算法分析与设计

主讲教师：刘峤

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
  - 例如n!的定义就是递归的：$n! = n \times (n - 1)!$
- 为什么要用递归？
  - 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进行求解
- 递归程序的要素？
  - 递归调用：问题得到简化
  - 程序出口：结束（返回）条件

递归（recursion）

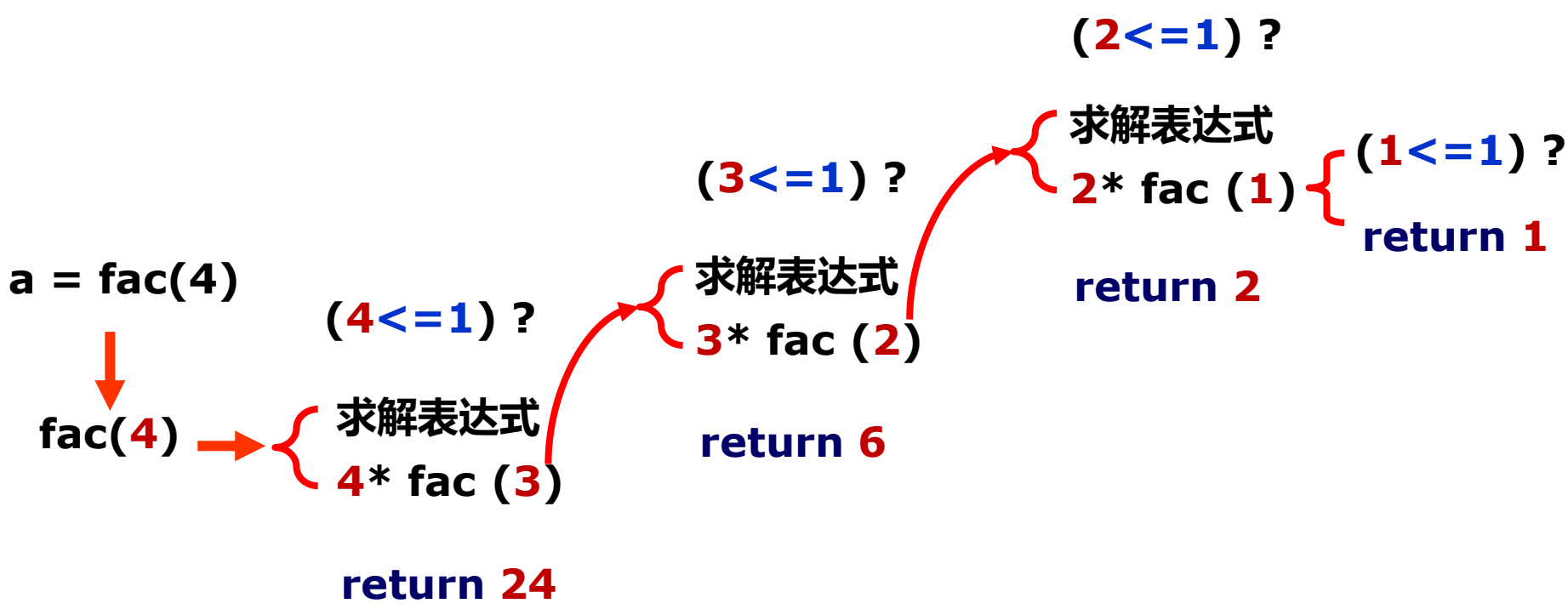
- 考察下面的函数：

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

- 为了解递归的工作原理，我们来跟踪 fact(4) 的执行

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

```
int main(void){
    int a = fact(4);
    printf("fact(3) = %d", a);
    return 0;
}
```



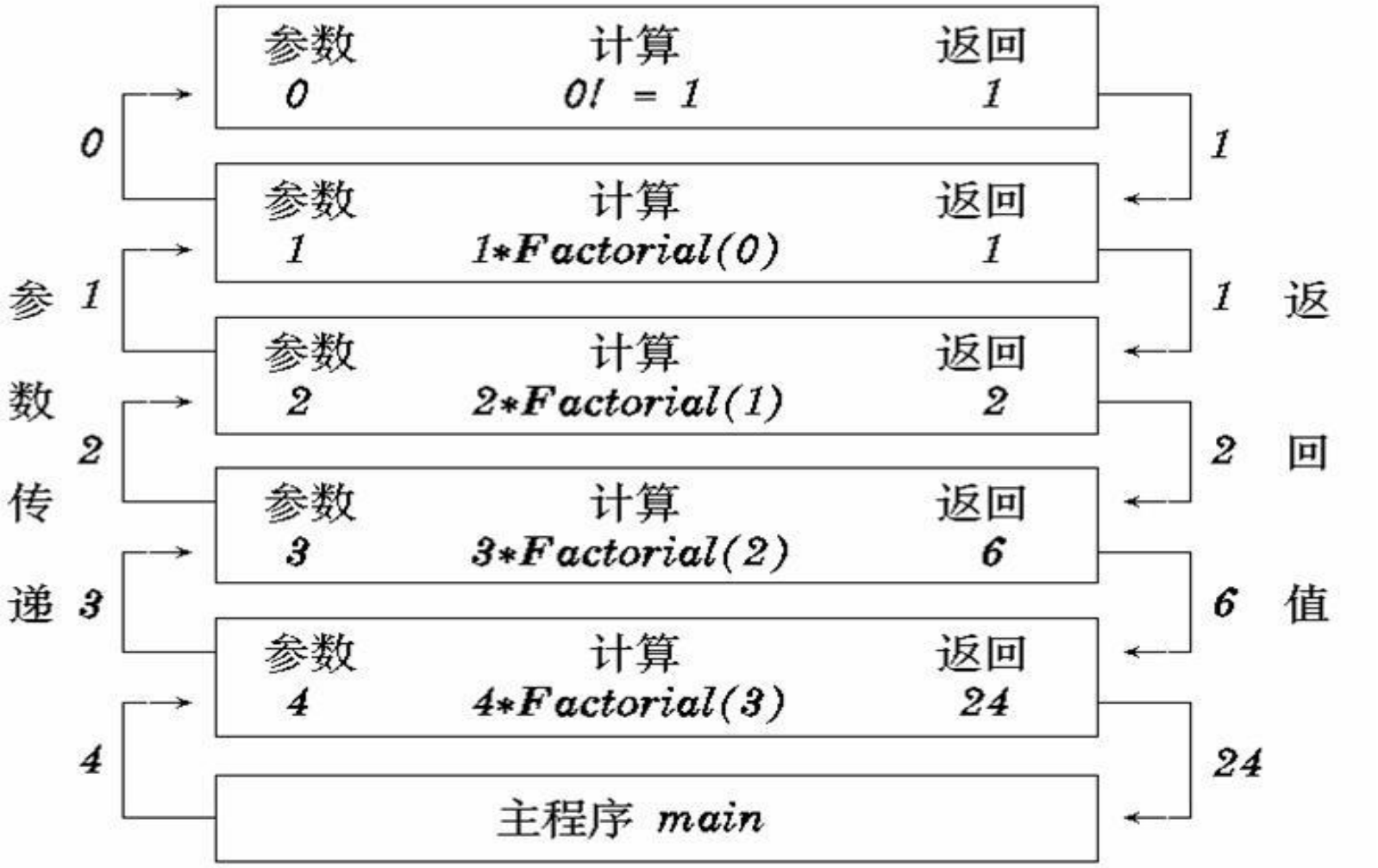
调用函数时系统的工作

- 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嵌套调用自身
- 信息传递和控制转移通过栈实现
  - 每次递归调用时需要为参数和局部变量另外分配存储空间
  - 层层向下递归，每层递归调用分配的空间形成递归工作记录
  - 用递归工作栈按照后进先出规则管理这些信息
  - 因此函数退出时的次序正好与递归调用的顺序相反

递归过程与递归工作栈


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
0
1
2
3
4
参数传递
1
1
2
6
24
返回值

递归的应用场景

- 遇到如下三种情况，可以考虑使用递归
  - 问题定义是递归的
    - 例如：阶乘、斐波纳契数列等
  - 解决问题时采用的数据结构是递归定义的
    - 例如：二叉链表
  - 问题的求解过程是递归的
    - 例如：汉诺塔问题，排列问题等

递归的应用场景

1. 问题的定义是递归的

斐波纳契数列（Fibonacci Sequence）

- 斐波纳契数列的物理模型（1202年）
  - 假设：第一个月初有一对刚出生的兔子
  - 假设：一对初生兔子要两个月才到成熟期
  - 假设：每对成熟的兔子每月会产下一对小兔子
  - 假设：在考察期间兔子不会死去
  - 问题：第12个月会有多少对兔子？

《Liber Abaci》

- 斐波纳契数列：1、1、2、3、5、8、13、21、……
  - F0=0，F1=1
  - Fn = F(n-1) + F(n-2) ，（n>=2，n∈N）

斐波纳契数列

- 可以将结果列表如下

1月	2月	3月	4月	5月	6月
1	1	2	3	5	8

	8月	9月	10月	11月	12月
13	21	34	55	89	144

- 12个月以后的小兔子数量是144对

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：

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

解法1：递归

```
long Fib1(int n){
    if (n <= 1) {
        return n;
    } else{
        return fib1(n - 1) + fib1(n - 2);
    }
}
```

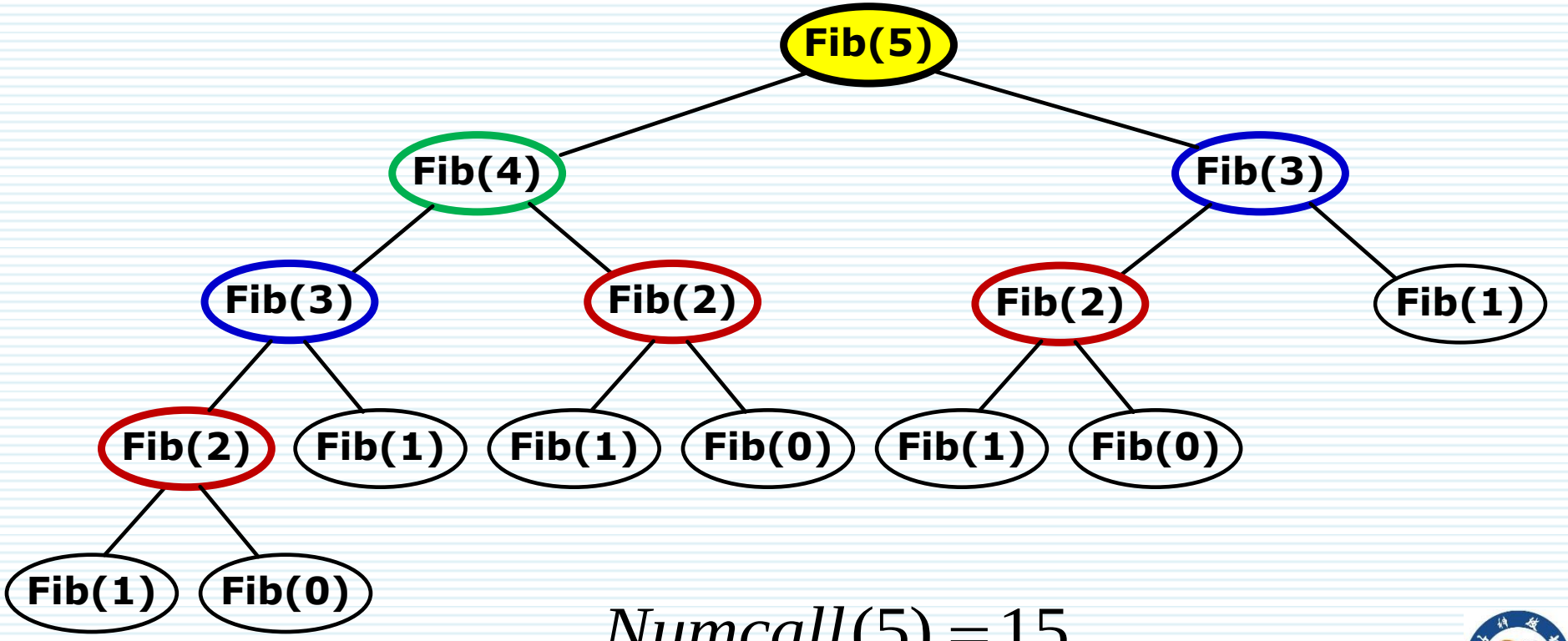
$$T(n) = O\left(\frac{1+\sqrt{5}}{2}\right)^n$$

斐波那契数列的递归求解过程

- fibonacci(5)的递归求解过程

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

$$(n \geq 2)$$



$$Numcall(5) = 15$$

斐波那契数列的递归求解过程

- Fib1函数的调用次数

G(n) = G(n-1) + G(n-2) + 1; n>=2;

G(n) = 2* **fibonacci(n+1) - 1**

Fib(0) = 0 G(0) = 1

Fib(1) = 1 G(1) = 1

Fib(2) = 1 G(2) = 1 + 1 + 1 = 3

Fib(3) = 2 G(3) = 3 + 1 + 1 = 5

Fib(4) = 3 G(4) = 5 + 3 + 1 = 9

Fib(5) = 5 G(5) = 9 + 5 + 1 = 15

Fib(6) = 8 G(6) = 9 + 15 + 1 = 25

……

证明：G(n) = 2* fibonacci(n+1) - 1

Fib1函数的调用次数：G(n) = G(n-1) + G(n-2) + 1; n>=2;

对比Fib(n)的定义可以看出：G(n)与Fib(n) 有线性关系

不妨设：G(n)=aFib(n+1)+b（因为：G(n)>Fib(n)）

由于：G(1)=1，Fib(2)=1，所以： a+b =1 ①

由于：G(2)=3，Fib(3)=2，所以：2a+b =3 ②

由①和②解得：a = 2；b = -1； 思考：问题的症结在哪里？

即： G(n)=2Fib(n+1)-1 对n=1和n=2成立

数学归纳：设G(n)=2Fib(n+1)-1 对n和n-1成立 ③

由定义式：G(n+1)=G(n)+G(n-1)+1 将③代入，整理得

G(n+1)= 2Fib(n+1)-1 + 2Fib(n)-1+1 = 2Fib(n+2)-1

Fib(n)的递归**调用次数**：G(n) = 2* fibonacci(n+1) - 1

函数调用对性能的影响：工作案例

```
for (int epoch = 0; epoch < nepoch; epoch++){ // nepoch = 1000
  res = 0;
  for (int batch = 0; batch < nbatches; batch++){ // nbatches = 100
    for (int k = 0; k < batchsize; k++){  // batchsize = 45000
      int i = rand_max(fb_h.size());
      int j = rand_max(entity_num);
      double pr = 1000*fb_r[i]/(fb_h[i]+fb_t[i]]);
      if (rand()%1000<pr){
        while (ok[make_pair(fb_h[i],fb_r[i])].count(j)>0)
          j = rand_max(entity_num);
        train_kb(fb_h[i],fb_l[i],fb_r[i],fb_h[i],j,fb_r[i]);
      }
      else{
        while (ok[make_pair(j,fb_r[i])].count(fb_l[i])>0)
          j = rand_max(entity_num);
        train_kb(fb_h[i],fb_l[i],fb_r[i],j,fb_l[i],fb_r[i]);
      }
    }
}
```

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 = 0, 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)**

斐波纳契数列解法3：矩阵

$$\begin{bmatrix} F_2 \\ F_1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad \begin{bmatrix} F_{n+2} \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}$$

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} F_{n-2} \\ F_{n-3} \end{bmatrix} = \cdots\cdots = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-2} \begin{bmatrix} F_2 \\ F_1 \end{bmatrix}$$

思考：方阵求n次幂的算法复杂度？

2阶方阵相乘：8次乘法，4次加法

若将2阶方阵相乘视为“原子操作”，则转化实数求n次幂问题

算法复杂度：**O(log(n))**

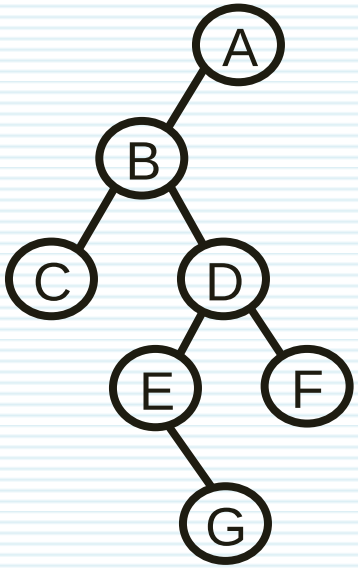
递归的应用场景

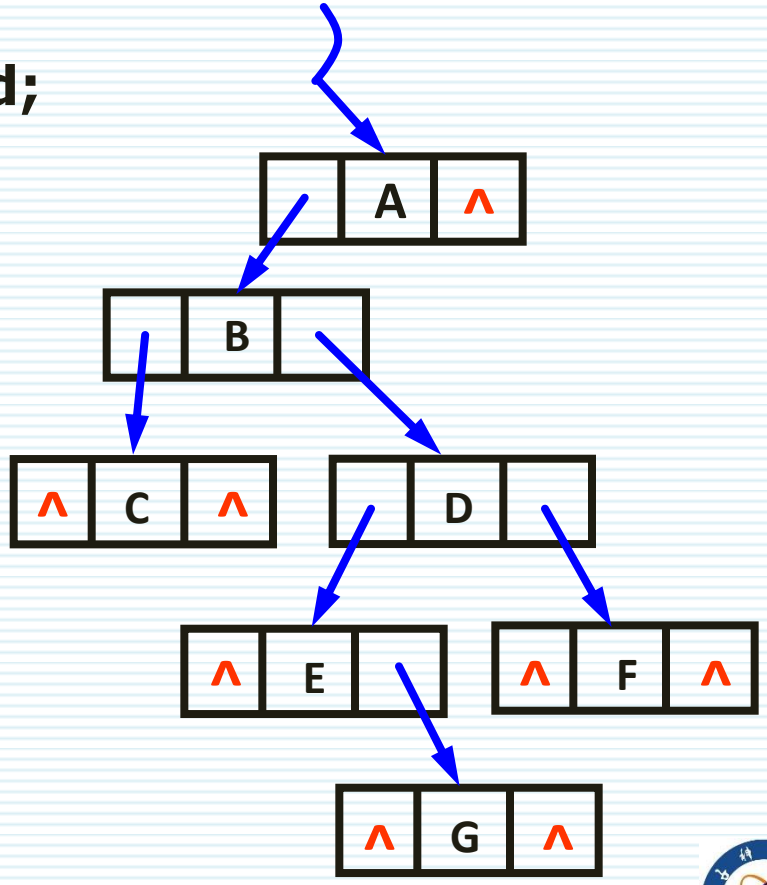
2. 问题采用的数据结构是递归的

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{
     Elemtype    data;
     struct  node  *lchild,  *rchild;
} TNode;
```



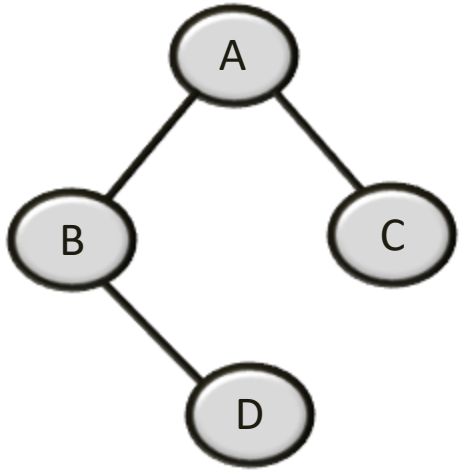


中序遍历递归算法

```
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;
}
```

统计二叉树中叶子结点个数

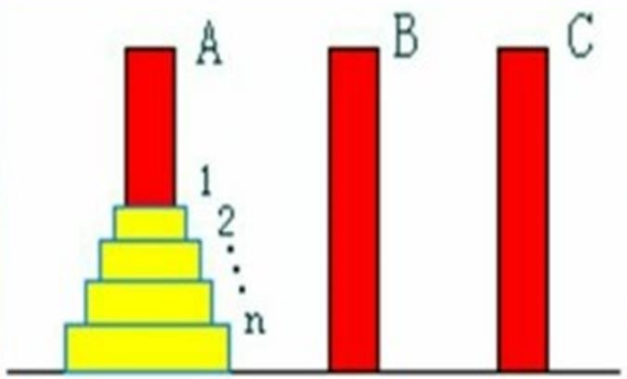
```
int count_leaves (BTPtr pbt){
    int num = 0;
    if( pbt == NULL ){ // 空树的叶节点数为零
        return 0;
    }
    if((!pbt->lchild) && (!pbt->rchild)){
        return 1;          // 左右子树均为空，则为叶节点
    }
    num = count_leaves (pbt->lchild)
        + count_leaves (pbt->rchild) ;
    return  num;           // 存在左子树或右子树时，递归调用
}
```

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

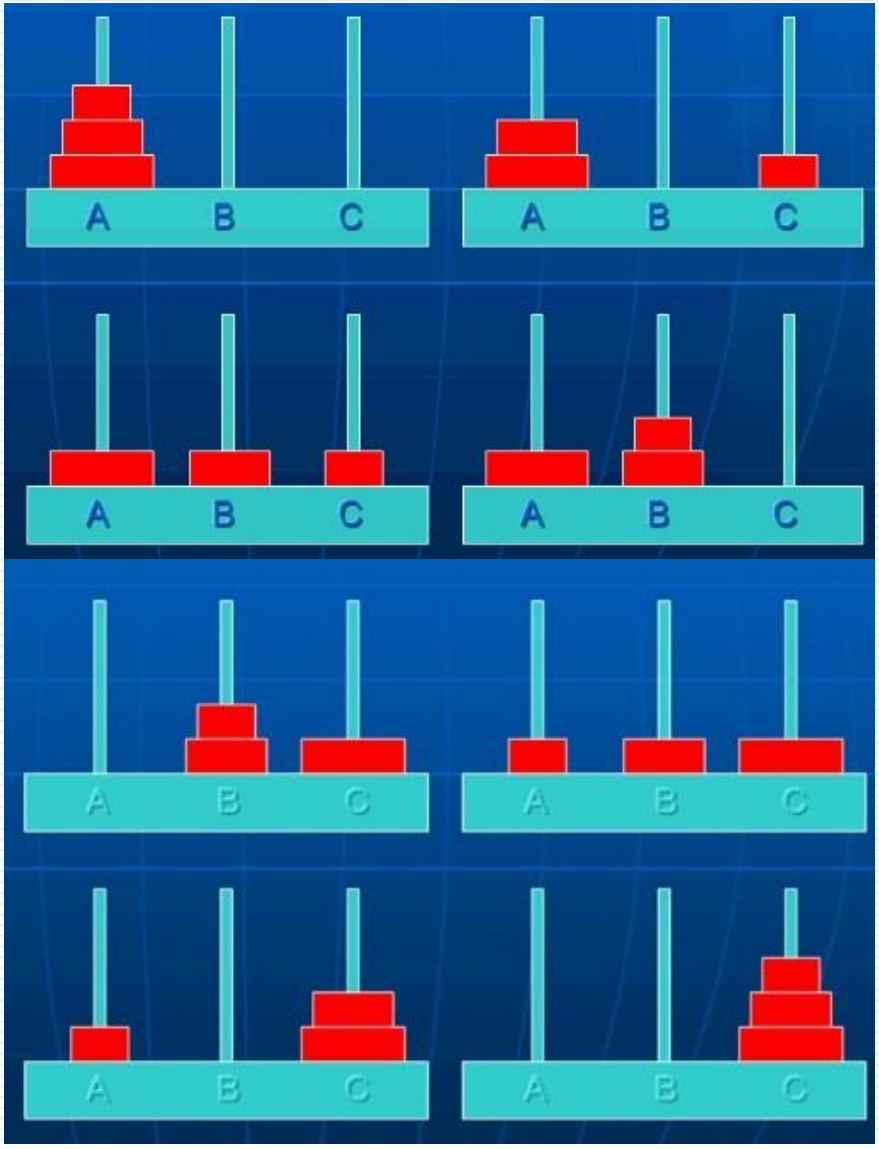
（3）1#C->B

（4）3#A->C

（5）1#B->A

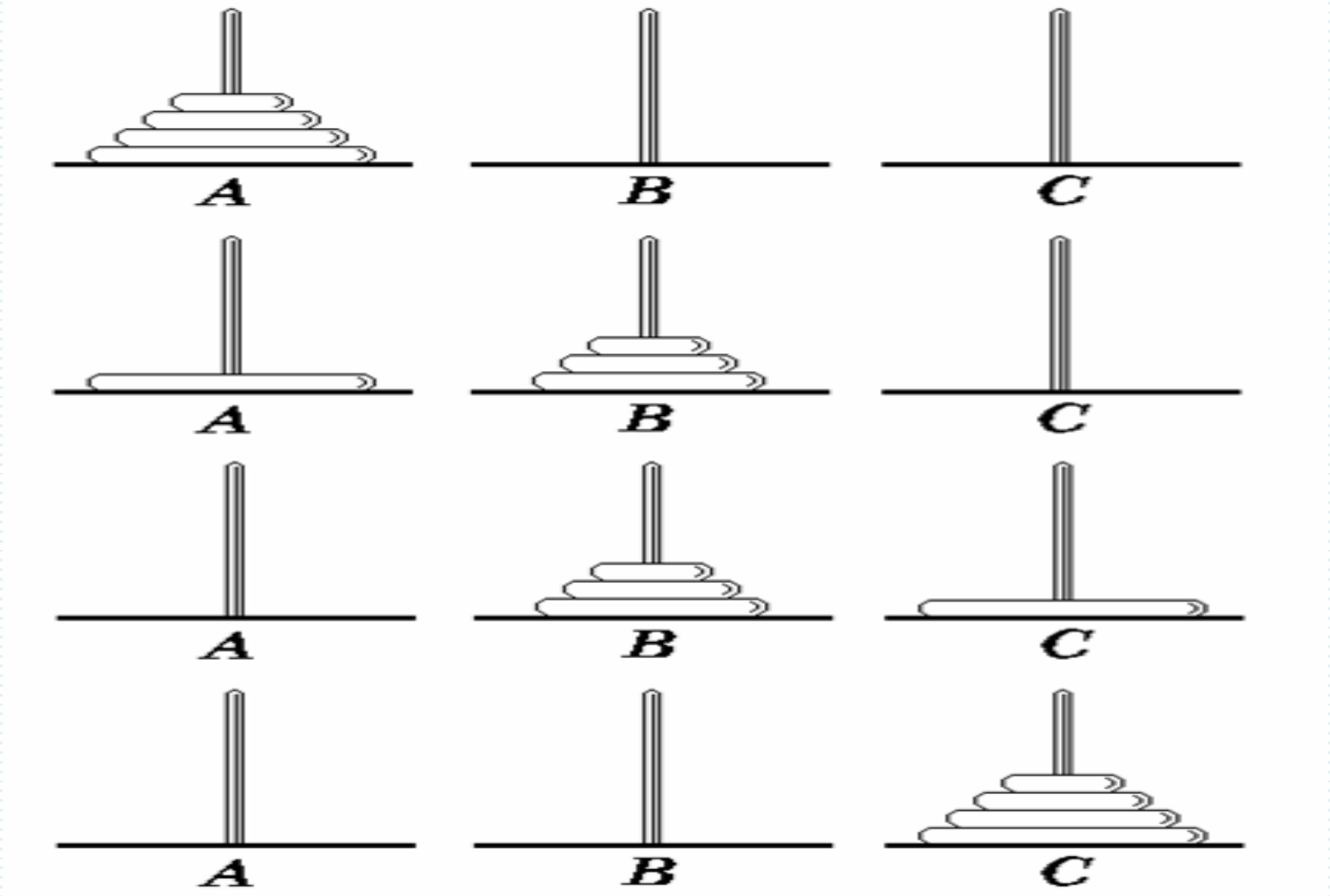
（6）2#B->C

（7）1#A->C

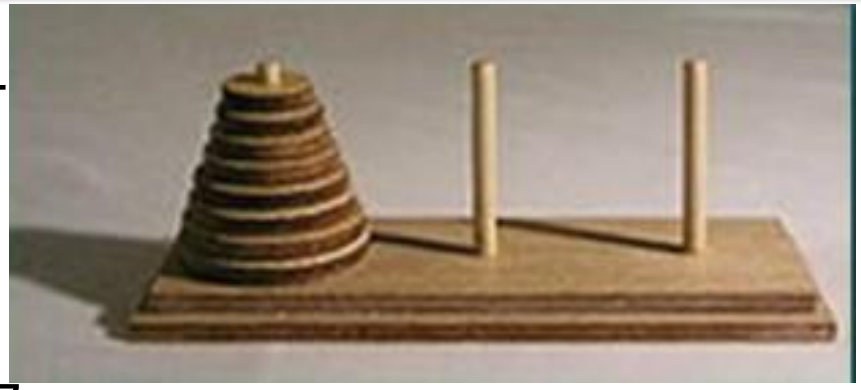


用递归技术求解汉诺塔问题

- 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=20，需移动1048575次（超过一百万次）
  - 如果n=64，需移动2^{64}-1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{$r_1, r_2, \ldots, r_n$}的全排列

N=3	N=4			
1 2 3	1 2 3 4	2 1 3 4	3 2 1 4	4 2 3 1
1 3 2	1 2 4 3	2 1 4 3	3 2 4 2	4 2 1 3
2 1 3	1 3 2 4	2 3 1 4	3 1 2 4	4 3 2 1
2 3 1	1 3 4 2	2 3 4 1	3 1 4 2	4 3 1 2
3 1 2	1 4 2 3	2 4 1 3	3 4 1 2	4 1 3 2
3 2 1	1 4 3 2	2 4 3 1	3 4 2 1	4 1 2 3

示例2：排列问题

- **设计一个递归算法：生成n个元素{$r_1,r_2,\ldots,r_n$}的全排列**
  - 设R={$r_1,r_2,\ldots,r_n$}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)

排列问题：产生list[k:m]的所有排列

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

思考：程序结束时数组R的状态是怎样的？

示例2：排列问题

- 设计一个递归算法：生成n个元素{$r_1, r_2, \ldots, r_n$}的全排列

N=3	N=4			
1 2 3	1 2 3 4	2 1 3 4	3 2 1 4	4 2 3 1
1 3 2	1 2 4 3	2 1 4 3	3 2 4 2	4 2 1 3
2 1 3	1 3 2 4	2 3 1 4	3 1 2 4	4 3 2 1
2 3 1	1 3 4 2	2 3 4 1	3 1 4 2	4 3 1 2
3 1 2	1 4 2 3	2 4 1 3	3 4 1 2	4 1 3 2
3 2 1	1 4 3 2	2 4 3 1	3 4 2 1	4 1 2 3

示例3：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
  - 问题：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。
  - 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求解，本例的递归关系在哪里？

示例3：整数划分问题

- 分析问题找规律：正整数6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
  - 规律：每一行最大的元素逐行递减
  - 思考：如何利用这种分析思路设计算法？

示例3：整数划分问题

- 问题分析：如果设p(n)为正整数n的划分数？
- 如果再增加一个自变量m？
  - 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)的递归关系如下：
  - q(n,1)=1 （n ≥ 1）； q(n,m)= q(n,n)（ m ≥ n ）
  - 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≤n-1的划分组成。
  - 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=m的划分和n_1≤m-1 的划分组成。

整数划分问题

- q(n,m)的递归式整理如下：

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

提问：正整数n的划分数？ **p(n)=q(n,n)**

类似问题：把N个球放到M个盒子, 有多少种放法?

思考：是否允许有空盒子存在？

整数划分问题

```
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);
}
```

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

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么？
  - 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

分治法的基本步骤

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

平衡(balancing)子问题思想

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。

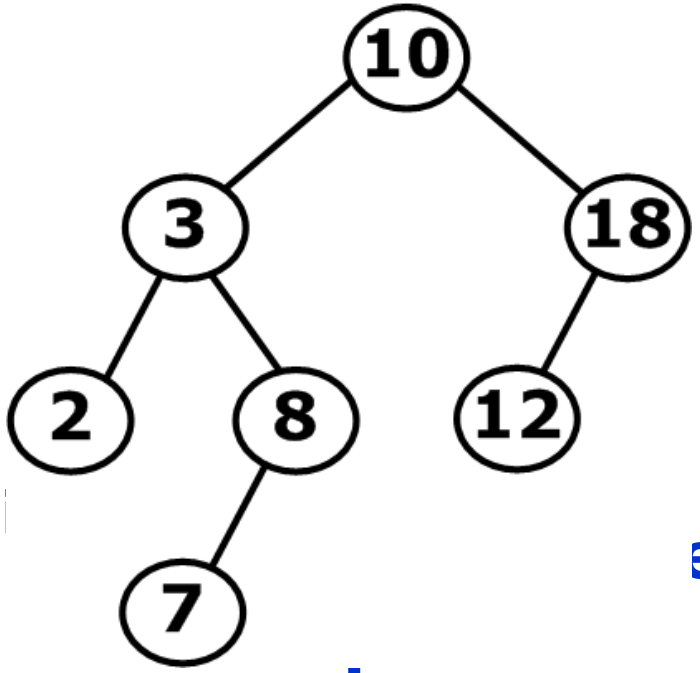
分治法（Divide-and-Conquer）

- 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**相互独立**，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(Tnode*  pbt){
    int dL = 0, dR = 0
    if( pbt == NULL ) {
        return 0;
    }
    if((!pbt->lchild) && (!pbt->rch
        return 1;
    }
    dL = get_depth(pbt->lchild);
    dR = get_depth(pbt->rchild);

    return 1 + ((dL > dR) ? dL : dR);
}
```



er

Divide

Combine

分治法的适用条件

- 分治法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
   - 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（应用分治法的前提）
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
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

示例1：二分搜索

- 问题：a[0:n-1]包含排好序的n个元素，从中搜索特定元素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
  - 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
  - 在子问题中的查询结果可以推广到整个问题
- 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

2. 大整数的乘法

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 |
  - $X = a\ 2^{n/2} + b \quad Y = c\ 2^{n/2} + d$
  - $XY = ac\ 2^{n} + (ad+bc)\ 2^{n/2} + bd$
- **复杂度分析**

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

主定理：$T(n) = aT(n/b) + f(n)$

(1) 若：$f(n) = O(n^{\log_b a - \epsilon})$ 则：$T(n) = \Theta(n^{\log_b a})$

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

(3) 若：$f(n) = \Omega(n^{\log_b a + \epsilon})$
且：$af(n/b) \leq cf(n)$ 则：$T(n) = \Theta(f(n))$

$$T(n) = 4T(n/2) + O(n)$$

满足条件(1)，因此有：$T(n) = O(n^2)$ ✘没有改进

思考：如何改进？

分治法示例2：大整数的乘法

- **改进大整数乘法的分治算法**
  - **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**

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

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

示例3：Strassen矩阵乘法

- 分治法求解矩阵乘法：C=AB
  - 与整数乘法类似，可以对矩阵A，B和C进行分块

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

- 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**

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

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

T(n)=O(n^3)

思考：怎样改进？

Strassen矩阵乘法

- 为了降低时间复杂度，必须减少乘法的次数

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

$$M_1 = A_{11}(B_{12} - B_{22})$$

$$M_2 = (A_{11} + A_{12})B_{22}$$

$$M_3 = (A_{21} + A_{22})B_{11}$$

$$M_4 = A_{22}(B_{21} - B_{11})$$

$$M_5 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M_6 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$M_7 = (A_{11} - A_{21})(B_{11} + B_{12})$$

$$C_{11} = M_5 + M_4 - M_2 + M_6$$

$$C_{12} = M_1 + M_2$$

$$C_{21} = M_3 + M_4$$

$$C_{22} = M_5 + M_1 - M_3 - M_7$$

$$T(n) = 7T(n/2) + \Theta(n^2)$$

$$\mathbf{T(n)=O(n^{\log 7})=O(n^{2.8})}$$

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？
  - 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 $O(n^{2.376})$
  - 开放问题：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.

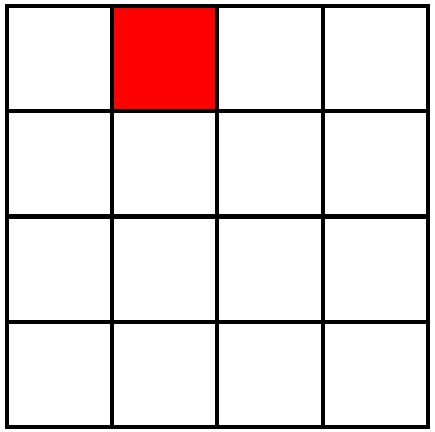
$O(n^3)$ vs. $O(n^{2.81})$

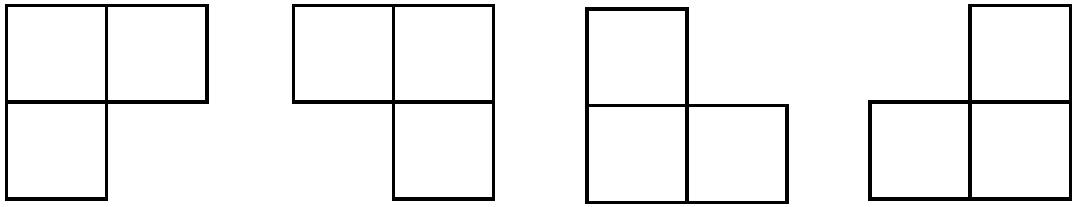
- **设 n = 100,000**
  - n^3 = 1000000000000000
  - $n^{2.81}$ = 112201845430196
  - $n^{2.376}$ = 758577575029
- 设CPU主频2.4GHz，则完成一次矩阵相乘所需时间估计为
  - 常规算法：388,051秒 (**108 小时**)
  - Strassen算法：43,540秒 (**12 小时**)
  - 当前最优算法：294秒 (**4.9 分钟**)

4. 棋盘覆盖问题

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

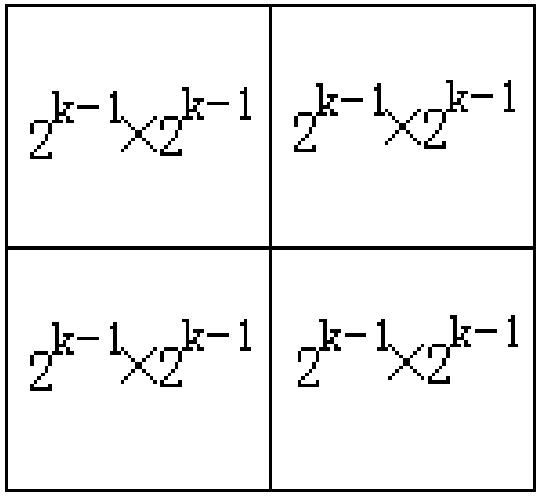


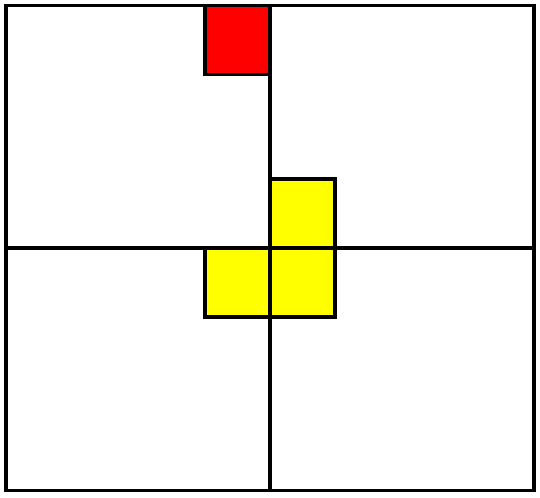


思考：分而治之？

棋盘覆盖问题

- 当k>0时，将$2^k \times 2^k$棋盘分割为4个$2^{k-1} \times 2^{k-1}$ 子棋盘
  - 特殊方格必位于4个较小的子棋盘其中之一
  - 其余3个子棋盘中无特殊方格……问题变化了？
- 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
- 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





棋盘覆盖问题

```
// 棋盘覆盖问题
int Board[MAX][MAX];      // 棋盘数组
int tile = 1;                          // L 型骨牌编号
// x0：棋盘左上角方格的行号（top-left corner）
// y0：棋盘左上角方格的列号
// x：特殊方格的行号
// y：特殊方格的列号
void ChessBoard(int x0 , int y0 , int x , int y ,int size) {
        if(size == 1) return;
        int t = tile++;          // L 型骨牌编号自增
        int s = size / 2;       // 分割棋盘
```

棋盘覆盖问题

```
// 覆盖左上角子棋盘
if(x < x0 + s && y < y0 + s) {
    // 如果特殊方格在此子棋盘中，则对其进行覆盖
    ChessBoard(x0 , y0 , x , y , s);
} else {
    // 如果特殊方格不在此子棋盘中
    // 则用编号为 t 的 L 型骨牌覆盖该子棋盘的右下角
    Board[x0 + s - 1][y0 + s - 1] = t;
    // 对经过处理的左上角子棋盘进行覆盖
    ChessBoard(x0 , y0 , x0 + s - 1 , y0 + s - 1 , s);
}
```

棋盘覆盖问题

```
// 覆盖右上角子棋盘
if(x < x0 + s && y >= y0 + s) {
    // 如果特殊方格在此子棋盘中，则对其进行覆盖
    ChessBoard(x0 , y0+s , x , y , s);
} else {
    // 如果特殊方格不在此子棋盘中
    // 则用编号为 t 的 L 型骨牌覆盖该子棋盘的左下角
    Board[x0 + s - 1][y0 + s] = t;
    // 对经过处理的右上角子棋盘进行覆盖
    ChessBoard(x0 , y0+s , x0+s-1 , y0+s , s);
}
```

棋盘覆盖问题

```
// 覆盖左下角子棋盘
if(x >= x0 + s && y < y0 + s) {
    // 如果特殊方格在此子棋盘中，则对其进行覆盖
    ChessBoard(x0 + s , y0 , x , y , s);
} else {
    // 如果特殊方格不在此子棋盘中
    // 用编号为 t 的 L 型骨牌覆盖该子棋盘的右上角
    Board[x0 + s][y0 + s - 1] = t;
    // 对经过处理的右上角子棋盘进行覆盖
    ChessBoard(x0 + s, y0 , x0 + s, y0 + s - 1, s);
}
```

棋盘覆盖问题

```
// 覆盖右下角子棋盘
if(x >= x0 + s && y >= y0+s) {
    // 如果特殊方格在此子棋盘中，则对其进行覆盖
    ChessBoard(x0 + s , y0 + s , x , y , s);
} else {
```

复杂度分析

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

$T(n)=O(4^k)$ 渐进意义下的最优算法

```
    ChessBoard(x0 + s , y0 + s , x0 + s , y0 + s , s);
}
}
```

分治法示例

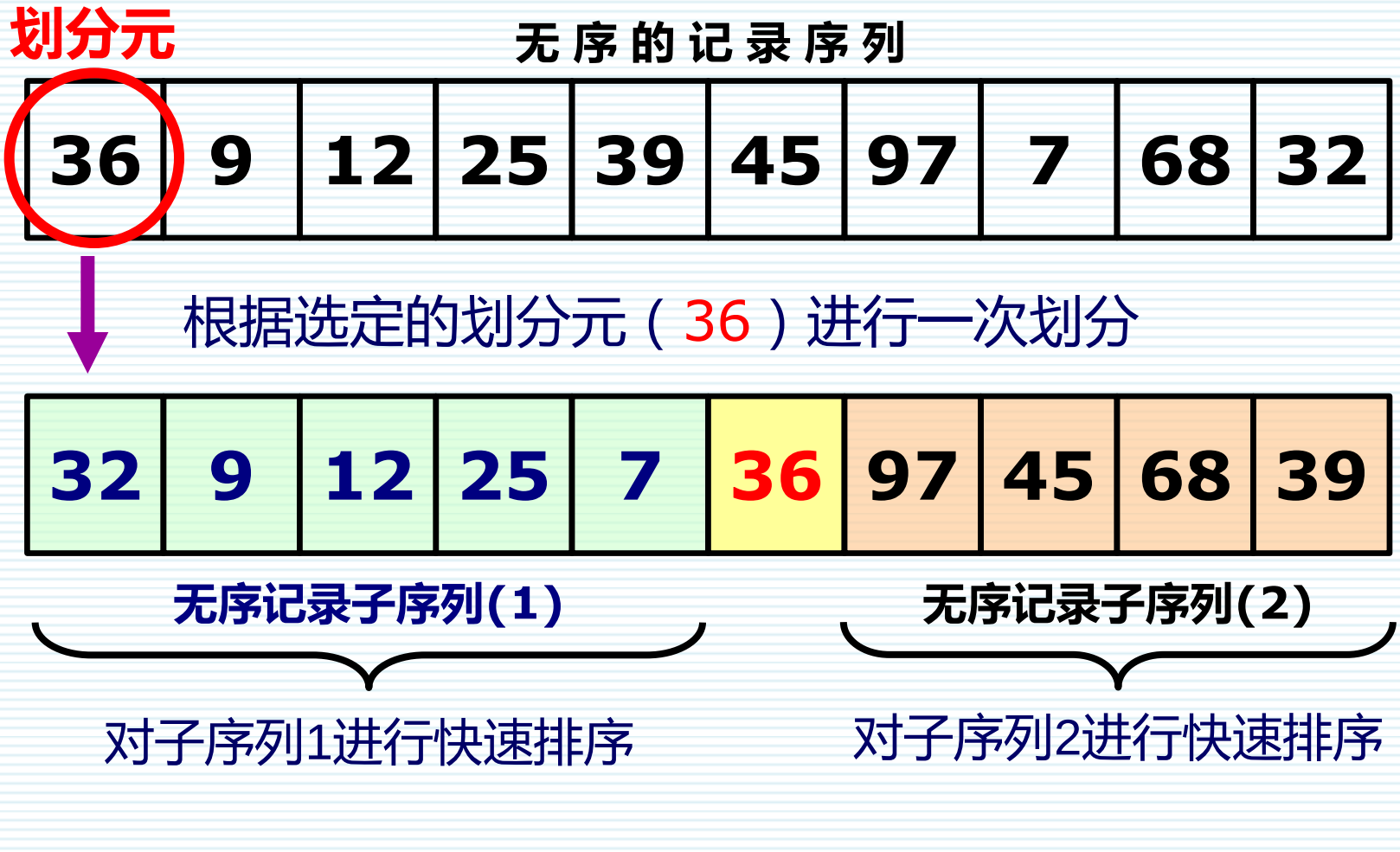
5. 快速排序算法

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**

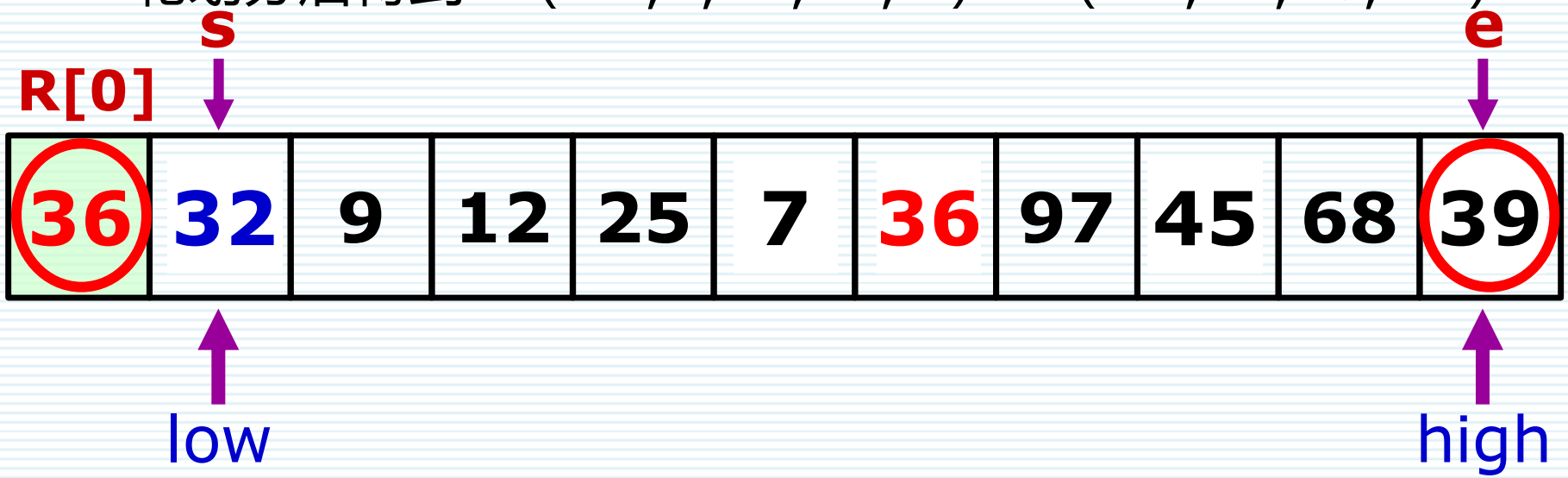
快速排序（Quick Sort）

- 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）



快速排序算法特点

- 时间复杂度
  - 最好情况
    - T(n)=O(n logn)（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）
    - 解决方案：三者取中，或者随机选取划分元
  - 快速排序算法的平均时间复杂度为：O(nlogn)
- 快速排序算法是不稳定的
  - 例如待排序序列： **49** 49 38 65
  - 快速排序结果为： 38 49 **49** 65

6. 线性时间元素选择问题

线性时间元素选择问题

- 问题描述
  - 给定线性序集中的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
  - 解法：可以通过排序求解元素选择问题： O(nlogn)
  - 问题：元素选择问题是否可以在O(n)时间内得到解决？
  - 思路：根据分治的思想，模仿快排算法选出第k小元素
    - 划分后只需对其中一个子数组进行递归处理
    - 子数组的选择与划分元和k相关

线性时间元素选择问题

```
int RandSelect(int A[], int start, int end, int k) {
    if (start ==end) return A[start];
    int i=Partition(A, start, end);      //划分元位置
    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);
}
```

在最坏情况下，算法RandSelect需要$O(n^2)$ 时间

但可以证明，算法运行的平均时间为$O(n)$

线性时间元素选择问题

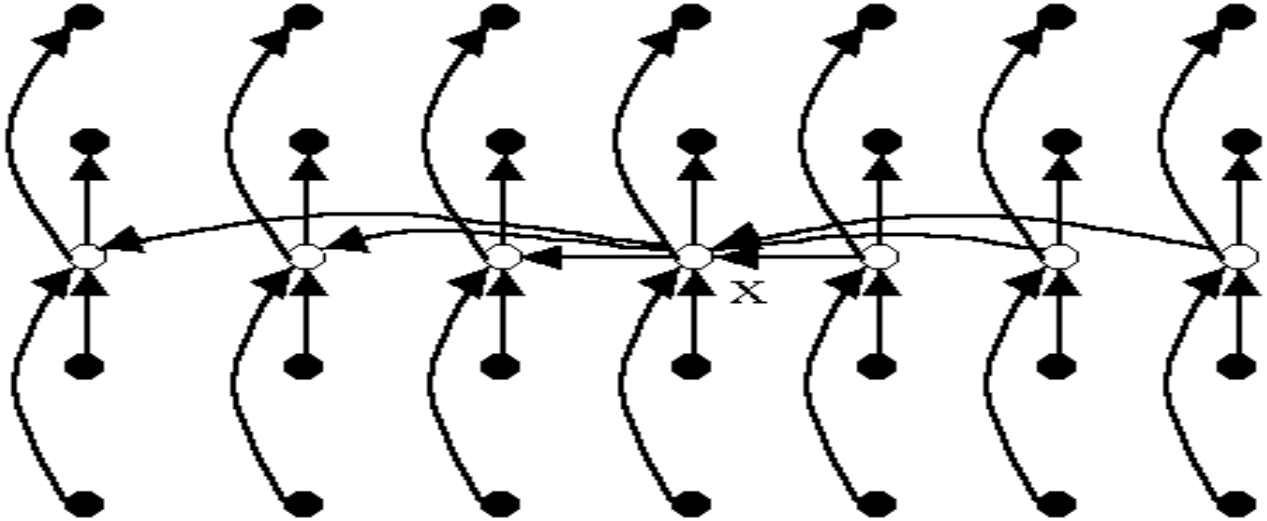
- 思考：如何实现在最坏情况下也能用O(n)时间完成选择？
- 提示：T(n)=T(n/2)+Θ(n) ➔ **T(n)=O(n)**
- 提示：T(n)=T(0.9n)+Θ(n) ➔ ？
  - 如果能在线性时间内找到一个划分基准
  - 使得两个子数组的长度都至少为原数组长度的ε倍(0<ε<1)
    - ε=0.9表示每次划分得到的子数组的长度至少缩短1/10
  - 那么就可以在最坏情况下用O(n)时间完成选择任务

(1) 若：$f(n)=O(n^{\log_b a-\epsilon})$ 则：$T(n)=\Theta(n^{\log_b a})$

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

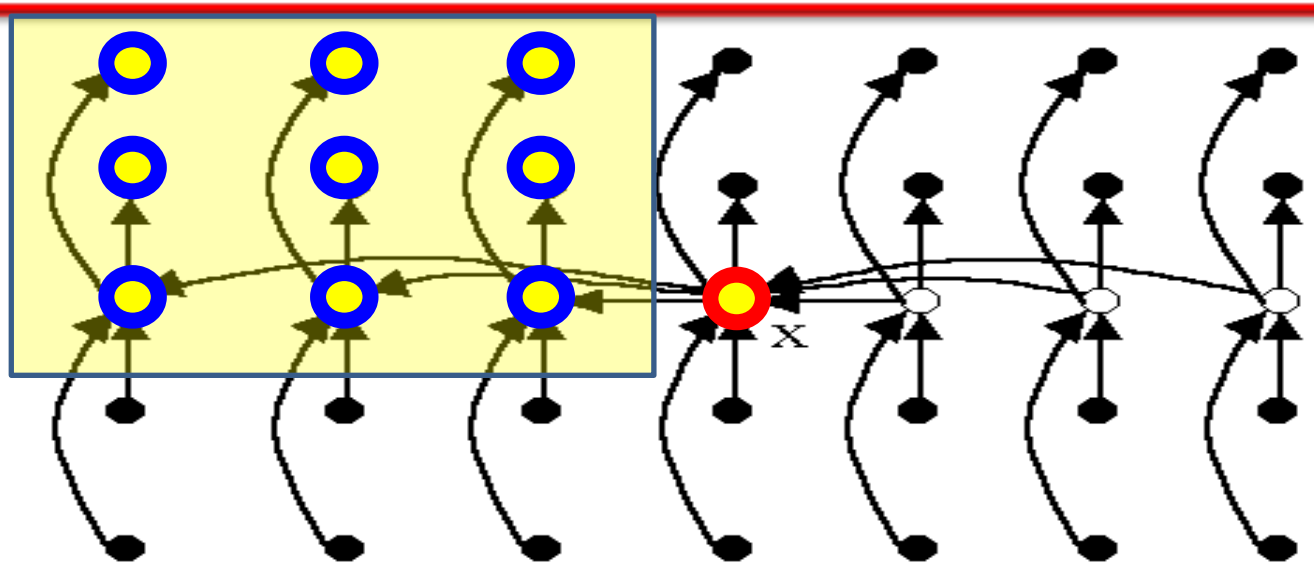
(3) 若：$f(n)=\Omega(n^{\log_b a+\epsilon})$ 则：$T(n)=\Theta(f(n))$

线性时间元素选择问题



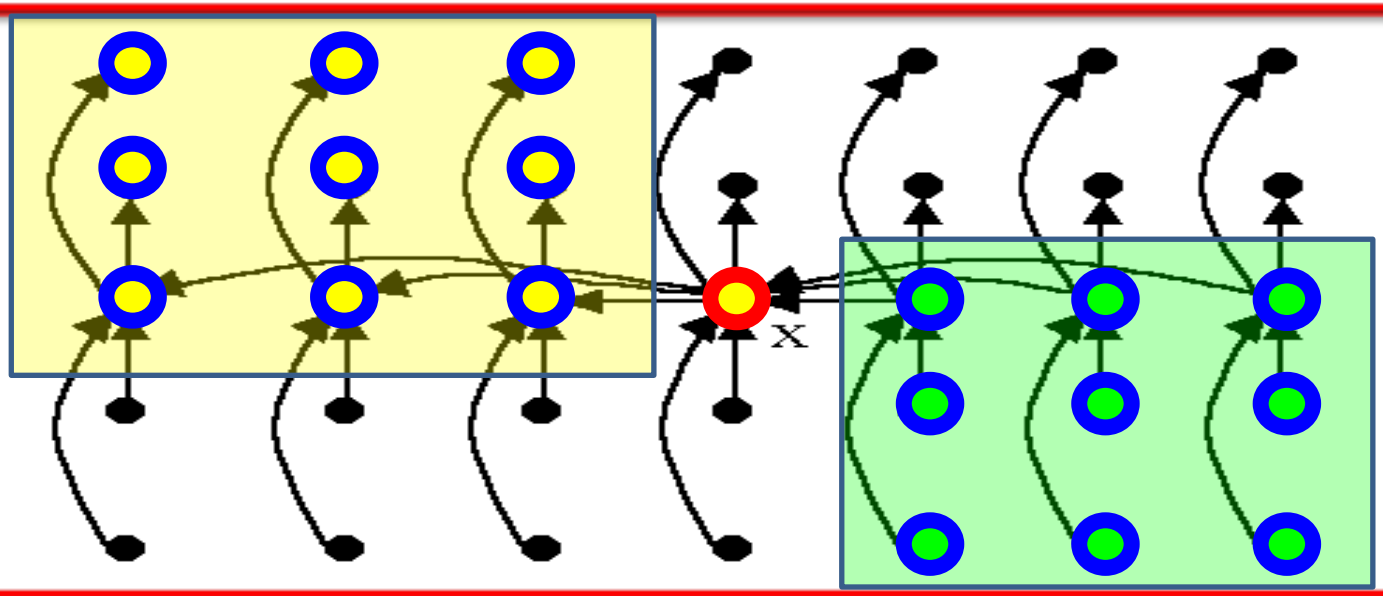
- 线性时间内找到合理划分基准的方法
  - 将n个输入元素划分成⌊ n/5 ⌋个组，每组5个元素
    - 则只可能有一个组不是5个元素
  - 用任意一种排序算法对每组中的元素排序　　**O(1)**
  - 然后取出每组的中位数，共⌊ n/5 ⌋个元素

线性时间元素选择问题



- 线性时间内找到合理划分基准的方法（续）
  - 找出这⌊ n/5 ⌋个元素的中位数 **x**
    - 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**
  - 不妨设所有元素互不相同
    - 则：**基准x**至少比3(n-5)/10个元素大

线性时间元素选择问题



- 线性时间内找到合理划分基准的方法（续）
  - 不妨设所有元素互不相同
    - 则：**基准x**至少比3(n-5)/10个元素**大**
    - 且：**基准x**至少比3(n-5)/10个元素**小**
  - 当n≥30时，3(n-5)/10≥n/4
    - 按此基准划分所得的2个子数组的长度都至少缩短1/4

线性时间元素选择问题

```
int Select(int A[], int s, int e, int k) {
    if (e-s<30) { 对数组a[s:e]排序; return A[s+k-1]; }
    for ( int i = 0; i<=(e-s-4)/5; i++ ) {
        将a[s+5*i]至a[s+5*i+4]的第3小元素与a[s+i]交换位置;
    }
    // 找中位数的中位数，e-s-4即n-5
    int x = Select(a, s, s+(e-s-4)/5, (e-s-4)/10);
    int b = Partition(a, s, e, x);
    int n = b-s+1;
    if (k<=n) return Select(a, s, b, k);
    else return Select(a, b+1, e, k-n);
}
```

算法复杂度分析

```
int Select(int A[],  int s,  int e,  int k) {          O(1)
    if (e-s<30) { 对数组a[s:e]排序;  return A[s+k-1]; }
    for ( int i = 0; i<=(e-s-4)/5; i++ ) {          O(n)
        将a[s+5*i]至a[s+5*i+4]的第3小元素与a[s+i]交换位置;
    }
    // 找中位数的中位数，e-s-4即n-5                    T(n/5)
    int x = Select(a, s, s+(e-s-4)/5, (e-s-4)/10);
    int b = Partition(a, s, e, x);                  O(n)
    int n = b-s+1;
    if (k<=n) return Select(a, s, b, k);            ≤T(3n/4)
    else return Select(a, b+1, e, k-n);             ≤T(3n/4)
}
```

线性时间元素选择问题

- 算法复杂度分析： ➔ **T(n)=O(n)**

$$T(n) \le \begin{cases} O(1) & n < 30 \\ O(n) + T(n/5 + 3n/4) & n \ge 30 \end{cases}$$

- 算法小结
  - 分组大小固定为5，以30作为是否递归调用的分界点
  - 这两点保证了T(n)的递归式中的系数ε<1
    - 即： n/5+3n/4=19n/20 ➔ ε=19/20<1
  - 这是使T(n)=O(n)的关键!!!
    - 当然，除了5和30之外，还有其他选择……

分治法示例

7. 最接近点对问题

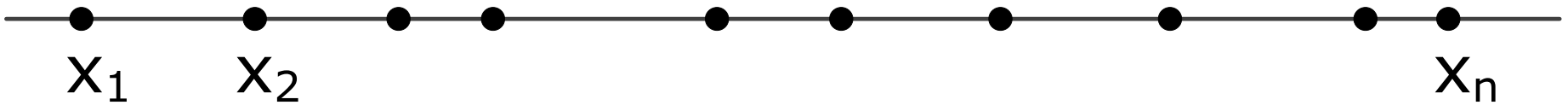
最接近点对问题

- 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

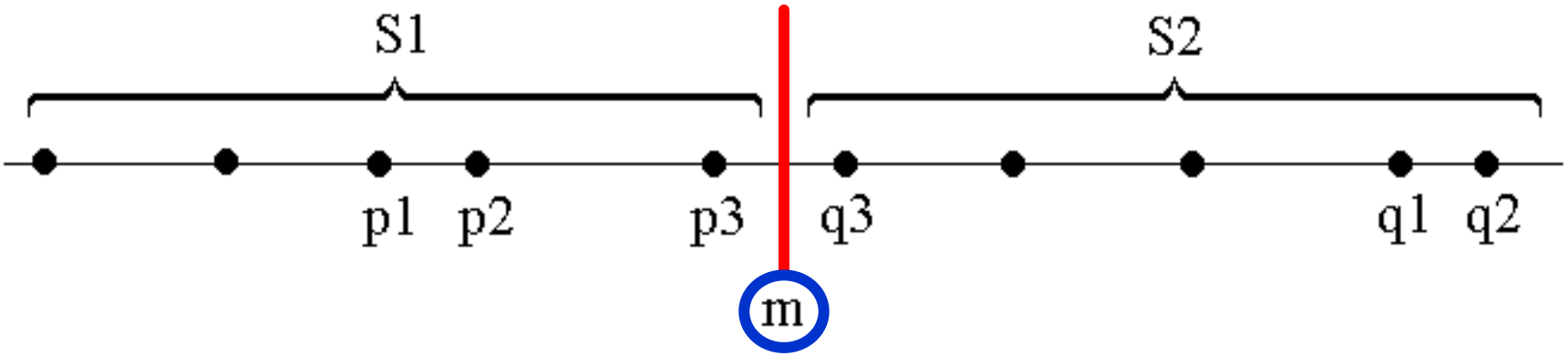
- **直观解法**
  - 计算每个点与其他n-1个点的距离，找出最小距离
  - 时间复杂度： $\mathbf{T(n)=n(n-1)/2+n=O(n^2)}$
- **思考：可否利用分治的思想简化计算？**
  - 分解：将n个点的集合分成大小近似相等的两个子集
  - 求解：递归地求解两个子集内部的最接近点对
  - 合并（关键问题）：从子空间内部最接近点对，和两个子空间之间的最接近点对中，选择最接近点对

分治法求解最接近点对问题



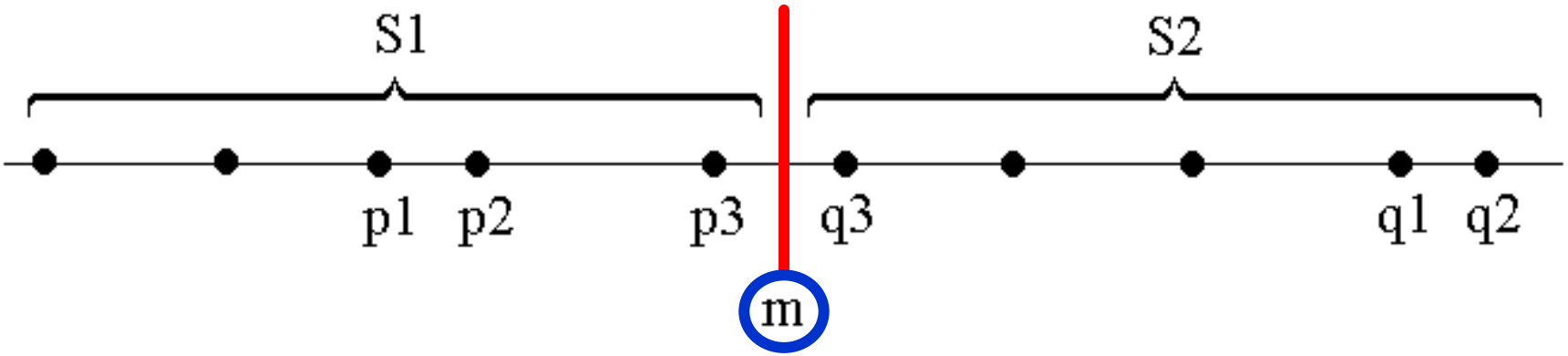
- 首先考虑一维的情况
  - 此时S中的n个点退化为x轴上的n个实数 x1, x2, …, xn
  - 最接近点对：即这n个实数中相差最小的两个实数
  - 通过排序可以在O(nlogn)时间解决问题（how？）
  - 缺点：方法难以推广到高维空间（我们仅考虑二维）
  - 思考：可否采用分而治之的思想给出一个通用解决方案？

分治法求解最接近点对问题



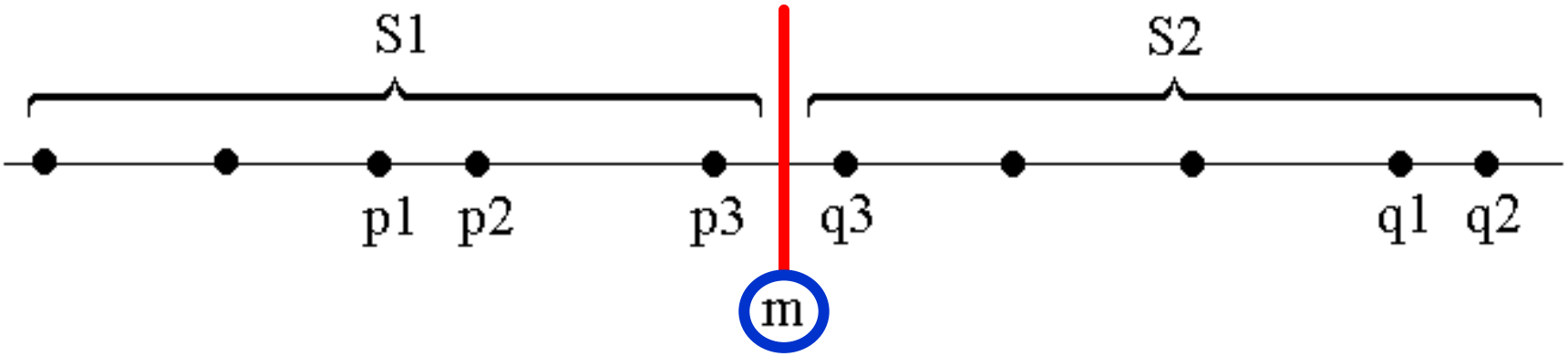
- 分治法
  - 假设我们用x轴上某个点m将S划分为两个子集S1和S2
    - 平衡子问题思想：用S中各点坐标的中位数作为分割点
  - 递归地在S1和S2上找出其最接近点对{p1,p2}和{q1,q2}
  - 则S中的最接近点对：
    - 或者是：{p1,p2} 与 {q1,q2} 中距离较小者
    - 或者是：{p3,q3}，其中：p3∈S1 且 q3∈S2

分治法求解最接近点对问题



- 分治法
  - 设d=min{|p1-p2|,|q1-q2|}
  - 断言：在S1中每个长度为d的半闭区间至多包含一个点
    - 若：S的最接近点对是{p3,q3}，即|p3-q3|<d
    - 则：p3和q3与m的距离均不超过d
    - 即：p3∈(m-d, m]，q3∈(m, m+d]

分治法求解最接近点对问题



- 在S1中每个长度为d的半闭区间至多包含一个点
  - 若：(m-d,m]中包含S中的一个点
    - 则：该点为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：线性时间内可找到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？
    - 分割点m的选取应使得划分出的子集尽可能平衡

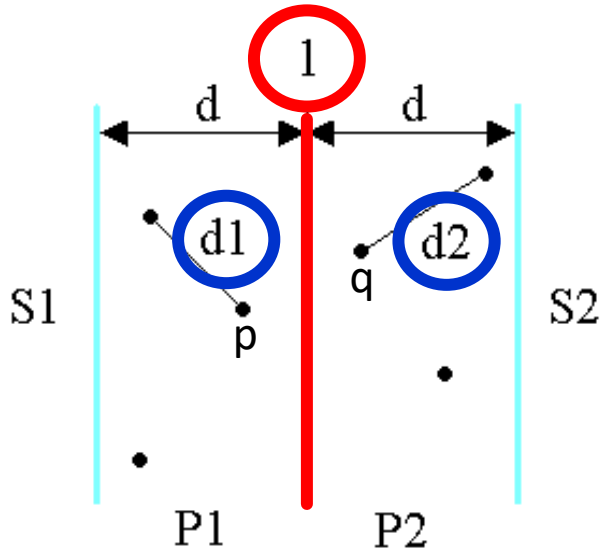
求一维点集S的最接近点对的算法

```
int cpair(int S[], int n){
    int  m, d, d1, d2;
    if(n<2) {return  INTMAX;}
    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}
    d1 = cpair(S1, n1);  d2 = cpair(S2, n2);
    p = max(S1);   q = min(S2);
    d = min(d1, d2, q-p);
    return d;
}
```

T(n)=O(nlogn)

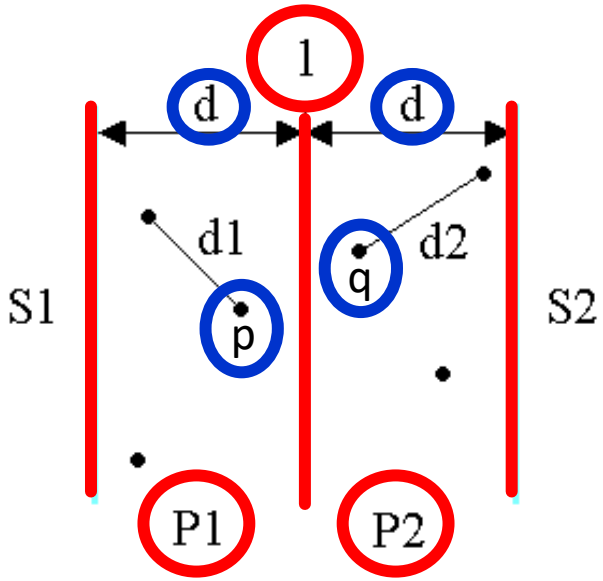
复杂度分析： $T(n)=\begin{cases} O(1) & n<2 \\ 2T(n/2)+\Theta(n) & n\geq 2 \end{cases}$

分治法求解最接近点对问题



- 考虑二维空间的情况
  - 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，m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

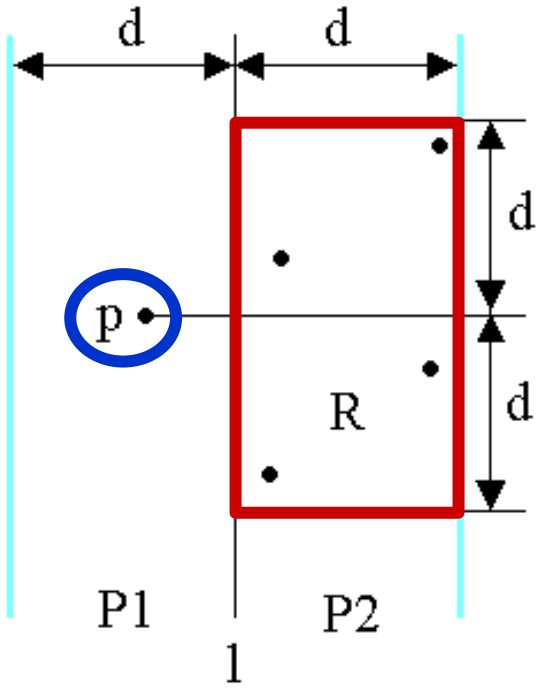
分治法求解最接近点对问题



- 考虑二维空间的情况 **是否仍然能在线性时间实现结果合并？**
  - 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（p∈S1，q∈S2）
  - 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

- 考虑P1中任意一点p
  - 若：它与P2中的点q构成最接近点对
  - 则：distance(p，q)＜d
  - P2中满足条件的点必定落在矩形R中
    - 矩形R的大小为：d×2d



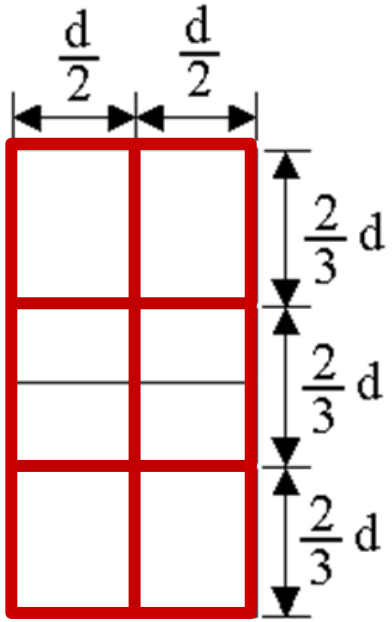
- 由d的定义可知：P2中任何两个点（$q_i \in S$）的距离都不小于d
  - 由此可知：矩形R中最多只包含6个S中的点（证明见下页）
  - 因此在合并过程中最多只需要检查6×n/2=3n个候选者

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个候选点**
  - 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
    - 至少其中一个小矩形包含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

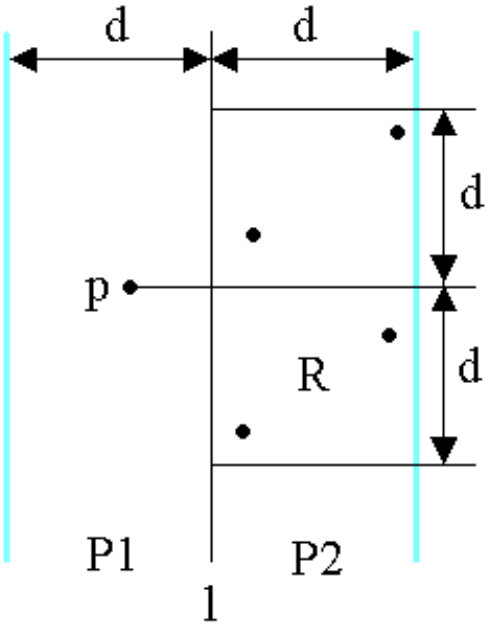
$$(x(u)-x(v))^2+(y(u)-y(v))^2 \le (d/2)^2+(2d/3)^2=\frac{25}{36}d^2$$

    - 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：对于给定点p如何确定需要检查的6个点？
  - 将点p和P2集合的所有点投影到垂线L上
  - 以符号$p_\perp$表示点p在直线 L 上的投影点
  - 设：$q \in P2$ 表示点p的候选点
    - 则：$|p_\perp - q_\perp| < d$
    - 显然满足该条件的投影点最多只有6个



- 思考：对于所有$p_i \in P1$，怎样快速找到所有的候选点？
  - 提示：若将P1和P2中的所有点按其y坐标排好序？
  - 则对与任意$p_i \in P1$：最多只需检查P2中排好序的相继6个点
  - 因此通过一次扫描就可以找出所有最接近点对的候选者

分治法求解最接近点对问题

- Locate函数：通过一次扫描找出所有最接近点对的候选者
  - 设：**P1**是S1中距分割线L的距离在d之内的所有点组成的集合
  - 设：**P2**是S2中距分割线L的距离在d之内的所有点组成的集合
  - 将：P1和P2中的点依其y坐标值排序
  - 设：**X1**和**X2**是相应的已排好序的点的序列
  - 对**X1**中的每个点，检查**X2**中与其距离在d之内的所有点
    - 当X1的扫描指针逐步移动时（即：$p_i \in$ P1发生变化）
    - X2的扫描指针可在宽为2d的区间内移动
  - 设：ds是按这种扫描方式找到的点对间的最小距离
- 结果合并：dmin=min(d, ds);

求一维点集S的最接近点对的算法

```
int cpair(int S[], int n){
    int  m, d, d1, d2;
    if(n<2) {return  INTMAX;}
    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}
    d1 = cpair(S1, n1);  d2 = cpair(S2, n2);
    d = min(d1, d2); ds = locate(S1, S1, d);
    dmin = min(d, ds);
    return d;
}
```

T(n)=O(nlogn)

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

分治法示例

8. 循环赛日程表问题

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
  - 将所有的选手均分为两部分
    - 只需对n/2个选手分别设计比赛日程表
  - 递归地用对选手进行分割
    - 直到只剩下2个选手时（最小的比赛日程单位）

循环赛日程表问题

```
void raceBoard(int k, int a[][N]){
    int size = 1<<k; int m = 1;   // 2m为当前分块的“边长”
    for(int i=1; i <= size; i++) a[1][i]=i; // 第一行为选手序号
    for(int s = 1; s <= k; s++){  // 划分的层次
        size /= 2;   // 每个层次上拥有分块的个数
        for(int t = 1; t <= size; t++){  // 对分块进行遍历
            for(int i = m+1; i<=2*m; i++){        // i为行序号
                for(int j = m+1; j<=2*m; j++){    // j为列序号
                    a[i][j+(t-1)*m*2] = a[i-m][j+(t-1)*m*2-m];
                    a[i][j+(t-1)*m*2-m] = a[i-m][j+(t-1)*m*2];
        } } }
        m *= 2;
}}
```

作业一

- 编程作业题目
  1. 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 ；当搜索元素x在数组中时，返回元素x在数组中的位置。 （完整源码，含测试代码）
  2. 实现棋盘覆盖算法（完整源码，含测试代码）
  3. 实现整数划分算法（完整源码，含测试代码）
- 作业提交方式：请发至邮箱 **cuestc@163.com**
  - 源代码以.c后缀保存为文本文件（第一题示例：1.c）
  - 将三道题的.c文件打包成一个rar文件（不要工程文件）
  - Rar文件命名方式：**学号-姓名-HW1.rar**
  - 截止日期：9**月30日 24:00**（过时不再收取）


UESTC
1956