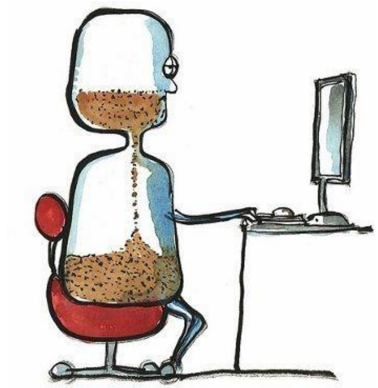
数据结构与算法分析



人工智能与自动化学院 陶文兵

wenbingtao@hust.edu.cn

参考教材

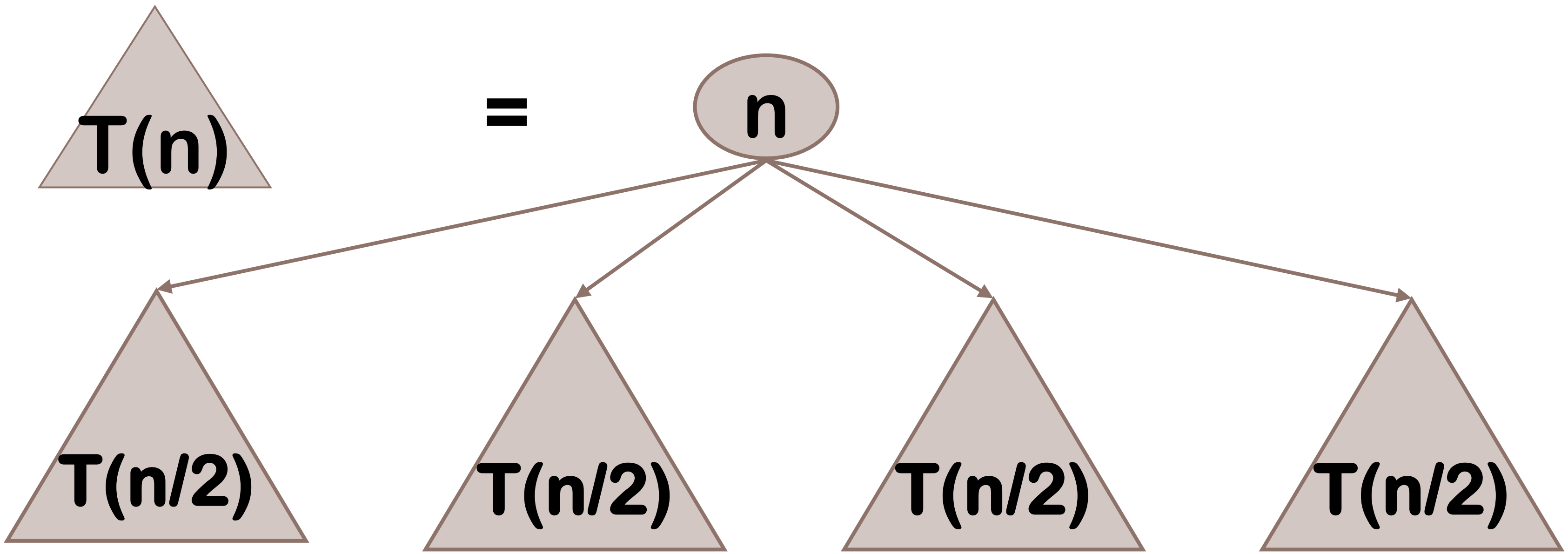
普通高等教育"十一五"国家级规划教材
国家精品课程教材
高等学校规划教材
计算机算法设计与分析
（第5版）
王晓东 编著
中国工信出版集团
电子工业出版社
http://www.phei.com.cn
（第五版）
高等学校规划教材
<code/>
计算机算法设计与分析
王晓东 / 编著

10. 递归与分治算法

- 分治法基本思想
- 递归概念回顾
- 根据实例掌握分治法
  - 二分搜索
  - 大整数的乘法
  - 棋盘覆盖
  - 最近点对问题
  - 循环赛日程表

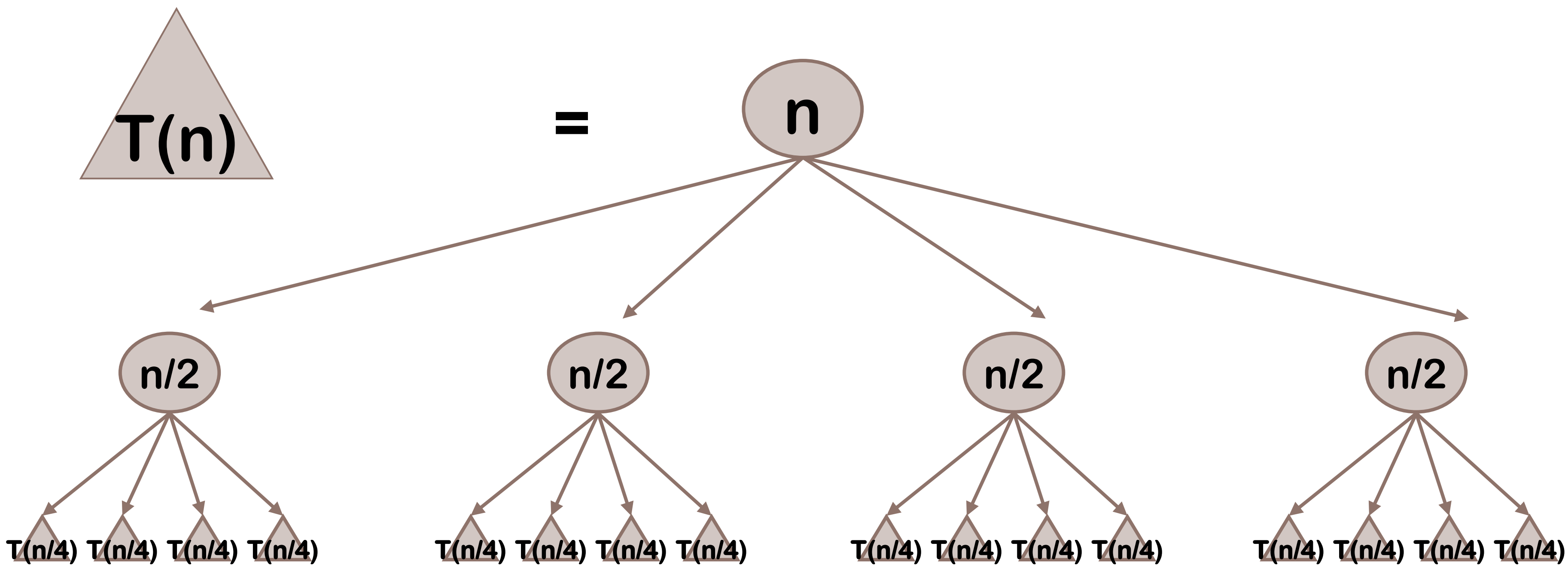
递归与分治算法基本思想

- 将待求解的问题分解成规模更小的子问题
  - 若子问题的规模仍然不够小，则再划分为k个子问题
  - 以上过程递归进行
  - 直到问题规模足够小，容易求出其解



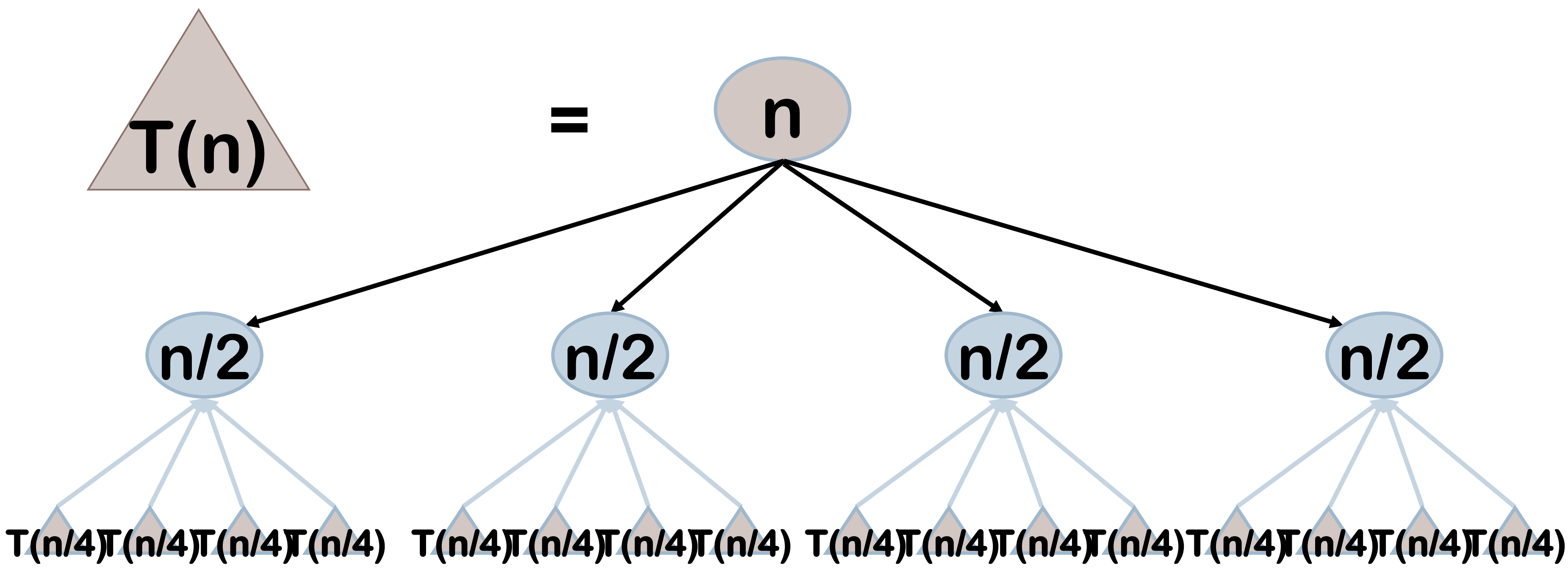
递归与分治算法基本思想

- **将小规模问题的解合并为一个更大规模的问题的解**
  - 若子问题的规模仍然不够小，则再划分为k个子问题
  - 自底向上逐步求出原来问题的解



递归与分治算法基本思想

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。



递归的概念

- 递归算法：直接或间接地调用自身的算法
- 递归函数:用函数自身给出定义的函数
- 分治法产生的子问题往往是原问题的较小模式，这为使用递归技术提供了方便
  - 反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解
  - 自然导致递归过程的产生
- 分治与递归像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法

（一） 递归的概念

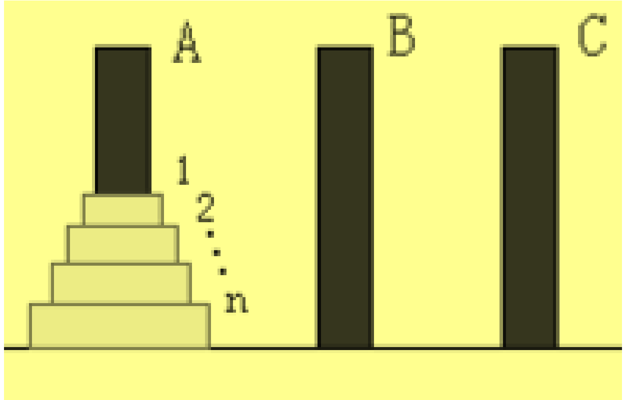
例1 Hanoi塔问题

设a, b, c是3个塔座。开始时，在塔座a上有一叠共n个圆盘，这些圆盘自下而上，由大到小地叠在一起。各圆盘从小到大编号为1, 2, …, n, 现要求将塔座a上的这一叠圆盘移到塔座b上, 并仍按同样顺序叠置。在移动圆盘时应遵守以下移动规则：

规则1： 每次只能移动1个圆盘；

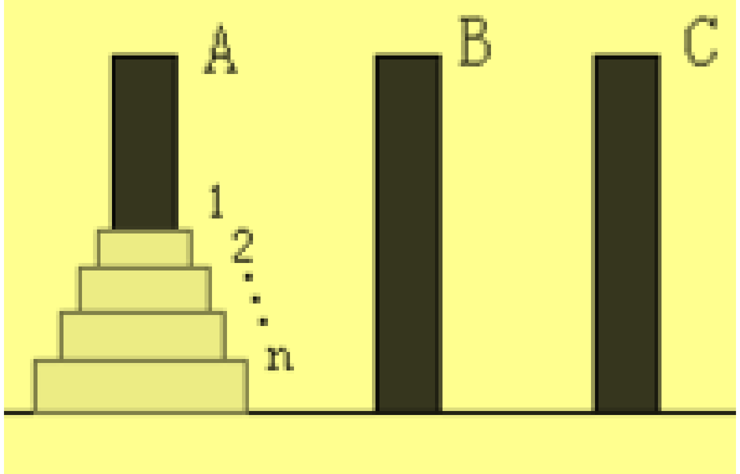
规则2： 任何时刻都不允许将较大的圆盘压在较小的圆盘之上；

规则3： 在满足移动规则1和2的前提下，可将圆盘移至a, b, c中任一塔座上。



（一） 递归的概念

```
void hanoi(int n, int a, int b, int c)
  {
     if (n > 0)
     {
       hanoi(n-1, a, c, b);
       move(a,b);
       hanoi(n-1, c, b, a);
     }
  }
```



（一） 递归的概念

```c
int height(BTree *p)
{
    int hi = 0,lh = 0,rh = 0;
    if (p == NULL)
        hi = 0;
    else
    {
        if (p->lchild ==NULL)
            lh = 0;
        else
            lh = height(p->lchild);//递归求解左子树的高度
        if (p->rchild ==NULL)
            rh = 0;
        else
            rh = height(p->rchild);//递归求解右子树的高度
        hi = lh>rh ? (lh + 1) : (rh + 1);
    }
    return hi;
}
```

递归的概念

- **例2 阶乘函数**

  阶乘函数可递归地定义为：

边界条件

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

递归方程

边界条件与递归方程是递归函数的二个要素，递归函数只有具备了这两个要素，才能在有限次计算后得出结果。

递归的概念

- **例2 阶乘函数**

阶乘函数可递归地定义为：

边界条件

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

递归方程

边界条件与递归方程是递归函数的二个要素，递归函数只有具备了这两个要素，才能在有限次计算后得出结果。

```
int fact(int n) {
    if (n < 0)
        return 0;
    else if(n == 0 || n == 1)
        return 1;
    else
        return n * fact(n - 1);
}
```

```
int fact (int n)
 {
     if( n < 0)
         return 0;
     if( n == 0)
         return 1;
     int a=1;
     for(int i=2; i++; i<=n)
         a=a*i;
     return (a);
 }
```

递归的概念

例3：Fibonacci数列

- 无穷数列1，1，2，3，5，8，13，21，34，55，……，称为Fibonacci数列。它可以递归地定义为：

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

边界条件

递归方程

第n 个Fibonacci 数可递归地计算如下：

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

非递归调用

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

递归的概念

- 例4：Ackerman函数
  - 当一个函数及它的一个变量是由函数自身定义时，称其为双递归函数
    - Ackerman函数A(n，m)定义如下：

$$A(n,m)=\begin{cases}2 & n=1,m=0\\ 1 & n=0\\ n+2 & m=0\\ A(A(n-1,m),m-1) & n,m\geq 1\end{cases}$$

例3可以找到非递归定义实现，Ackerman函数则无此类定义

递归的概念

- A(n, m) 自变量m 的每一个值都定义了一个单变量函数

  - m=0时， A(n,0)=n+2
  - m=1时， A(n,1)=A(A(n-1,1),0)=A(n-1,1)+2， A(1,1)=2 故： A(n, 1)=2*n
  - m=2时， A(n,2)=A(A(n-1,2),1)=2A(n-1,2)， A(1,2)=A(A(0,2),1)=A(1,1)=2

    故： A(n, 2)= 2^n
  - m=3时， 类似的可以推出$A(n,3)=2^{2^{\cdot^{\cdot^{2}}}}$,其中2的层数为n
  - m=4时， A(n,4)的增长速度非常快， 以至于没有适当的数学式子来表示这一函数

华中科技大学 人工智能与自动化学院

A-ambitious 积极进取 U-unique 锐意创新 T-tolerant 包容并蓄 O-open 开放交流

尾递归的概念

```c
int fact(int n) {
    if (n < 0)
        return 0;
    else if(n == 0 || n == 1)
        return 1;
    else
        return n * fact(n - 1);
}
```

```c
int facttail(int n, int res)
{
    if (n < 0)
        return 0;
    else if(n == 0)
        return 1;
    else if(n == 1)
        return res;
    else
        return facttail(n - 1, n *res);
}
```

如果一个函数中所有递归形式的调用都出现在函数的末尾，我们称这个递归函数是尾递归的。当编译器检测到一个函数调用是尾递归的时候，它就覆盖当前的活动记录而不是在栈中去创建一个新的。不用尾递归，函数的堆栈耗用难以估量，需要保存很多中间函数的堆栈。其关键就是通过参数传递结果，达到不压栈的目的。

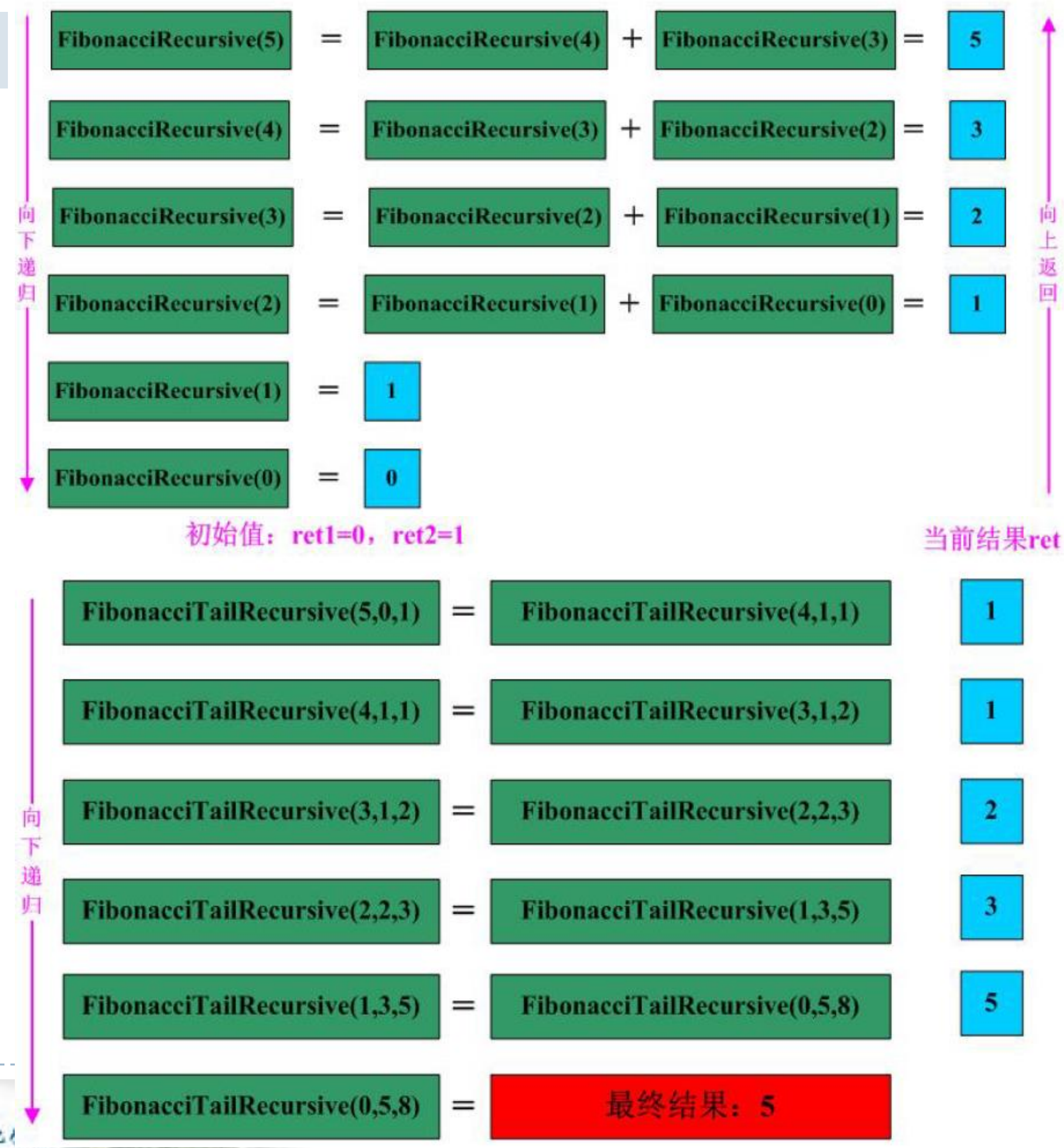
（一） 尾递归的概念

0、1、1、2、3、5、8、13、21、34

```
int Fibo (int n)
 {
    if( n < 2)
       return n;
    return (Fibo (n-1)+Fibo (n-2));
 }
```

```
int FiboTail (int n,int ret1,int ret2)
{
  if(n==0)
     return ret1;
   return FiboTail (n-1,ret2,ret1+ret2);
}
```

初始：ret1=0，ret2=1



递归小结

- **优点**

结构清晰，可读性强，而且容易用数学归纳法来证明算法的正确性，因此它为设计算法、调试程序带来很大方便。

- **缺点**

递归算法的运行效率较低，无论是耗费的计算时间还是占用的存储空间都比非递归算法要多。

递归小结

- 解决办法：在递归算法中消除递归调用，使其转化为非递归算法
    - 采用一个用户定义的栈来模拟系统的递归调用工作栈

      该方法通用性强，但本质上还是递归，只不过人工做了本来由编译器做的事情，优化效果不明显

    - 用递推来实现递归函数

后一种方法在时空复杂度上有较大改善，但适用范围有限

分治法一般步骤

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

平衡(balancing)子问题思想

人们从大量实践中发现，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个子问题的处理方法是行之有效的，它几乎总是比子问题规模不等的做法要好。

分治法适用条件

- 分治法所能解决的问题一般具有以下几个特征：
  - 规模缩小到一定的程度，可容易解决
  - 子问题为相同问题，具有最优子结构性质
  - 子问题的解可以合并为该问题的解
  - 子问题间相互独立，不含公共子问题

这条特征涉及到分治法的效率，如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然也可用分治法，但一般用动态规划较好。

分治法的复杂性分析

- 规模为n的问题被分成k个规模为n/m的子问题：
  - 设分解阈值n0=1,adhoc解规模为1的问题耗费1个单位时间
  - 设分解为k个子问题及用merge将子问题的解合并为原问题需用f(n)个单位时间
  - T(n)表示分治法解规模|p|=n的问题所需计算时间

采用分治法将一个规模为n的问题分成a个规模为n/b的子问题进行求解

复杂度分析

$$T(n)=\begin{cases} O(1) & n=1 \\ aT(n/b)+f(n) & n>1 \end{cases}$$

代入递推得

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

若 $f(n)=O(n^d),\ d\geq 0$

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

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]，现要在这n个元素中找出一特定元素x。
- 分析：

  - 该问题的规模缩小到一定的程度就可以容易地解决；
  - 该问题可以分解为若干个规模较小的相同问题；
  - 分解出的子问题的解可以合并为原问题的解；
  - 分解出的各个子问题是相互独立的。

分析：很显然此问题分解出的子问题相互独立，即在a[i]的前面或后面查找x是独立的子问题，因此满足分治法的第四个适用条件。

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]，现要在这n个元素中找出一特定元素x。
- 据此容易设计出二分搜索算法：l:left,r:right;

```
template<class Type>
int BinarySearch(Type a[], const
Type& x, int l, int r)
{
     while (r >= l){
        int m = (l+r)/2;
        if (x == a[m]) return m;
        if (x < a[m]) r = m-1;
        else  l = m+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) 。

大整数的乘法

- 请设计一个有效的算法，可以进行两个n位大整数的乘法运算
  - 小学的方法：O(n2)　　✘效率太低
  - 分治法：

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

大整数的乘法

- 请设计一个有效的算法，可以进行两个n位大整数的乘法运算
  - 小学的方法：$O(n^2)$ ✗效率太低
  - 分治法： $XY = ac\ 2^n + (ad+bc)\ 2^{n/2} + bd$

为了降低时间复杂度，必须减少乘法的次数。

1. $XY = ac\ 2^n + ((a-c)(b-d)+ac+bd)\ 2^{n/2} + bd$
2. $XY = ac\ 2^n + ((a+c)(b+d)-ac-bd)\ 2^{n/2} + bd$

复杂度分析

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

$T(n)=O(n^{\log 3}) = O(n^{1.59})$ ✓较大的改进

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+1位的结果，使问题的规模变大，故不选择第2种方案。

例2-大整数的乘法

- 请设计一个有效的算法，可以进行两个n位大整数的乘法运算
  - 小学的方法：O(n2) ✘效率太低
  - 分治法:O(n1.59) ✓较大的改进
  - 更快的方法??

✓ 如果将大整数分成更多段，用更复杂的方式把它们组合起来，将有可能得到更优的算法。

✓ 最终的，这个思想导致了**快速傅利叶变换**(Fast Fourier Transform)的产生。该方法也可以看作是一个复杂的分治算法。

矩阵乘法

◆传统方法：$O(n^3)$

A和B的乘积矩阵C中的元素C[i,j]定义为:$C[i][j]=\sum_{k=1}^{n}A[i][k]B[k][j]$

若依此定义来计算A和B的乘积矩阵C，则每计算C的一个元素C[i][j]，需要做n次乘法和n-1次加法。因此，算出矩阵C的 个元素所需的计算时间为$O(n^3)$

矩阵乘法–简单的分治策略

◆传统方法：O(n³)

◆分治法:

使用与上例类似的技术，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:

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

复杂度分析

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

T(n)=O(n³) ×没有改进☹

合并算法的复杂性为O(n²)?

矩阵乘法—Strassen

为了降低时间复杂度，必须减少乘法的次数。

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

$$\Rightarrow$$

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

复杂度分析

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

$T(n)=O(n^{\log 7}) =O(n^{2.81})$ ✓ **较大的改进☺**

矩阵乘法—Strassen

◆传统方法：$O(n^3)$
◆分治法: $O(n^{2.81})$
◆更快的方法??

➢Hopcroft和Kerr已经证明(1971)，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，就不能再基于计算2×2矩阵的7次乘法这样的方法了。或许应当研究 3 × 3 或 5 × 5 矩阵的更好算法。

➢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。目前最好的计算时间上界是 **$O(n^{2.376})$**

➢是否能找到$O(n^2)$的算法？？？目前为止还没有结果。

合并排序

- **基本思想**：将待排序元素分成大小大致相同的2个子集合，分别对2个子集合进行排序，最终将排好序的子集合合并成为所要求的排好序的集合。

```
void MergeSort(Type a[], int left, int right)
  {
    if (left<right) {//至少有2个元素
    int i=(left+right)/2;  //取中点
    mergeSort(a, left, i);
    mergeSort(a, i+1, right);
    merge(a, b, left, i, right);  //合并到数组b
    copy(a, b, left, right);    //复制回数组a
    }
  }
```

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

T(n)=O(nlogn) 渐进意义下的最优算法

合并排序

- 算法mergeSort的递归过程可以消去。

初始序列	[49] [38] [65] [97] [76] [13] [27]
第一步	[38 49] [65 97] [13 76] [27]
第二步	[38 49 65 97] [13 27 76]
第三步	[13 27 38 49 65 76 97]

快速排序

```c
int qusort(int s[],int start,int end)    //自定义函数 qusort()
{
    int i,j;    //定义变量为基本整型
    i=start;    //将每组首个元素赋给i
    j = end;    //将每组末尾元素赋给j
    s[0]=s[start];    //设置基准值
    while(i<j)
    {
        while(i<j&&s[0]<s[j])
        j--;    //位置左移
        if(i<j)
        {
            s[i]=s[j];    //将s[j]放到s[i]的位置上
            i++;    //位置右移
        }
        while(i<j&&s[i]<=s[0])
            i++;    //位置左移
        if(i<j)
        {
            s[j]=s[i];    //将大于基准值的s[j]放到s[i]位置
            j--;    //位置左移
        }
    }
    s[i]=s[0];    //将基准值放入指定位置
    if (start<i)
        qusort(s,start,j-1);    //对分割出的部分递归调用qusort()函数
    if (i<end)
        qusort(s,j+1,end);
    return 0;
}
```

- **最坏时间复杂度：O(n²)**

  每次极不对称划分（1，n-1）

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

- **最好时间复杂度：O(nlogn)**

  每次对称划分

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

循环赛日程表

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

按分治策略，将所有的选手分为两半，n个选手的比赛日程表就可以通过为n/2个选手设计的比赛日程表来决定。递归地对选手进行分割，直到只剩下2个选手时。这时只要让这2个选手进行比赛就可以了。

循环赛日程表

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

按分治策略，将所有的选手分为两半，n个选手的比赛日程表就可以通过为n/2个选手设计的比赛日程表来决定。递归地对选手进行分割，直到只剩下2个选手时。这时只要让这2个选手进行比赛就可以了。

1	2
2	1

(a) $2^k(k=1)$个选手比赛

加2

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

(b) $2^k(k=2)$个选手比赛

加4

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

(c) $2^k(k=3)$个选手比赛

- 求解过程是自底向上的迭代过程，其中图(c)左上角和左下角分别为选手1至选手4以及选手5至选手8前3天的比赛日程
- 将左上角部分的所有数字按其对应位置抄到右下角，将左下角的所有数字按其对应位置抄到右上角，这样，就分别安排好了选手1至选手4以及选手5至选手8在后4天的比赛日程，如图(c)所示。具有多个选手的情况可以依此类推。

循环赛日程表

```
const int maxn = 10000;
int a[maxn][maxn];
inline void dfs(int n,int k)
{
    if(n == 2)
    {
        a[k][0] = k+1;
        a[k][1] = k+2;
        a[k+1][0] = k+2;
        a[k+1][1] = k+1;
    }
    else
    {
        dfs(n/2,k);
        dfs(n/2,k+n/2);
        for(int i = k; i < k+n/2; i++)
        {
            for(int j = n/2; j < n; j++) a[i][j] = a[i+n/2][j-n/2];
        }
        for(int i = k+n/2; i < k+n; i++)
        {
            for(int j = n/2; j < n; j++) a[i][j] = a[i-n/2][j-n/2];
        }
    }
}
```

时间复杂度

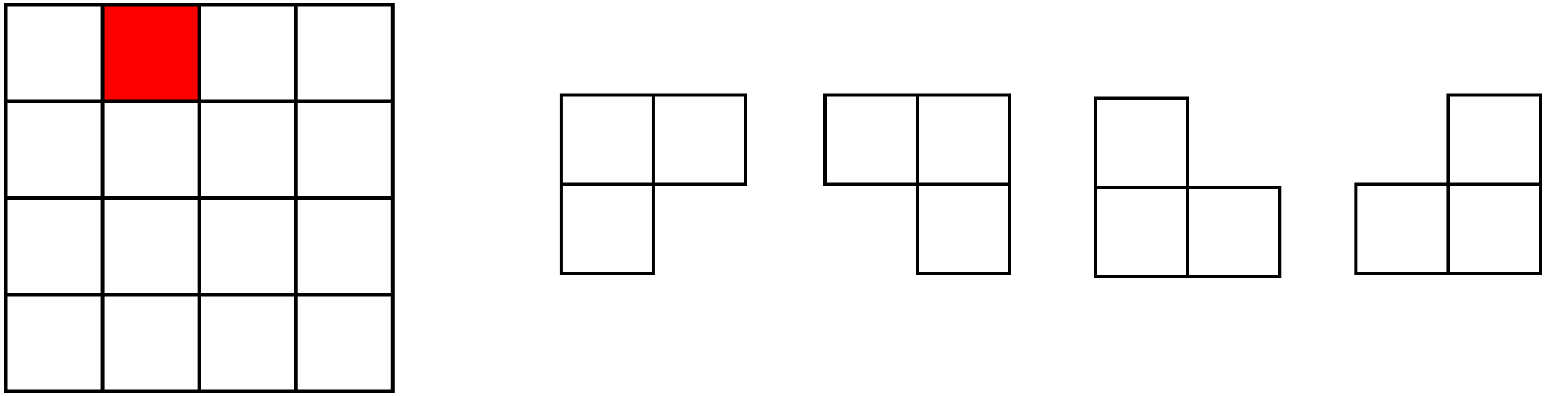
$$T(n) = \begin{cases} O(1) & n < 4 \\ 2T(\frac{n}{2}) & n \geq 4 \end{cases}$$

$$T(n) = O(nlogn)$$

```
int main()
{
    int n;
    while(scanf("%d",&n)!=EOF)
    {
        dfs(n,0);
        for(int i = 0; i < n; i++)
        {
            for(int j = 0; j < n; j++) printf("%d ",a[i][j]);
            printf("\n");
        }
    }
    return 0;
}
```

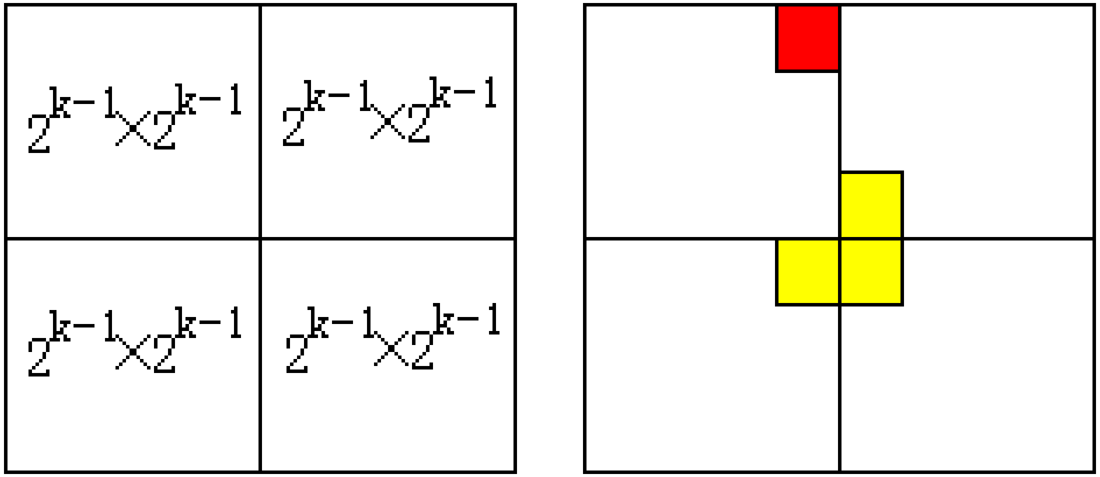
棋盘覆盖

- 在一个$2^k \times 2^k$ 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，要用图示的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。



棋盘覆盖

- 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(a)所示。
- 特殊方格必位于4个较小子棋盘之一中，其余3个子棋盘中无特殊方格。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，如 (b)所示，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，直至棋盘简化为棋盘1×1。



(a) (b)

棋盘覆盖

复杂度分析

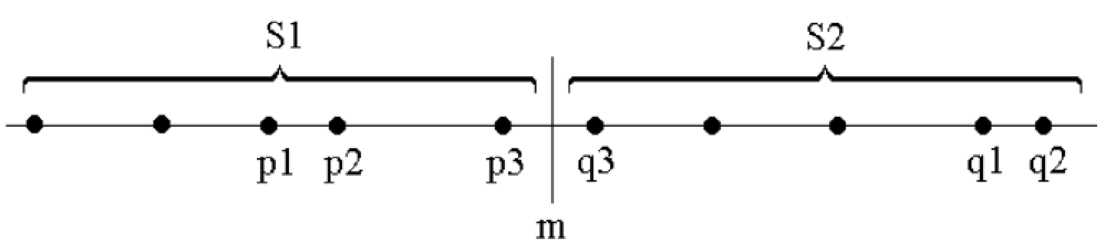
$$T(k)=\begin{cases} O(1) & k=0 \\ 4T(k-1)+O(1) & k>0 \end{cases}$$

$T(n)=O(4^k)$ 渐进意义下的最优算法

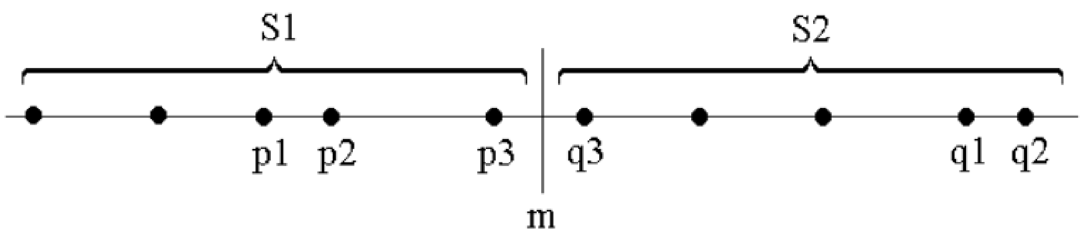
例4-最近点对问题

问题：给定平面上n个点S中，找其中一对点，使得其在n个点组成的所有点对中，该点对间的**距离最小**

- 为了使问题易于理解和分析，先来考虑一维的情形。此时，S中的n个点退化为x轴上的n个实数x1,x2,…,xn，最接近点对即为这n 个实数中相差最小的2个实数
  - 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}，并设d=min{|p1-p2|,|q1-q2|}，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
  - 能否在线性时间内找到p3, q3？



例4–最近点对问题



- 能否在线性时间内找到p3, q3？

➢ 如果S的最接近点对是$\{p_3,q_3\}$，即$|p_3-q_3|<d$，则p_3和q_3两者与m的距离不超过d，即$p_3\in(m-d,m]$，$q_3\in(m,m+d]$

➢ 由于在S_1中，每个长度为d的半闭区间至多包含一个点（否则必有两点距离小于d），并且m是S_1和S_2的分割点，因此$\mathbf{(m-d,m]}$中至多包含S中的一个点。由图可以看出，如果$(m-d,\ m]$中有S中的点，则此点就是S_1中最大点。对$(m,m+d]$区间同理。

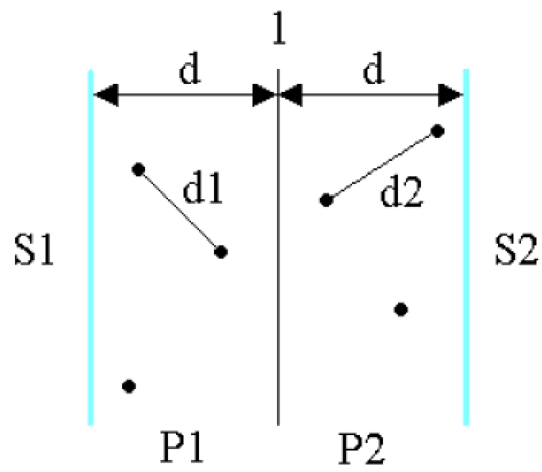
➢ 因此，我们用线性时间就能找到区间$(m-d,m]$和$(m,m+d]$中所有点，即p_3和q_3。从而我们用线性时间就可以将S_1的解和S_2的解合并成为S 的解

例4–最近点对问题

```
double cpair1(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={x∈S|x<=m},
    //S2={x∈S|x>m}
2、 d1=cpair1(S1); d2=cpair1(S2);
3、 p=max(S1); q=min(S2);
4、 dm=min(d1,d2,q-p);
      returen d;
```

例4–最近点对问题

- 进一步，考虑二维的情形
  - 选取一垂直线l:$x=m$来作为分割直线，其中m为S 中各点x坐标的中位数，由此将S 分割为S_1和S_2
  - 递归地在S1和S2上找出其最小距离d1和d2，并设d=min{d1,d2}，S中的最接近点对或者是d，或者是某个{p,q}，其中p∈S1且q∈S2
  - 能否在线性时间内找到p3, q3？



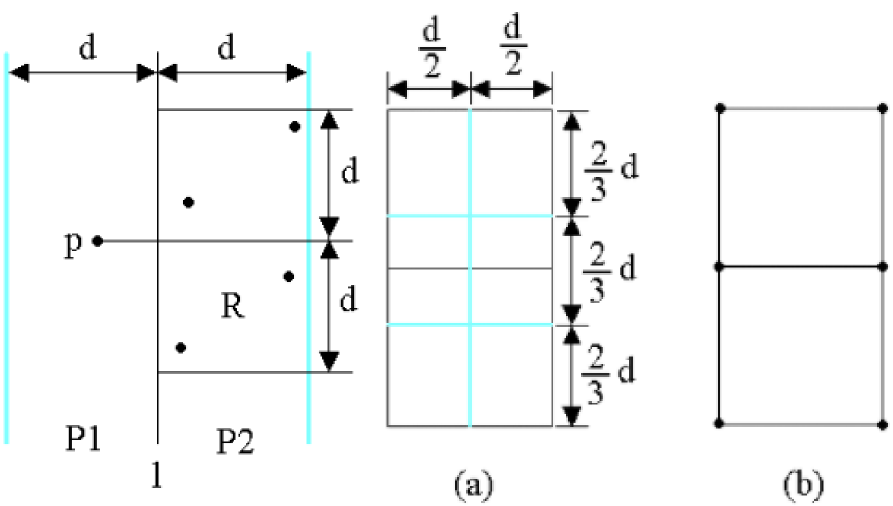
例4–最近点对问题

- 能否在线性时间内找到p3, q3？

➢ 考虑P_1中任意一点p，它若与P_2中的点q构成最接近点对的候选者，则必有distance(p，q) < d，满足这个条件的P_2中的点一定落在一个$d\times 2d$的矩形R中

➢ 由d的意义可知，P_2中任何2个S中的点的距离都不小于d，由此可以推出矩形R中最多只有6个S中的点

➢ 因此，在分治法的合并步骤中最多只需要检查$6\times n/2=3n$个候选者



反证： 将矩形R的长为$2d$的边3等分，将它的长为d的边2等分，由此导出6个$(d/2)\times(2d/3)$的矩形。若矩形R中有多于6个S中的点，则由鸽舍原理易知至少有一个$(d/2)\times(2d/3)$的小矩形中有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

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

故disntance(u,v)<d

例4-最近点对问题

- 为了确切地知道要检查哪6个点，可以将p 和P_2中所有S_2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S_2中点一定在矩形R中，所以它们在直线l 上的投影点距p在l 上投影点的距离小于d
  - 由上面的分析可知，这种投影点最多只有6 个
- 因此，若将$P1$和$P2$中所有S中点按其y坐标排好序，则对$P1$中所有点，对排好序的点列作一次扫描，就可以找出所有最接近点对的候选
  - 对P_1中每一点最多只要检查P_2中排好序的相继6 个点

例4–最近点对问题

```
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、扫描X 以及对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 是按该扫描方式找到的点对间的最小距离；

6、d=min(dm,dl);

return d;

例4–最近点对问题

时间复杂度

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

$$T(n)=O(nlogn)$$

Next:贪心算法