - 最好情况
    - $T(n)=O(n \log_2 n)$ （每次总是选到中间值作划分元）
  - 最坏情况
    - $T(n)=O(n^2)$ （每次总是选到最小或最大元素作划分元）

快速排序算法特点

- 算法性能与序列中关键字的排列顺序和划分元的选取有关
  - 当初始序列按关键字有序（正序或逆序）时，快速排序蜕化为冒泡排序，此时算法性能最差
    - 时间复杂度为$O(n^2)$
  - 可以用“三者取中”法来选取划分元
    - 设：数组首记录为r[s]、尾记录为r[t]
    - 取：r[s]、r[t]和r[(s+t)/2]三者的中间值为划分元
  - 也可采用随机选取划分元的方式

快速排序—随机

•快速排序算法的性能取决于划分的对称性。通过修改算法partition，可以设计出采用随机选择策略的快速排序算法。

•在快速排序算法的每一步中，当数组还没有被划分时，可以在a[p:r]中随机选出一个元素作为划分基准，这样可以使划分基准的选择是随机的，从而可以期望划分是较对称的。

```
template<class Type>
int RandomizedPartition (Type a[], int p, int r)
{
        int i = Random(p,r);
        Swap(a[i], a[p]);
        return Partition (a, p, r);
}
```

分治法示例6：线性时间选择

- 元素选择问题描述
  - 给定线性序集中n个元素和一个整数k（1≤k≤n）
  - 要求找出这n个元素中**第k小**的元素
- 问题分析
  - 方法一：可以通过排序求解元素选择问题： $O(n\log_2 n)$
  - 问题：元素选择问题是否可以在O(n)时间内得到解决?
  - 可以采用分治算法：模仿快排对输入数组进行递归划分
    - 提示1：只对划分出的子数组之一进行递归处理
    - 提示2：子数组的选择与划分元和k相关

线性时间选择

```
int RandSelect(int A[], int start, int end, int k) {
    if (start ==end) return A[start];
    int i= RandomizedPartition(A, start, end); //划分元位置i
    int n=i-start+1;  // 左子数组A[start :i]的元素个数
    if (k<=n)
        return RandSelect (A, start, i, k);
    else
        return RandSelect(A, i+1, end, k-n);
}
```

平均时间复杂度与n呈线性关系，为O(n)(数学证明过程略过,可参考论文:《线性时间选择算法时间复杂度深入研究》by 王云鹏)。

但可以证明，算法运行的平均时间为O(n)

线性时间选择

- 算法改进：是否可以在**最坏情况**下用O(n)时间就完成选择?
- 问题分析
  - 如果能在线性时间内找到一个划分基准，使得划分得到的两个子数组的长度都至少为原数组长度的ε倍(0<ε<1)
  - 那么就可以在最坏情况下用O(n)时间完成选择任务。
- 例如
  - 若ε=0.9：每次划分得到的子数组的长度至少缩短1/10
  - 则最坏情况下算法的计算时间：$T(n) \leq T(0.9n)+O(n)$
  - 由此可得：$T(n)=O(n)$

线性时间选择

- 线性时间内找到合理划分基准的步骤（select函数）
  - 将n个输入元素划分成⌈n/5⌉个组，每组5个元素
    - 则只可能有一个组不是5个元素
  - 用任意一种排序算法对每组中的元素排序
  - 然后取出每组的中位数，共⌈n/5⌉个元素
  - 递归调用select函数找出这⌈n/5⌉个元素的中位数
  - 如果⌈n/5⌉为偶数，则选择2个中位数中较大的一个
  - 以这个选出的元素作为划分基准

例子

- **按递增顺序，找出下面29个元素的第18个元素：8,31,60,33,17,4,51,57,49,35,11,43,37,3,13,52,6,19,25,32,54,16,5,41,7,23,22,46,29.**

  (1) 把前面25个元素分为6(=ceil(29/5))组；(8,31,60,33,17),(4,51,57,49,35),(11,43,37,3,13),(52,6,19,25,32),(54,16,5,41,7),(23,22,46,29).

  (2) 提取每一组的中值元素，构成集合{31,49,13,25,16,29}；

  (3) 递归地使用算法求取该集合的中值，得到m=29；

  (4) 根据m=29, 把29个元素划分为3个子数组：

  - P={8,17,4,11, 3,13,6,19, 25,16,5,7,23,22}
  - Q={29}
  - R={31,60,33,51,57,49,35,43,37,52,32,54,41,46}

例子（续）

(5) 由于|P|=14,|Q|=1,k=18，所以放弃P,Q，使k=18-14-1=3，对R递归地执行本算法；

(6) 将R划分成3(ceil(14/5))组：{31,60,33,51,57},{49,35,43,37,52},{32,54,41,46}

(7) 求取这3组元素的中值元素分别为：{51,43,46}，这个集合的中值元素是43;

(8) 根据43将R划分成3组：

{31, 33, 35,37,32, 41},{43},{60, 51,57, 49, 52,54, 46}

例子（续）

(9) 因为k=3，第一个子数组的元素个数大于k，所以放弃后面两个子数组，以k=3对第一个子数组递归调用本算法；

(10) 将这个子数组分成5个元素的一组：{31,33,35,37,32}、{41}，取其中值元素为33；

(11) 根据33，把第一个子数组划分成{31,32},{33},{35,37};

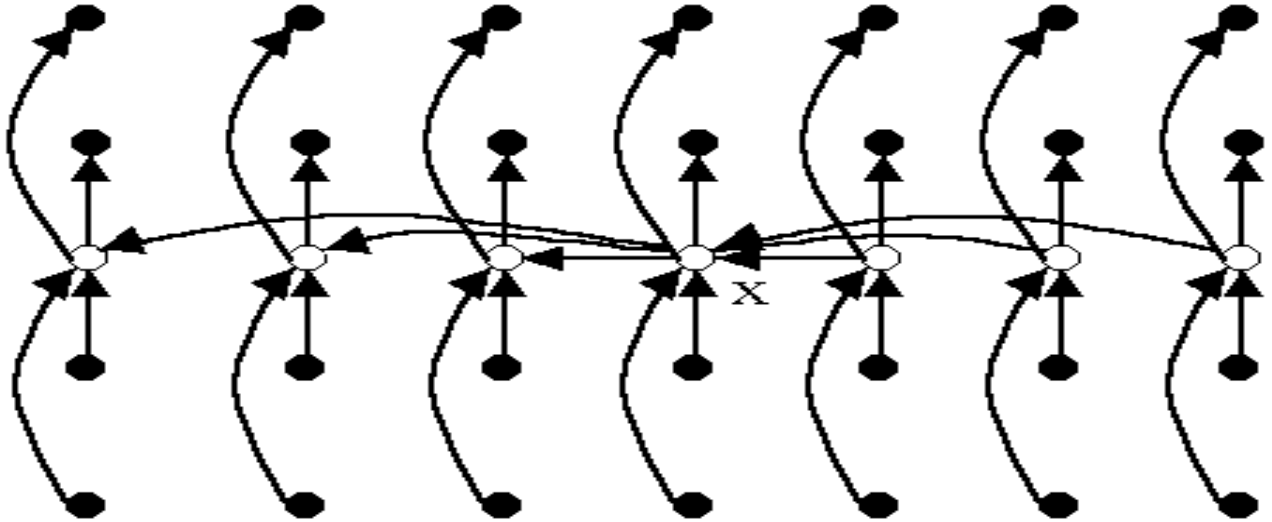
(12) 因为k=3,而第一、第二个子数组的元素个数为3，所以33即为所求取的第18个小元素。

线性时间选择

```
int Select(int a[], int start, int end, int k) {
    if (end-start<70) {
        用某个简单排序算法对数组a[start:end]排序;
        return a[start+k-1];
    }
    for ( int i = 0; i<=(end-start-4)/5; i++ ) {
        将a[start+5*i]至a[start+5*i+4]的第3小元素与a[start+i]交换位置;
    } //找中位数的中位数，end-start-4即n-5
    int x = Select(a, start, start+(end-start-4)/5, (end-start-4)/10);
    int i = Partition(a, start, end, x); // 划分元位置
    int n = i-start+1; // 划分元左边的数组长度
    if (k<=n) return Select(a, start, i, k); // 在左边寻找第k大元素
    else return Select(a, i+1, end, k-n);   // 在右边寻找第k-j大元素
}
```

线性时间选择

- 线性时间内找到合理划分基准的步骤（select函数）



- 设所有元素互不相同。在这种情况下，找出的**基准x至少比3⌊(n-5)/10⌋个元素大**，因为在每一组中有2个元素小于本组的中位数，而⌊n/5⌋个中位数中至少又有⌊(n-5)/10⌋个小于基准x。同理，**基准x也至少比3⌊(n-5)/10⌋个元素小**
- 当n≥75时， $3\lfloor (n-5)/10 \rfloor \geq n/4$
- 所以按此基准划分所得的2个子数组的长度都至少缩短1/4

```
Type Select(Type a[], int p, int r, int k)
{
    if (r-p<75) {
      用某个简单排序算法对数组a[p:r]排序;
      return a[p+k-1];
      };
    for ( int i = 0; i<=(r-p-4)/5; i++ )
       将a[p+5*i]至a[p+5*i+4]的第3小元素与a[p+i]交换位置; //每组的中位数从第3个位置交换到第1个位置
    Type x = Select(a, p, p+(r-p-4)/5, (r-p-4)/10);  //找中位数的中位数，r-p-4即上面所说的n-5
    int i=Partition(a,p,r, x),
    j=i-p+1;
    if (k<=j) return Select(a,p,i,k);
    else return Select(a,i+1,r,k-j);
}
```

线性时间选择

- 算法复杂度分析
  - 当n<75时，算法所用的计算时间不超过一个常数C_1
  - 分组求中位数的for循环执行时间为O(n)
  - 以中位数x为基准对数组进行划分，需要O(n)时间
  - 设：对n个元素的数组调用select需要T(n)时间
    - & 则找出中位数的中位数x，至多需要T(n/5)时间
    - & 已证明划分得到的子数组长度不超过3n/4
    - & 无论对哪个子数组调用select至多需T(3n/4)时间

线性时间选择

- 算法复杂度分析：T(n)=O(n)

$$T(n) \le \begin{cases} C_1 & n < 75 \\ O(n) + T(n/5 + 3n/4) & n \ge 75 \end{cases}$$

- 算法分析
  - 分组大小固定为5，以75作为是否递归调用的分界点
  - 这两点保证了T(n)的递归式中2个自变量之和 n/5+3n/4=19n/20=εn，0<ε<1
  - 这是使T(n)=O(n)的关键之处
  - 当然，除了5和75之外，还有其他选择……

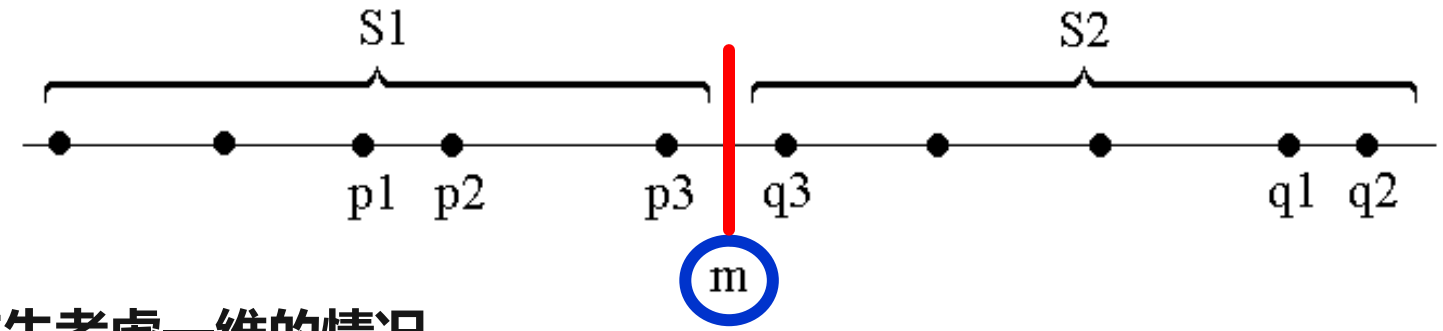
分治法示例7：最接近点对问题

- 计算机应用中经常采用点、圆等简单的几何对象表示物理实体，常需要了解其邻域中其他几何对象的信息
  - 例如：在空中交通控制中，若将飞机作为空间中的一个点来处理，则具有最大碰撞危险的两架飞机，就是这个空间中距离最接近的一对点
  - 这类问题是计算几何学中研究的基本问题之一
  - 我们着重考虑平面上的最接近点对问题
- 最接近点对问题：给定平面上的n个点，找出其中的一对点，使得在n个点组成的所有点对中，该点对的距离最小

求解最接近点对问题

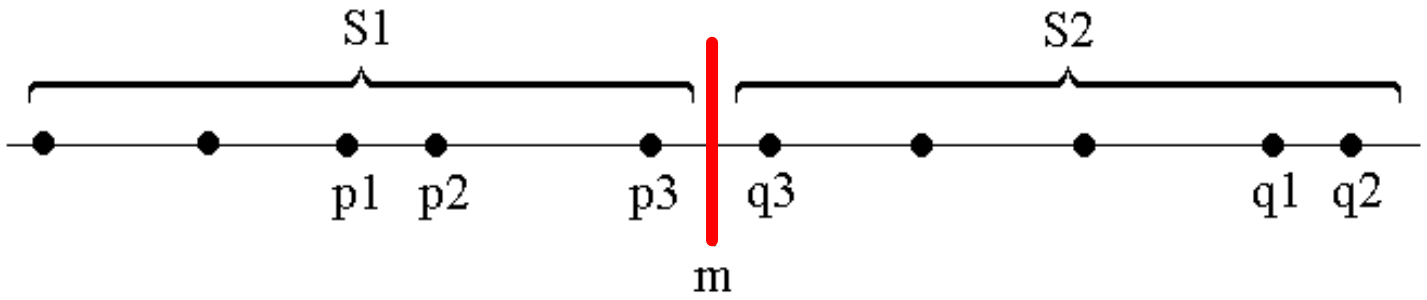
- 直观解法
  - 将每一个点与其他n-1个点的距离算出，找出最小距离
  - 时间复杂度： $T(n)=n(n-1)/2+n=O(n^2)$
- 分治法
  - 分解：将n个点的集合分成大小近似相等的两个子集
  - 求解：递归地求解两个子集内部的最接近点对
  - 合并（关键问题）：从子空间内部最接近点对，和两个子空间之间的最接近点对中，选择最接近点对

分治法求解最接近点对问题



- 首先考虑一维的情况
  - 此时，S中的n个点退化为x轴上的n个实数 $x_1, x_2, \ldots, x_n$
  - 最接近点对即为这n个实数中相差最小的2个实数
  - 思考：排序可以在O(nlogn)时间解决问题　缺点：难以推广到高维
- 分治法
  - 假设我们用x轴上某个点m将S划分为2个子集S1和S2
    - 基于平衡子问题的思想，用S中各点坐标的中位数来作分割点
  - 递归地在S1和S2上找出其最接近点对{p1,p2}和{q1,q2}
  - 设d=min{|p1-p2|,|q1-q2|}，则S中的最接近点对或者是{p1,p2}，或者是{q1,q2}，或者是某个{p3,q3}，其中p3∈S1且q3∈S2

分治法求解最接近点对问题



- **如果S的最接近点对是{p3,q3}，即|p3-q3|<d**
  - 则p3和q3两者与m的距离不超过d
  - 即：p3∈(m-d, m]，q3∈(m, m+d]
- **问题分析** **分割点m的选取应使得划分出的子集尽可能平衡**
  - 在S1中每个长度为d的半闭区间至多包含一个点
  - 由于m是S1和S2的分割点，因此(m-d,m]中至多包含S中的一个点
  - 如果(m-d,m]中有S中的点，则此点就是S1中最大点（S2同理）
  - 因此用线性时间可找到(m-d, m]和(m, m+d]中所有点（即p3和q3）
  - 所以，用线性时间就可以将S1的解和S2的解合并成为S的解
  - 思考：影响算法性能的主要因素是什么？

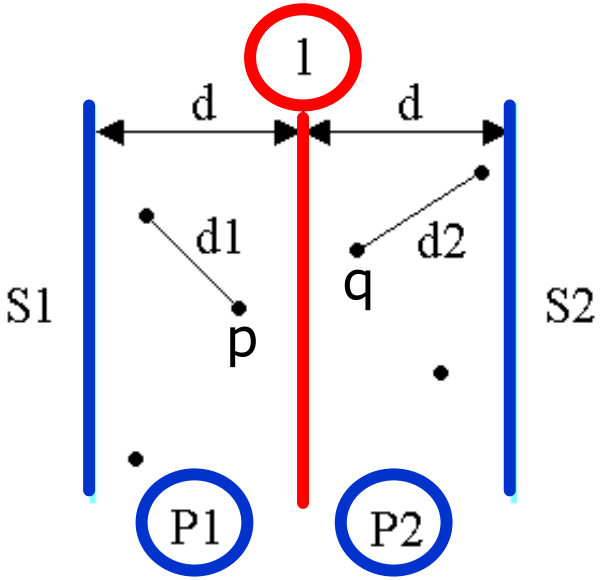
求一维点集S的最接近点对的算法

```
bool Cpair1(S,d)
{  n=|S|;
   if(n<2) { d=∞; return false; }
   m=S中各点坐标的中位数;
   构造S1和S2;
   //S1={x ∈ S|x<=m}, S2={x ∈ S|x>m}
   Cpair1(S1,d1);
   Cpair1(S2,d2);
   p=max(S1);
   q=min(S2);
   d=min(d1,d2,q-p);
   return true;
}
```

T(n)=O(nlogn)

复杂度分析： $T(n)=\begin{cases} O(1) & n<2 \\ 2T(n/2)+O(n) & n\geq 2 \end{cases}$

分治法求解最接近点对问题

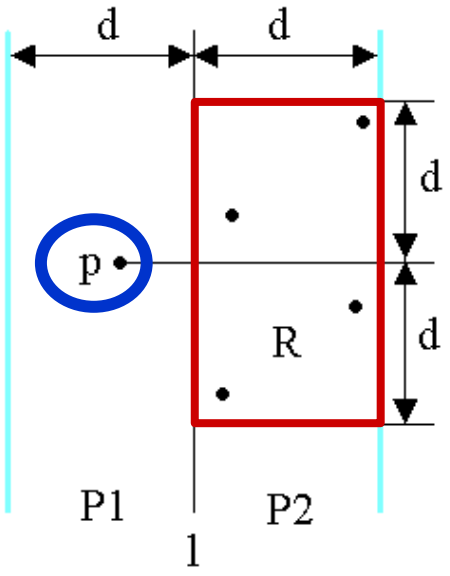


- **考虑二维的情况**
  - 选取二维平面的一条垂直线**L：x=m** 作为分割线
    - 其中m为S中各点**x坐标的中位数**，由此将S分割为S1和S2
  - 递归地在S1和S2上找出其最小距离d1和d2
  - 设：**d=min{d1,d2}**，S中的最接近点对间的距离或者是d，或者是某个点对**{p,q}**之间的距离，其中p∈S1且q∈S2
  - 如果用符号P1和P2分别表示直线 **L** 的左右两边宽为d的区域
    - 则必有p∈P1且q∈P2

分治法求解最接近点对问题

问题分析

- 考虑P1中任意一点p
  - 它若与P2中的点q构成最接近点对的候选者
  - 则必有：distance(p，q) < d
  - P2中满足条件的点一定落在矩形R中
  - 矩形R的大小为：d×2d
- 由d的定义可知：P2中任何2个点（$\mathbf{q}_i \in \mathbf{S}$）的距离都不小于d
  - 由此可以推出矩形R中最多只有6个S中的点（证明见下页）
- 因此，在分治法的**合并**步骤中最多只需要检查6×n/2=3n个候选者

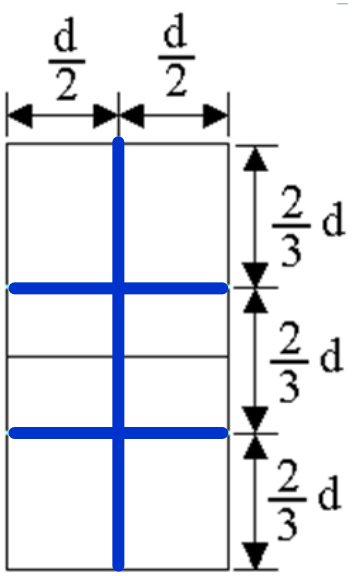


分治法求解最接近点对问题

- **证明：合并时最多只需检查6×n/2=3n个候选者**
  - 将矩形R长为2d的边3等分，将长为d的边2等分
  - 由此导出6个(d/2)×(2d/3)的小矩形，如图
  - 若矩形R中有多于6个S中的点，由鸽笼原理易知
  - 至少有一个小矩形中有2个以上S中的点
  - 设u，v是位于同一小矩形中的2个点，则

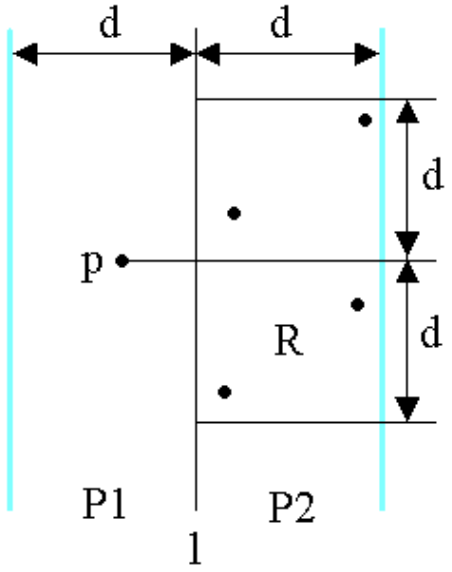
$$(x(u)-x(v))^2+(y(u)-y(v))^2 \le (d/2)^2+(2d/3)^2 = \frac{25}{36}d^2$$

  - 即：distance(u,v)<d，这与d的意义相矛盾，命题得证。



分治法求解最接近点对问题

- **问题：如何确定需要检查的6个点?**
  - 可以将**p**和**P2**中所有S2的点投影到垂直线l上
  - 由于能与**p**点一起构成最接近点对候选者的S2中的点一定在矩形R中，所以它们在直线 **L** 上的投影点与 **p** 在 **L** 上投影点的距离小于d
  - 根据上述分析，这种投影点最多只有6个
  - 因此，若将区域P1和P2中所有S中的点按其y坐标排好序
  - 则：对P1中的所有点，只需一次扫描就可以找出所有候选者
    - 对排好序的点作一次扫描，可以找出所有最接近点对的候选者
    - 对P1中每个点，最多只需检查P2中排好序的相继6个点



最接近点对问题

```
double cpair2(S)
{
    n=|S|;
    if (n < 2) return …;
1、  m=S中各点x间坐标的中位数;
    构造S1和S2;
    //S1={p∈S|x(p)<=m},
   S2={p∈S|x(p)>m}
2、  d1=cpair2(S1);
    d2=cpair2(S2);
3、  dm=min(d1,d2);
```

4、设P1是S1中距垂直分割线l的距离在dm之内的所有点组成的集合；

P2是S2中距分割线l的距离在dm之内所有点组成的集合；

将P1和P2中点依其y坐标值排序；

并设X和Y是相应的已排好序的点列；

5、通过扫描X以及对于X中每个点检查Y中与其距离在dm之内的所有点(最多6个)可以完成合并；

当X中的扫描指针逐次向上移动时，Y中的扫描指针可在宽为2dm的区间内移动；

设dl是按这种扫描方式找到的点对间的最小距离；

```
6、  d=min(dm,dl);
    return d;
}
```

分治法求解最接近点对问题

- 算法复杂度分析

$$T(n)=\begin{cases} O(1) & n<4 \\ 2T(n/2)+O(n) & n\geq 4 \end{cases}$$

思考：为什么是n/2？

因为每次分割是根据子空间中点的x坐标的中位数进行划分

思考：为什么是O(n)？

因为在每个划分的层次上都要对全部的点进行扫描

T(n)=O(nlogn)

循环赛日程表问题

- **设计一个满足以下要求的比赛日程表：**
  - 每个选手必须与其他n-1个选手各赛一次
  - 每个选手一天只能赛一次
  - 循环赛一共进行n-1天

1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

循环赛日程表问题

1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

- **分治算法策略**
  - 将所有的选手分为两半，n个选手的比赛日程表可以通过为n/2个选手设计的比赛日程表来决定
  - 递归地用对选手进行分割，直到只剩下2个选手时，只要让这2个选手进行比赛就可以了

循环赛日程表问题

1. 只有两个选手的日程安排

1	2
2	1

解释：如果只有两个选手，那么第0列看作选手编号（我们从0开始对列编号，我们的第0列可以看作每个选手第0天在和自己打--姑且看作自己在做心态调整），第1列就是在第一天，每个选手要比赛的选手号。

循环赛日程表问题

2. 如果选手的个数为2^2=4

解释：

- 如果有4个选手，分别设计4/2=2个选手的比赛日程表，1-2选手前一天的比赛日程表如上图表格左上角的绿色子表格部分，3-4选手前一天的比赛日程表如上图表格左下角的子表格。据此，后两天的日程表可以将左上角的子表按其对应位置抄到右下角的子表，左下角的子表可以按其对应位置抄到右上角的子表。
- 表的行列均为参赛选手数$2^2 = 4$，在用分治法求行、列均为（2^2）/2的长度的子表时，首先确定左上角的子表，左下角的子表可以由左上角的子表加（2^2）/2得到

循环赛日程表问题

2.如果选手的个数为$2^3=8$

1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

解释：

- 选手人数为8时，左上角的子表是选手1至选手4的前三天的比赛日程，左下角是选手5至选手8前三天的比赛日程。据此后四天的比赛日程，就是分别将左上角子表按其对应位置抄到右下角，将左下角的子表按其对应位置抄到右上角。这样就完成了比赛日程的安排。

循环赛日程表问题

在每次迭代求解的过程中，可以看作4部分：

1）求左上角子表：左上角子表是前2^{k-1}个选手的比赛前半程的比赛日程。

2）求左下角子表：左下角子表是剩余的2^{k-1}个选手的比赛前半程比赛日程。这个子表和左上角子表的对应关系式，对应元素等于左上角子表对应元素加2^{k-1}。

3）求右上角子表：等于左下角子表的对应元素。

4）求右下角子表：等于左上角子表的对应元素。

思考题

1. 说明调用下面的swap为什么无法交换实际参数的值

```
public static void swap(int x, int y){
    int temp = x;
    x = y;
    y = temp;
}
```

- 值参数传递不能实现交换两个整数, 要想实现两个值的交换可以已数组的形式实现,代码如下:

```
//实现两个整数的交换
public class SwapInteger {
public static void swap(int a[]){
//数组传递实现交换两个整数
int t;
t = a[0];
a[0] = a[1];
a[1] = t;
}
```

作业一

1. 假设某算法在输入规模为n时的计算时间为$T(n) = 3 \times 2^n$。在某台计算机上完成该算法的时间为t秒。现有另一台计算机，其运行速度为第一台的64倍，那么在这台新机器上用同一算法在t秒内能解输入规模多大的问题？ （TXT文档）

   - 若上述算法的计算时间改进为$T(n) = n^2$，其余条件不变，则在新机器上用t秒时间能解输入规模多大的问题?
   - 若上述算法的计算时间进一步改进为$T(n) = 8$，其余条件不变，那么在新机器上用t秒时间能解输入规模多大的问题?

2. 请尝试用三者取中法完成快速排序，并编写算法与取第一个元素为枢纽的快速排序方法进行比较测试（完整源码，含测试代码）

作业一

3. 设a[0:n-1]是已经排好序的数组，请改写二分搜索算法，使得：
   - 当搜索元素x不在数组中时，返回小于x的最大元素位置i 和大于x的最小元素位置 j （完整源码，含测试代码）
   - 当搜索元素x在数组中时，返回元素x在数组中的位置
4. 实现棋盘覆盖算法（完整源码，含测试代码）
5. 实现二路归并排序算法（完整源码，含测试代码）

☞ **作业要求**

- 5道题存放于5个文本文件中，打包成一个rar文件
- 文件名为题目编号（如：1.txt，2.c，2.cpp 或2.java）
- 压缩文件格式：学号-姓名-HW1.rar