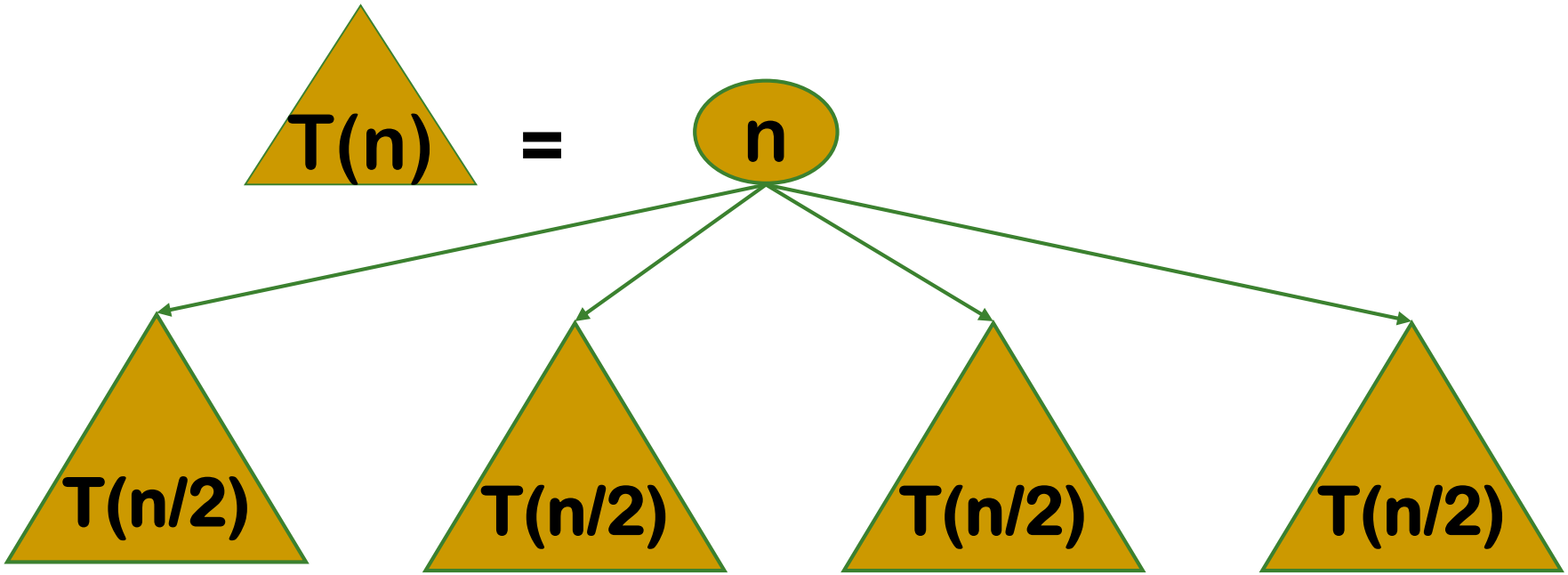
# 第2章 递归与分治策略

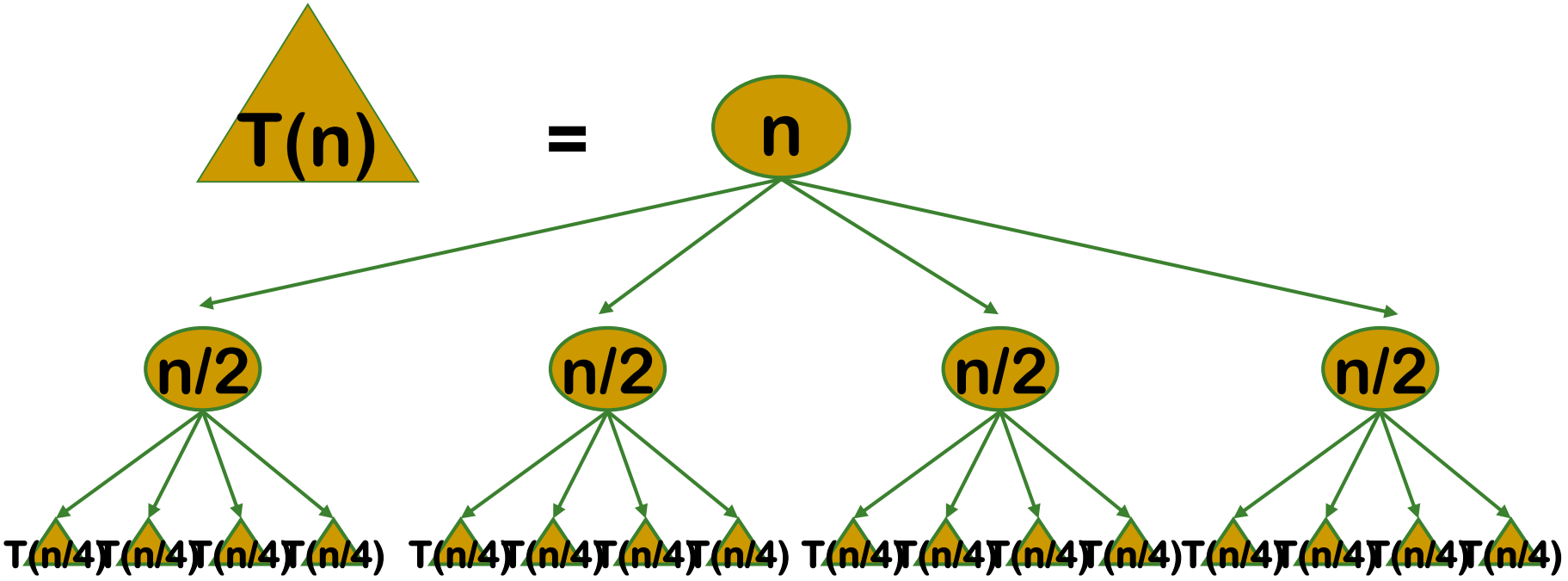
# 算法总体思想

- 将一个规模为n的问题分解为k个规模较小的子问题；



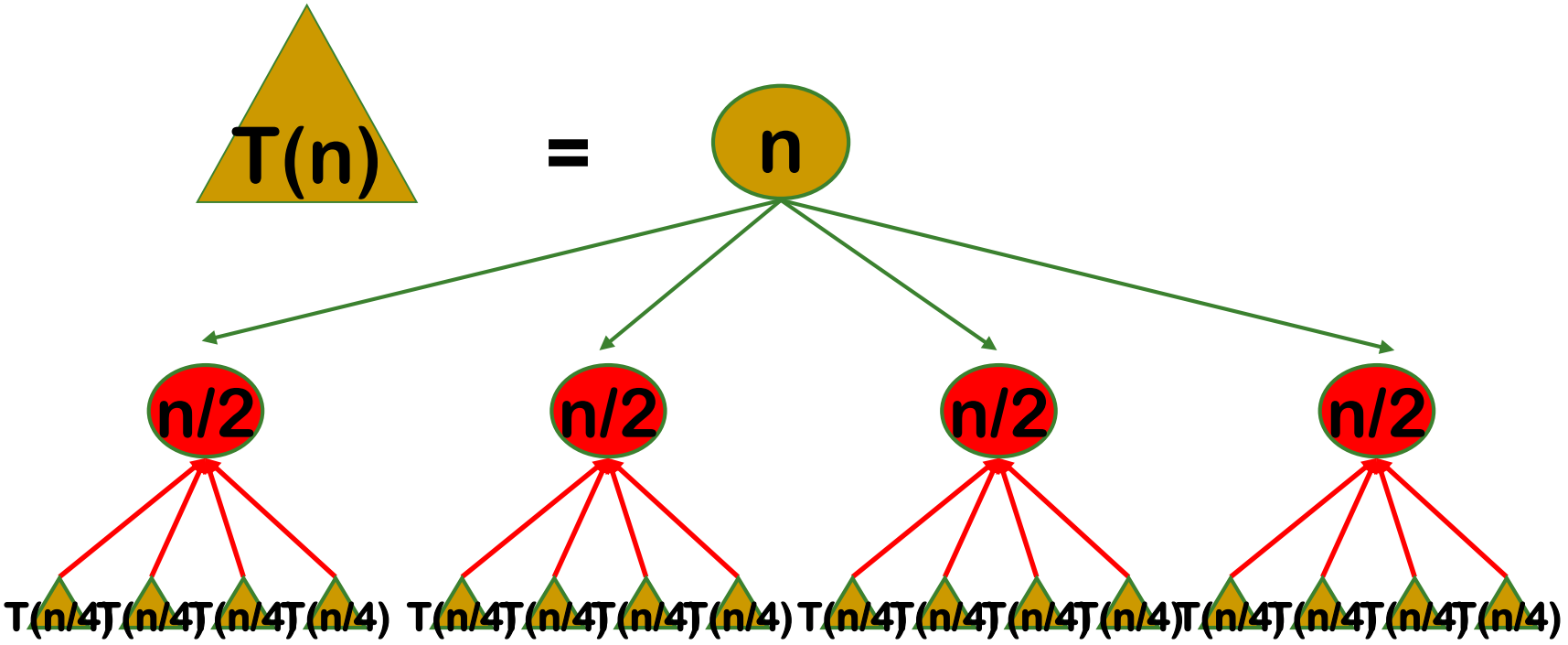
# 算法总体思想

- 对这k个子问题分别求解。如果子问题的规模仍然不够小，则再将子问题继续划分为k个更小的子问题，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

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。

凡治众如治寡，分数是也。

----孙子兵法

# 2.1 递归的概念

- 直接或间接地调用自身的算法称为**递归算法**。用函数自身给出定义的函数称为**递归函数**。
- 由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，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。这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。

下面来看几个实例。

# 2.1 递归的概念

**例1 阶乘函数**

阶乘函数可递归地定义为：

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

边界条件

递归方程

边界条件与递归方程是递归函数的两个要素，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。

```
public static int factorial(int n){
        if(n==0) return 1;
        return n*factorial(n-1);
}
```

边界条件

递归方程

# 2.1 递归的概念

## 例2 Fibonacci数列

无穷数列1，1，2，3，5，8，13，21，34，82，…，被称为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：

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

边界条件

递归方程

# 2.1 递归的概念

## 例3 Ackerman函数

当一个函数及它的一个变量是由函数自身定义时，称这个函数是**双递归函数。**

**Ackerman函数A(n，m)定义如下：**

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

# 2.1 递归的概念

## 例3 Ackerman函数

前2例中的函数都可以找到相应的非递归方式定义：

$$n!=1\cdot 2\cdot 3\cdot\cdots\cdot(n-1)\cdot n$$

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

但本例中的Ackerman函数却无法找到非递归的定义。

# 2.1 递归的概念

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

- A(n，m)的自变量m的每一个值都定义了一个单变量(n)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*1
    = A(A(n-2,1), 0)+2
    = A(n-2,1)+2 +2
    = A(n-2,1)+2*2
    …
    = A(n-i,1)+2*i
    …
    = A(n-(n-1), 1)+2*(n-1)
    = A(1,1)+2*(n-1)

# 2.1 递归的概念

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

- m=1时(续)，A(n,1) = A(1,1)+2*(n-1)

  = A(A(0,1), 0) + 2n-2

  = A(1,0)+2n-2

  = 2+2n-2

  = 2n

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

- m=1时，A(n,1) = 2n
- m=2时，A(n,2)= A(A(n-1,2),1)

= $2^1$*A(n-1,2)

= $2^1$*A(A(n-2,2),1)

= $2^1$*2*A(n-2,2)

= $2^2$*A(n-2,2)

…

= $2^i$*A(n-i,2)

…

= $2^{n-1}$*A(n-(n-1),2)

= $2^{n-1}$*A(1,2)

= $2^{n-1}$*A(A(0,2),1)

= $2^{n-1}$*A(1,1)

= $2^n$

# 2.1 递归的概念

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

- m=1时，A(n,1) = 2n
- m=2时，$A(n,2)=2^n$
- m=3时，类似的可以推出 $A(n,3)=\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{n}$
- 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。

# 2.1 递归的概念

- 定义单变量的Ackerman函数A(n)为，A(n)=A(n，n)。

  A(0)=1, A(1)=2, A(2)=4, A(3) = 16, A(4) = $\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{65536}$

- 定义其拟逆函数α(n)为：α(n)=min{k | A(k)≥n}。即α(n)是使n≤A(k)成立的最小的k值。
- α(n)在复杂度分析中常遇到。对于通常所见到的正整数n，有**α(n)≤4**。但在理论上α(n)没有上界，随着n的增加，它以难以想象的慢速度趋向正无穷大。

# 2.1 递归的概念

## 例4 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)表示在全排列perm(X)的每一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)由$(r_1)$perm$(R_1)$，$(r_2)$perm$(R_2)$，…，$(r_n)$perm$(R_n)$构成。

# 例4　排列问题

**全排列算法**:

```
public static void Perm(int[ ] list, int k, int m)
 {
      if(k==m)
           for(int i=0;i<=m;i++)
          print(list[i]);
      else
           for(int i=k;i<=m;i++)
         swap(list[k], list[i]);
         Perm(list, k+1, m);
         swap(list[k], list[i]);
}
```

时间复杂度T(n) = n*T(n-1)

解得: T(n) = n!

Perm(list, k, m)递归地产生所有前缀是list[k:m]的全排列。
主函数是Perm(list, 0, n-1);

# 2.1 递归的概念

## 例5 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。

# 2.1 递归的概念

## 例5 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。观察例题：

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。

## 例5 整数划分问题

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:

(1) q(n,1)=?

(2) q(n,m)= ? , n<m

(3) q(n,n)=?

(4) q(n,m)=? , n>m>1

# 2.1 递归的概念

## 例5 整数划分问题

将最大加数$n_1$不大于m的划分个数记作q(n,m)。q(n,m)的递归关系:

(1) q(n,1)=1

当最大加数$n_1$不大于1时，任何正整数n只有一种划分形式，加数全是1。

(2)q(n,m)= q(n,n),n<m,最大加数$n_1$实际上不能大于n。

(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$n_1$=m的划分和$n_1 \le m-1$ 的划分组成。

# 2.1 递归的概念

## 例5 整数划分问题

q(n,m)递归关系为：

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

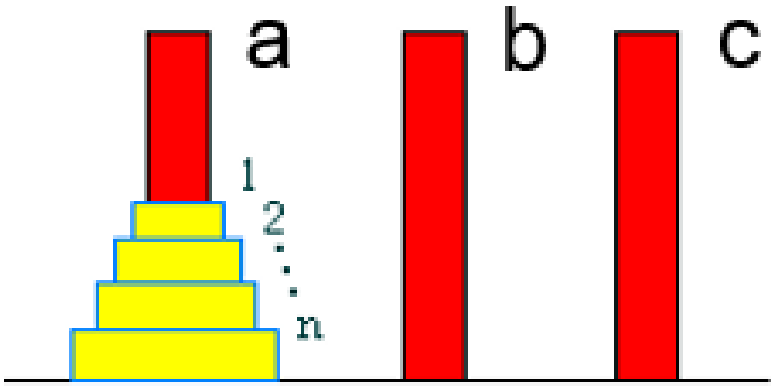
正整数n的划分数 p(n)=q(n,n)。

# 2.1 递归的概念

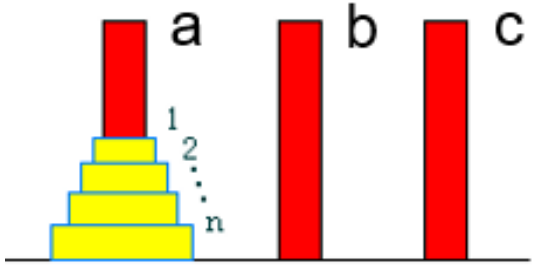
## 例6 Hanoi塔问题

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。



# 2.1 递归的概念




## 例6 Hanoi塔问题

在问题规模较大时，较难找到一般的方法，

因此我们尝试用递归技术来解决这个问题。

- 当n=1时，问题比较简单。此时，只要将编号为1的圆盘从塔座a直接移至塔座b上即可。
- 当n > 1时，需要利用塔座c作为辅助塔座。此时若能设法将n-1个较小的圆盘依照移动规则从塔座a移至塔座c，然后，将剩下的最大圆盘从塔座a移至塔座b，最后，再设法将n-1个较小的圆盘依照移动规则从塔座c移至塔座b。

由此可见，n个圆盘的移动问题可分为2次n-1个圆盘的移动问题，这又可以递归地用上述方法来做。由此可以设计出解Hanoi塔问题的递归算法。

# 2.1 递归的概念

## 例6 Hanoi塔问题

个数 源 目标 辅助

```
public static 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);
}
```

思考题：如果塔的个数变为a,b,c,d四个，现要将n个圆盘从a全部移动到d，移动规则不变，求移动步数最小的方案。

# 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

**解决方法：**在递归算法中消除递归调用，使其转化为非递归算法。

1. 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。
2. 用**递推**来实现递归函数。
3. 将一些递归转化为**尾递归**，从而迭代求出结果。

(当递归调用是整个函数体中最后执行的语句且它的返回值不属于表达式的一部分时，这个递归调用就是尾递归。)

后两种方法在时空复杂度上均有较大改善，但其适用范围有限。

# 递归算法的复杂性分析方法

**分析方法：**

（1）根据算法建立复杂度的递归方程。

（2）求解递归方程，得到时间复杂度。

如：阶乘函数：

```
int factorial(int n)
{
    if (n == 0) return 1;
    return n*factorial(n-1);
}
```

递归方程 $T(n)=\begin{cases}1 & n=0\\ T(n-1)+1 & n>0\end{cases}$

求解得 $T(n)=n+1=\Theta(n)$

求解递归方程有多种方法，这里介绍两种比较简单的求解方法：

1、递推求解

2、推测验证

# 递归方程的求解

## 1. 递推求解

**例** 汉诺塔求解算法：$T(n)=\begin{cases}1 & n=1\\ 2T(n-1)+1 & n>1\end{cases}$

递推展开：T(n) = 2T(n-1)+1

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

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

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

…

=$2^i$T(n-i)+1+2+$2^2$+$2^3$+…+$2^{i-1}$

…

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

= 1+2+$2^2$+$2^3$+…+$2^{n-2}$+$2^{n-1}$

= $2^n$-1 = O($2^n$)

# 1. 递推求解

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

设a,b,c均为正整数,

$a \geq 1$, b>1, c>0, 递归方程

递推展开

T(n) = aT(n/b)+c

= $a^2$T(n/$b^2$)+ac+c

= $a^3$T(n/$b^3$)+$a^2$c+ac+c

……

= $a^k$T(n/$b^k$)+$a^{k-1}$c+ … +ac+c　　$k=\log_b n$

= $a^k$T(1)+$a^{k-1}$c+ … +ac+c

= c($a^k$+$a^{k-1}$+ … +a+1)

$$= c\frac{a^{k+1}-1}{a-1}$$

有：$$T(n)=\begin{cases} c(\log_b n+1) & a=1 \\ c\dfrac{an^{\log_b a}-1}{a-1} & a\neq 1 \end{cases}$$

其中：$k=\log_b n$；$a^{\log_b n}=n^{\log_b a}$

# 1. 递推求解

设a,b,c均为正整数, $a \geq 1$, $b>1$, $c>0$, 递归方程

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

设$n>b^k$, $k>0$, 递推展开

$T(n)= ca^k+\sum_{i=0}^{k-1} a^i(cn/bi) = ca^k+cn\sum_{i=0}^{k-1}(a/b)^i$

a=b $T(n)=ca^k+cnk=cb^k+cn\log_b n=cn+cn\log_b n$

a>b $\log_b a>1, n^{\log_b a}>n$, $T(n) = ca^k+cn\dfrac{(a/b)^k-1}{(a/b)-1} \approx 2c\,n^{\log_b a}$

a<b $\log_b a<1, n^{\log_b a}<n$, $T(n) \approx 2cn$

解得：

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

# 1. 递推求解

一般化，设a,b,c均为正整数, $a \geq 1$, $b>1$, $c>0$, 递归方程

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

设$n=b^k$，递推展开

$T(n) = aT(n/b)+d(n)$

$= a^2T(n/b^2)+ad(n/b)+d(n)$

$= a^3T(n/b^3)+a^2d(n/b^2)+ad(n/b)+d(n)$

……

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

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

# 对这两项的讨论

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

- 第一项 $a^k$

由$n= b^k$, 可得$k=\log_b n$, 从而$a^k = a^{\log_b n} = n^{\log_b a}$, 对这个结果, 一般讲如b保持不动, a越大指数越高复杂度阶数越高；如a保持不动, b越大指数越低复杂度阶数越低。

# 对这两项的讨论

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

- 第二项 $\sum_{j=0}^{k-1} a^j\, d(b^{k-j})$

可积 $\forall n,m>0$有$f(nm)= f(n)\cdot f(m)$

一般讲不给定d(b)的解析式是无法解出∑的；如d(b)是可积函数, 则$d(b^{k-j})=(d(b))^{k-j}$,

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

$$= (d(b))^k \sum_{j=0}^{k-1} (a/d(b))^j = d^k(b)\frac{(a/d(b))^k - 1}{a/d(b) - 1} = \frac{a^k - d^k(b)}{a/d(b) - 1}$$

# 例：快速排序的复杂度分析

```
void QSort(int R[], int left, int right){
    if(left<right)
        int m=Partition(R,left,right);  //Partition复杂度为O(right-left+1)
        QSort(R,left,m-1);
        QSort(R,m+1,right);
}//Qsort
```

最好情况：$T(n)=\begin{cases} c & n=1 \\ 2T(n/2)+cn & n>1 \end{cases}$ $T(n)=\Theta(n\log n)$

最坏情况：$T(n)=\begin{cases} c & n=1 \\ T(n-1)+cn & n>1 \end{cases}$ $T(n)=\Theta(n^2)$

# 例：快速排序的复杂度分析

平均情况：

设T(n)代表快速排序对n个待排序记录进行排序平均时间，假设每次划分产生一个长度为r的子序列，另一个长度为n-r-1的子序列。则T(n)=n-1+T(r)+T(n-r-1)。

假定在1到n之间进行划分，选择了第p个位置为基准位置。在划分后，左边的子序列为{1，2，...，p-1}，右边的子序列为{p+1，p+2，...，n}。假设p在1到n之间的取值是等概率的，即为 1/n。则：

$$T(n) = n - 1 + \frac{1}{n}\sum_{p=1}^{n} T[p-1] + T[n-p]$$

$$= n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} T[k]$$

## 递推求解

$$T(n)=n-1+\frac{2}{n}\sum_{k=0}^{n-1}T[k] \qquad (1)$$

$$T(n-1)=n-2+\frac{2}{n-1}\sum_{k=0}^{n-2}T[k] \qquad (2)$$

由(1)和(2)可以得到：

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

将(n-1)T(n-1)移到右侧，然后左右两侧同除n(n+1)，可以得到：

$$\frac{T(n)}{n+1}=\frac{T(n-1)}{n}+2\frac{n-1}{n(n+1)}=\frac{T(n-1)}{n}+2\left[-\frac{1}{n}+\frac{2}{n+1}\right]$$

$$=\frac{T(n-2)}{n-1}+2\left[-\frac{1}{n-1}+\frac{2}{n}\right]+2\left[-\frac{1}{n}+\frac{2}{n+1}\right]=\frac{T(n-2)}{n-1}+2\left[-\frac{1}{n-1}+\frac{1}{n}+\frac{2}{n+1}\right]=\ldots$$

$$=\frac{T(2)}{3}+2\left[-\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{n}+\frac{2}{n+1}\right]$$

$$T(n)=2(n+1)[\ln n+O(1)]=2n\ln n+O(n)$$

## 2. 推测验证

先推测递归方程的一个解，然后用数学方法证明解的正确性。例如：

$$T(n)=\begin{cases} c_1 & n\le 6 \\ T(\frac{n}{2})+T(\frac{n}{3})+c_2 n & n>6 \end{cases}$$

推测：

首先，由于$T(n)>c_2 n$，所以$T(n)=\Omega(n)$。其次，由于$1/2+1/3<1$，可猜测$T(n)=O(n)$。

## 2. 推测验证

$$T(n)=\begin{cases} c_1 & n\le 6 \\ T(\frac{n}{2})+T(\frac{n}{3})+c_2 n & n>6 \end{cases}$$

验证：

只需证明$T(n)=O(n)$。用数学归纳法。

当$n<=6$时，$T(n)=c_1<=c_1 n$，$T(n)=O(n)$ 成立(取$c=c_1, n_0=1$)。

假设当$n<=k$时，$T(n)=O(n)$ 成立。即存在一个$c$和$n_0$，当$n>n_0$ 时，$T(n)<=cn$。

当$n=k+1$时，$T(n)=T(\frac{n}{2})+T(\frac{n}{3})+c_2 n \le c\frac{n}{2}+c\frac{n}{3}+c_2 n$

$$=(\frac{5}{6}c+c_2)n$$

取$c=\frac{5}{6}c+c_2$，即$c=6c_2$时，$T(n)\le cn$。

所以，取$c=\max\{c_1, 6c_2\}$，$n_0=1$，当$n>=n_0$ 时，$T(n)<=cn$。

即：$T(n)=O(n)$

# 分治法的适用条件

分治法所能解决的问题一般具有以下几个特征：

① 该问题的规模缩小到一定的程度就可以容易地解决；

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

③ 利用该问题分解出的子问题的解可以合并为该问题的解；

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。

# 分治法的基本步骤

```
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); //递归地解各子问题
    return merge(y1,...,yk); //将各子问题的解合并为原问题的解
}
```

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。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。

# 分治法的复杂性分析

一个分治法将规模为n的问题分成k个规模为n／m的子问题去解。设分解阀值$n_0$=1，且adhoc解规模为1的问题耗费1个单位时间。再设将原问题分解为k个子问题以及用merge将k个子问题的解合并为原问题的解需用f(n)个单位时间。用T(n)表示该分治法解规模为|P|=n的问题所需的计算时间，则有：

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

通过迭代法求得方程的解：$T(n)=n^{\log_m k}+\sum_{j=0}^{(\log_m n)-1} k^j f(n/m^j)$

# 分治法的复杂性分析

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

解得：$T(n)=n^{\log_m k}+\sum_{j=0}^{(\log_n n)-1} k^j f(n/m^j)$

**注意**：递归方程及其解只给出n等于m的方幂时T(n)的值，但是如果认为T(n)足够平滑，那么由n等于m的方幂时T(n)的值可以估计T(n)的增长速度。通常假定T(n)是单调上升的，从而当$m^i \le n < m^{i+1}$时，$T(m^i) \le T(n) < T(m^{i+1})$。

# 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。

分治法适应条件：

① **该问题的规模缩小到一定的程度就可以容易地解决；**

② 该问题可以分解为若干个规模较小的相同问题；

③ 分解出的子问题的解可以合并为原问题的解；

④ 分解出的各个子问题是相互独立的。

分析：如果n=1即只有一个元素，则只要比较这个元素和x就可以确定x是否在表中。因此这个问题满足分治法的第一个适用条件

# 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。

① 该问题的规模缩小到一定的程度就可以容易地解决；

② **该问题可以分解为若干个规模较小的相同问题；** ■

③ **分解出的子问题的解可以合并为原问题的解；** □

④ 分解出的各个子问题是相互独立的。

分析：比较x和a的中间元素a[mid]，若x=a[mid]，则x在L中的位置就是mid；如果x<a[mid]，由于a是递增排序的，因此假如x在a中的话，x必然排在a[mid]的前面，所以我们只要在a[mid]的前面查找x即可；如果x>a[i]，同理我们只要在a[mid]的后面查找x即可。□

无论是在前面还是后面查找x，其方法都和在a中查找x一样，只不过是查找的规模缩小了。■

这就说明了此问题满足分治法的第二个和第三个适用条件。

# 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。

① 该问题的规模缩小到一定的程度就可以容易地解决；
② 该问题可以分解为若干个规模较小的相同问题；
③ 分解出的子问题的解可以合并为原问题的解；
④ **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。

# 二分搜索技术

在 a[0] <= a[1] <= ... <= a[n-1] 中搜索 x，找到x时返回其在数组中的位置，否则返回-1

**二分搜索算法**：

```
public static int binarySearch(int [] a, int x, int n){
      int left = 0; int right = n - 1;
      while (left <= right)
            int middle = (left + right)/2;
            if (x == a[middle])    return  middle;
            if (x > a[middle])     left = middle + 1;
            else    right = middle - 1;
    return -1; // 未找到x
}
```

**思考题：给定a，用二分法设计出求$a^n$的算法。**

**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：$O(n^2)$ ✘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**

◆小

◆分

XY

**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☺**

为了降低时间复杂度，必须减少乘法的次数。

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

**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m+1位的结果，使问题的规模变大，故不选择第2种方案。

# 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: $O(n^{1.59})$ ✓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，对于大整数乘法，它能在O(nlogn)时间内解决。
- ➢是否能找到线性时间的算法？？？目前为止还没有结果。

# Strassen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n个元素所需的计算时间为$O(n^3)$

# Strassen矩阵乘法

◆传统方法：$O(n^3)$

◆分治法:

使用与

成4个大小

**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☹

由此可得：

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

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

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

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

# Strassen矩阵乘法

为了降低时间复杂度，必须减少乘法的次数。

**复杂度分析**

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

$T(n)=O(n^{\log 7})=O(n^{2.82})$ ✓ **较大的改进☺**

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

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

7次n/2阶矩阵乘法，和18次n/2阶矩阵的加减

# Strassen矩阵乘法

◆更快的方法??

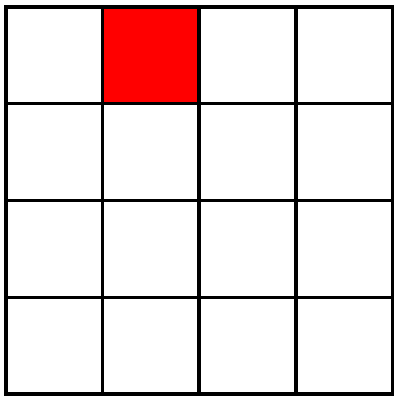
➢Hopcroft和Kerr已经证明(1971)，计算2个 2 × 2 矩阵的乘积，7次乘法是必要的。因此，要想进一步改进矩阵乘法的时间复杂性，就不能再基于计算2×2矩阵的7次乘法这样的方法了。或许应当研究 3 × 3 或 5 × 5 矩阵的更好算法。

➢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 $O(n^{2.382})$

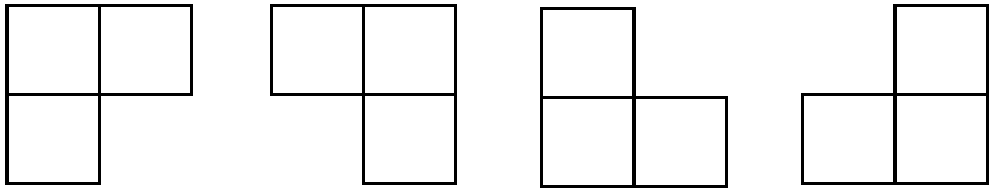
➢是否能找到$O(n^2)$的算法？？？目前为止还没有结果。

# 棋盘覆盖

在一个$2^k \times 2^k$个方格组成的棋盘中，恰有一个方格与其他方格不同，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。



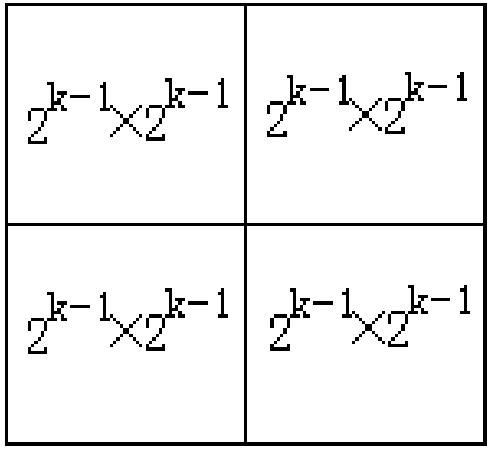
棋盘



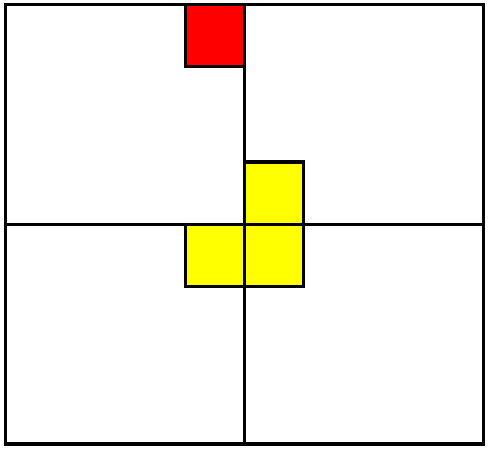
L型骨牌

# 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，如(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。

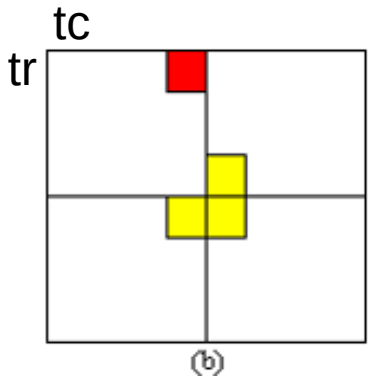


(a)

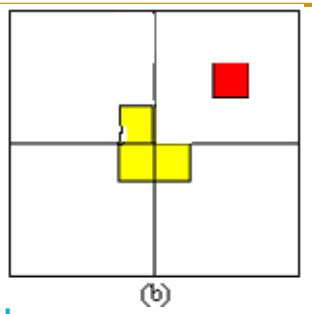

(b)

# 棋盘覆盖

```
public void chessBoard(int tr, int tc, int dr, int dc, int size)
// tr、tc: 棋盘左上角方格的行号、列号
// dr、dc: 特殊方格所在的行号、列号
// size: 2^k，棋盘每一行或每一列方格数
        if (size == 1)    return;
        int t = tile++,     // L型骨牌号，tile是一个全局变量
        s = size/2;         // 分割棋盘
      // 覆盖左上角子棋盘
      if (dr < tr + s && dc < tc + s)       // 特殊方格在此棋盘中
            chessBoard(tr, tc, dr, dc, s);
      else  // 此棋盘中无特殊方格
            board[tr + s - 1][tc + s - 1] = t; // 用t号L型骨牌覆盖右下角
            chessBoard(tr, tc, tr+s-1, tc+s-1, s);  // 覆盖其余方格
```



# 棋盘覆盖




```
// 覆盖右上角子棋盘
if (dr < tr + s && dc >= tc + s)   // 特殊方格在此棋盘中
        chessBoard(tr, tc+s, dr, dc, s);
else // 此棋盘中无特殊方格
        board[tr + s - 1][tc + s] = t; // 用 t 号L型骨牌覆盖左下角
        chessBoard(tr, tc+s, tr+s-1, tc+s, s); // 覆盖其余方格

// 覆盖左下角子棋盘
if (dr >= tr + s && dc < tc + s) // 特殊方格在此棋盘中
        chessBoard(tr+s, tc, dr, dc, s);
else
        board[tr + s][tc + s - 1] = t; // 用 t 号L型骨牌覆盖右上角
        chessBoard(tr+s, tc, tr+s, tc+s-1, s);  // 覆盖其余方格
```

# 棋盘覆盖

```
    // 覆盖右下角子棋盘
    if (dr >= tr + s && dc >= tc + s) // 特殊方格在此棋盘中
          chessBoard(tr+s, tc+s, dr, dc, s);
    else
         board[tr + s][tc + s] = t; // 用 t 号L型骨牌覆盖左上角
         chessBoard(tr+s, tc+s, tr+s, tc+s, s); // 覆盖其余方格
}
```

**复杂度分析**

T(k)是覆盖一个$2^k*2^k$棋盘所需的时间

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

$T(n)=O(4^k)$ 渐进意义下的最优算法

**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

**基本思**

别对2

要求的

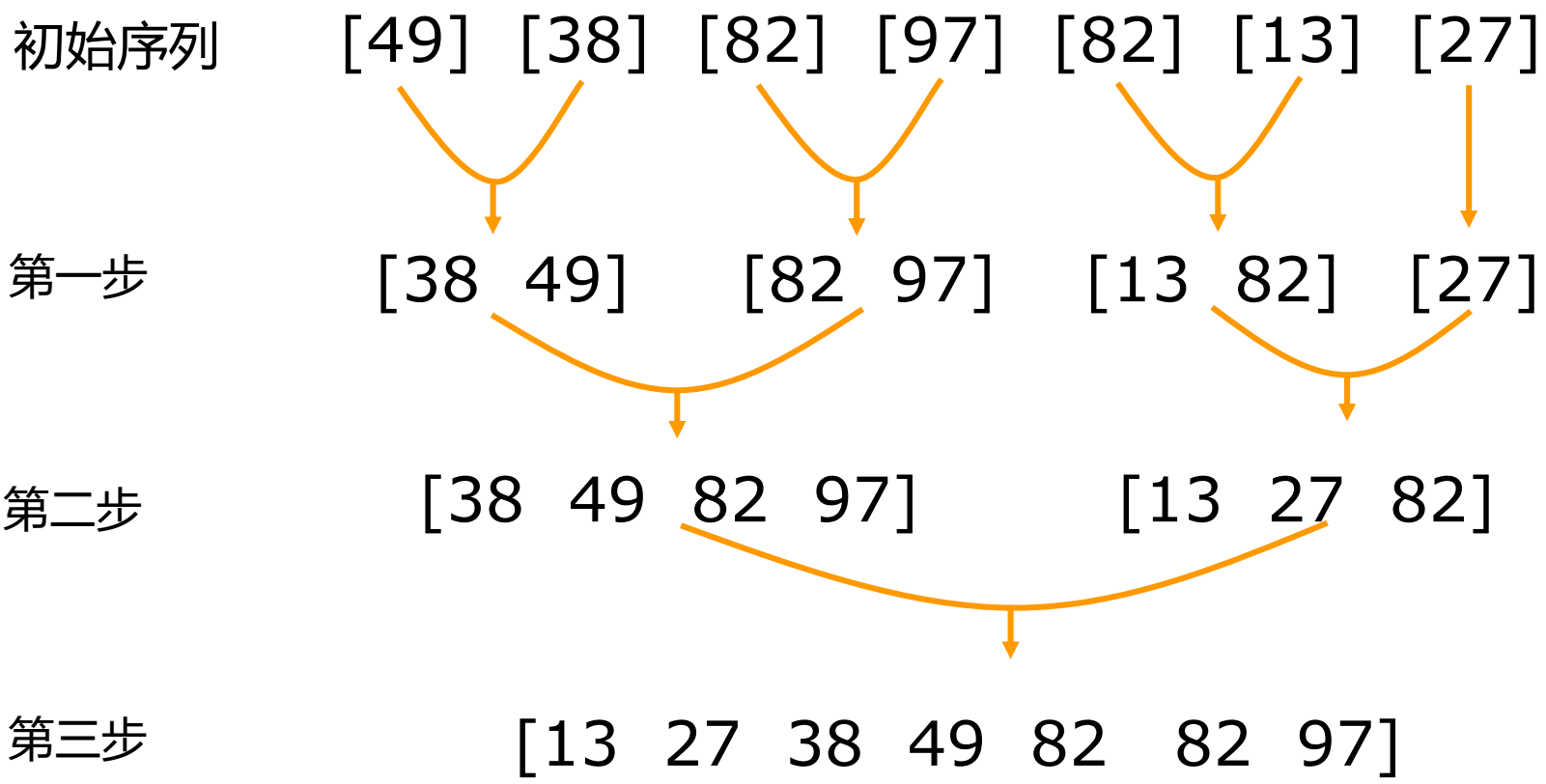
分

所

```
public static void mergeSort(Comparable 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
 }
```

# 合并排序

算法**mergeSort**的递归过程可以消去。



# 合并排序

- 最坏时间复杂度：O(nlogn)
- 平均时间复杂度：O(nlogn)
- 辅助空间：O(n)
- 稳定性：稳定 (不会改变相同元素的相对位置)

# 快速排序

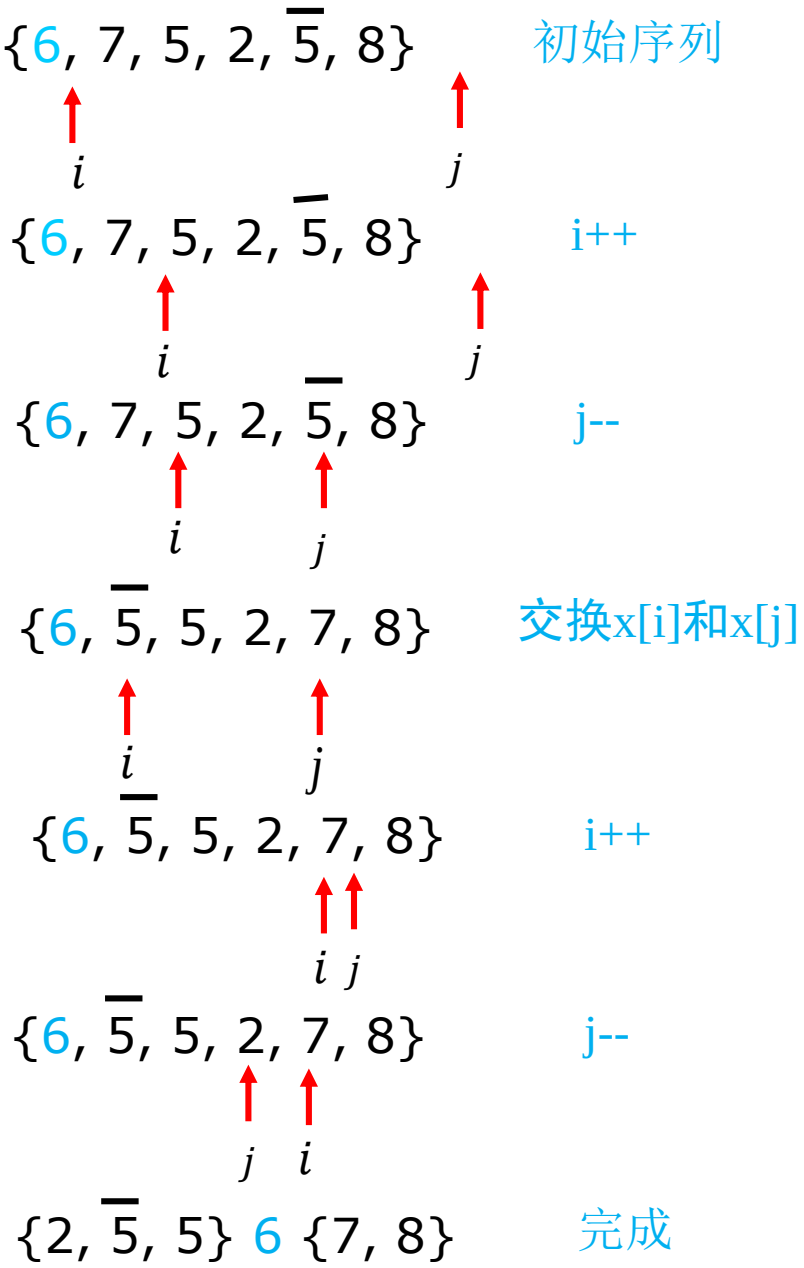
快速排序是对气泡排序的一种改进方法，它是由C.A.R. Hoare于1962年提出的。

```
private static void qSort(int p, int r) {
      if (p<r)
          int q=partition(p, r)
          qSort (p, q-1); //对左半段排序
          qSort (q+1, r); //对右半段排序
}
```

**partition**(p,r)以a[p]为基准元素将a[p:r]划分成3段a[p:q-1],a[q]和a[q+1:r]，使得a[p:q-1]中任何元素小于等于a[q]，a[q+1:r]中任何元素大于等于a[q]。下标q在划分过程中确定。

# 快速排序

```
private static int partition (int p,int r)
{
    int i = p,  j = r + 1;
    Comparable 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[++i].compareTo(x) < 0);
        while (a[--j].compareTo(x) > 0);
        if (i >= j)  break;
        MyMath.swap(a, i, j);
    a[p] = a[j];
    a[j] = x;
    return j;
}
```




**快速排序具有不稳定性。**

# 快速排序

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。

```
private static int randomizedPartition (int p, int r)
    {
        int i = random(p,r);
        MyMath.swap(a, i, p);
        return partition (p, r);
    }
```

# 快速排序

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。

- 最坏时间复杂度：$O(n^2)$
- 平均时间复杂度：$O(n\log n)$
- 辅助空间：$O(n)$或$O(\log n)$
- 稳定性：不稳定

# 线性时间选择

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

```
private static Comparable randomizedSelect(int p,int r,int k)
   {
      if (p==r) return a[p];
      int i=randomizedPartition(p,r),
          j=i-p+1;
      if (k<=j) return randomizedSelect(p,i,k);
      else return randomizedSelect(i+1,r,k-j);
   }
```

在最坏情况下，算法**randomizedSelect**需要$O(n^2)$计算时间但可以证明，算法**randomizedSelect**可以在O(n)平均时间内找出n个输入元素中的第k小元素。

# 线性时间选择

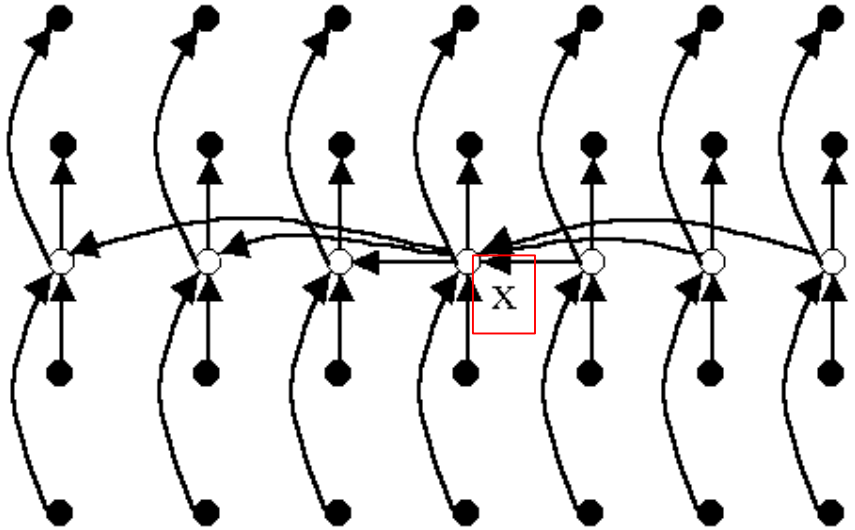
如果能在线性时间内找到一个划分基准，使得按这个基准所划分出的2个子数组的长度都至多为原数组长度的ε倍(0<ε<1是某个正常数)，那么就可以**在最坏情况下**用O(n)时间完成选择任务。

例如，若ε=9/10，算法递归调用所产生的子数组的长度至少缩短1/10。所以，在最坏情况下，算法所需的计算时间T(n)满足递归式T(n)≤T(9n/10)+O(n) 。由此可得T(n)=O(n)。

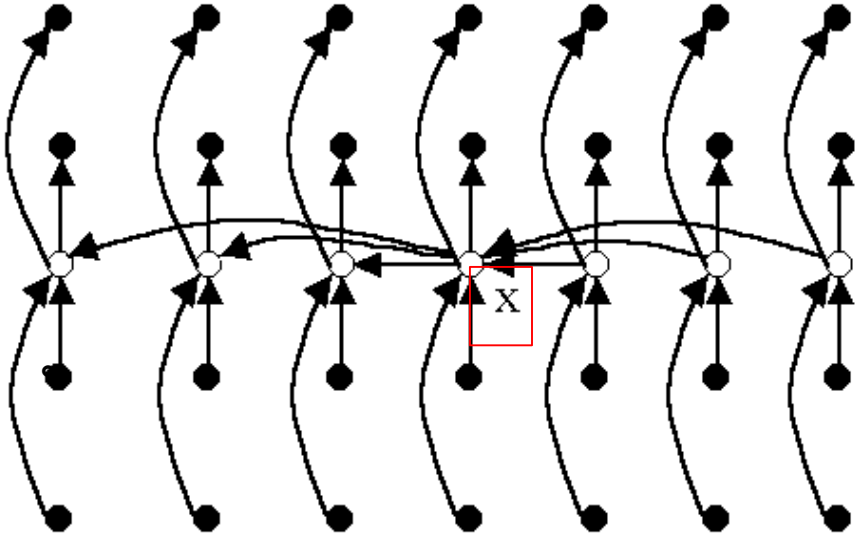
# 线性时间选择

可以按如下步骤找到满足要求的划分基准

- 将n个输入元素划分成⌈n/5⌉个组，每组5个元素，只可能有一个组不是5个元素。用任意一种排序算法，将每组中的元素排好序，并取出每组的中位数，共⌈n/5⌉个。
- 递归调用**select**来找出这⌈n/5⌉个元素的中位数。如果⌈n/5⌉是偶数，就找它的2个中位数中较大的一个。以这个元素作为划分基准。



# 线性时间选择




设所有元素互不相同，则基准x至少比3(n-5)/10个元素小。因为：

- x左侧有(n-5)/2个元素，每组有5个元素，故左侧共有(n-5)/10组；
- 左侧每组的中位数均小于x，且每一组中有2个元素小于本组的中位数，故左侧共有3(n-5)/10个元素小于x；

同理，基准x也至少比3(n-5)/10个元素大。而当n≥75时，3(n-5)/10≥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。

```
private static Comparable select (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;
                bubbleSort(p,r);
                return a[p+k-1];
        //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;
        //将所有中位数放到p到p+ (n-5)/5
        //r-p-4即上面所说的n-5
        for ( int i = 0; i<=(r-p-4)/5; i++ )
                int s=p+5*i,
                t=s+4;
                for (int j=0;j<3;j++)
                        bubble(s,t-j);
                MyMath.swap(a, p+i, s+2);
```

```
    //找中位数的中位数
    Comparable x = select(p, p+(r-p-4)/5, (r-p+6)/10);
    int i=partition(p,r,x); //以x为基准对a[p:r]进行划分
    int j=i-p+1;
    if (k<=j)
            return select(p,i,k);
    else
            return select(i+1,r,k-j);
}
```

# 线性时间选择

**复杂度分析**

- 找各组中位数的for循环执行n/5次，每次O(1)，共O(n)
- n个元素时间复杂度为T(n)，中位数n/5个，找中位数的中位数T(n/5)
- 子数组长度至多为原数组3/4，select至多用T(3/4n)

$$T(n) \le \begin{cases} C_1 & n < 75 \\ C_2 n + T(n/5) + T(3n/4) & n \ge 75 \end{cases}$$

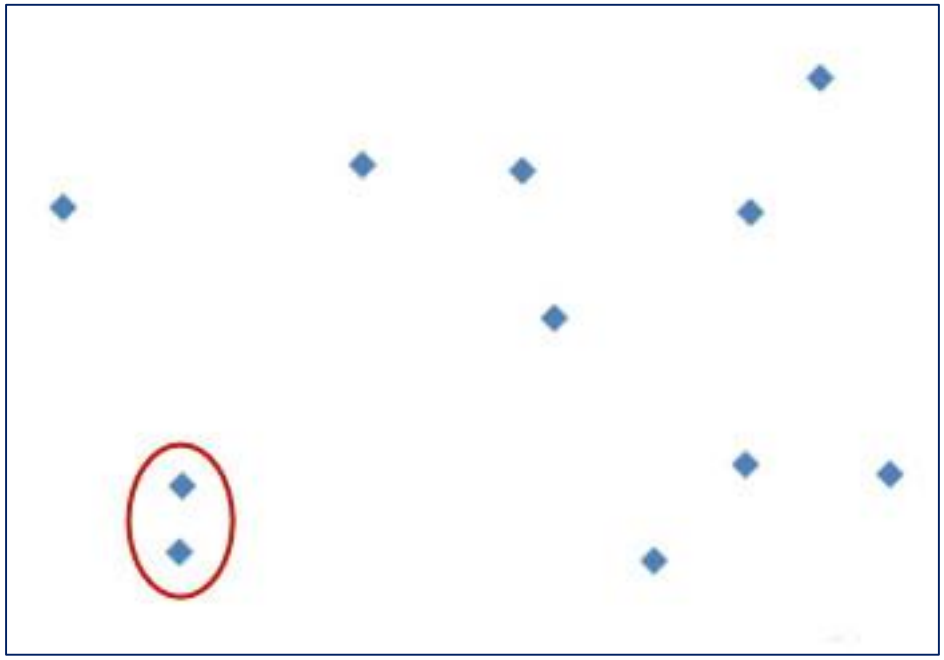
T(n)=**O(n)**

上述算法将每一组的大小定为5，并选取75作为是否作递归调用的分界点。这2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。

# 最接近点对问题

给定平面上n个点的集合S，找其中的一对点，使得在n个点组成的所有点对中，该点对间的距离最小。

若将飞机作为空间中移动的一个点，则具有最大碰撞危险的两架飞机就是这个空间中最接近的一对点。



# 最接近点对问题

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。

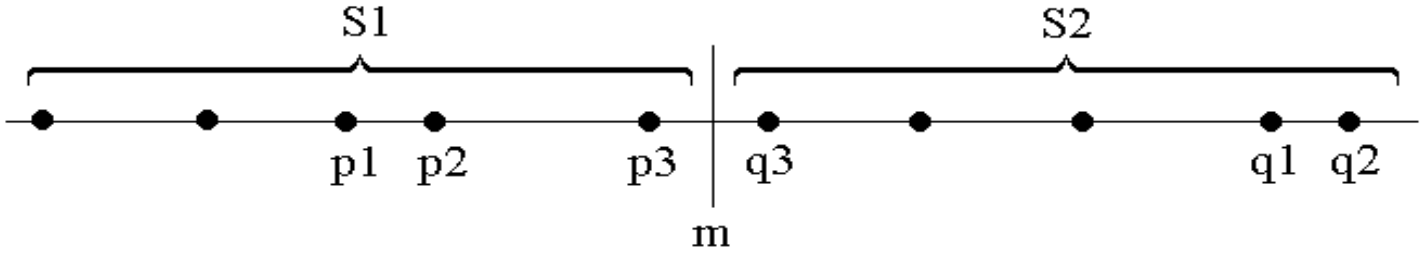
- 可以先将x1, x2, …, xn排好序，然后用一次扫描就可以找出最接近点对

主要时间花费在排序上，因此时间复杂度为O(nlogn)。

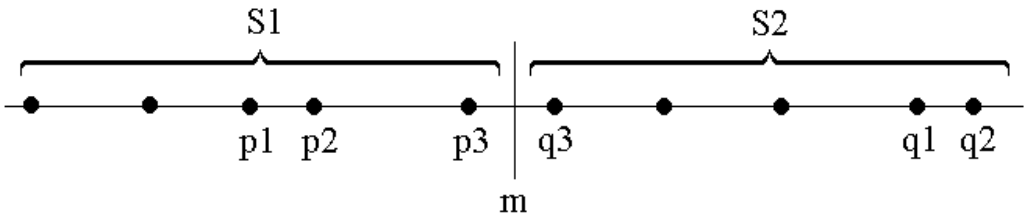
然而，该方法无法直接推广到二维的情形。

# 最接近点对问题

- 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中点坐标的中位数来作分割点。用选取中位数的线性时间算法可以在O(n)时间内确定一个平衡的分割点m;
- 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?**



# 最接近点对问题




**能否在线性时间内找到p3,q3?**

- 如果S的最接近点对是{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]。**
- 由于在S1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S1和S2的分割点，因此(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S1中最大点。**
- 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p3和q3。**从而我们用线性时间就可以将S1的解和S2的解合并成为S的解。**

# 最接近点对问题

该方法的分割步骤和合并步骤总共耗时O(n)。因此，算法耗费的时间T(n)满足递归方程：

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

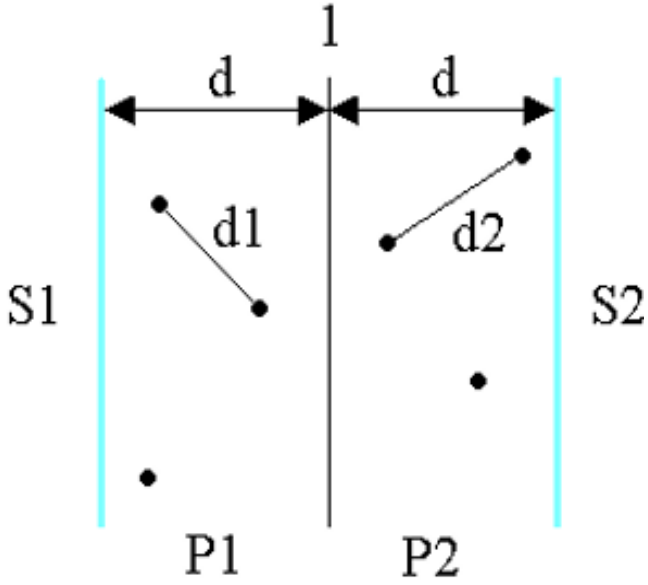
解得T(n)=O(nlogn)。

该方法比排序方法复杂，但可以推广到二维的情形。

# 最接近点对问题

- 下面来考虑二维的情形。

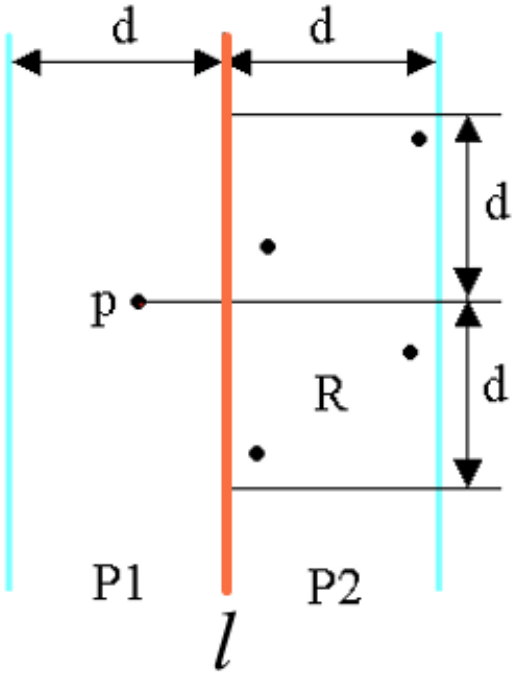
- 选取一垂直线 ***l*:x=m**来作为分割直线。其中m为S中各点x坐标的中位数。由此将S分割为S1和S2。
- 递归地在S1和S2上找出其最小距离d1和d2，并设 **d=min{d1,d2}**，S中的最接近点对或者是d，或者是某个{p,q}，其中p∈P1且q∈P2，P1和P2分别表示直线*l*的左侧和右侧宽为d的两个垂直长条区域。



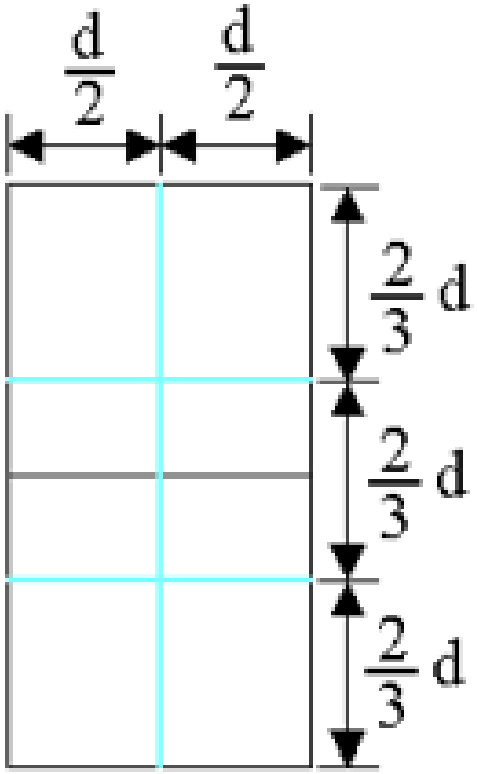

- **能否在线性时间内找到p,q?**

# 最接近点对问题

- 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。**满足这个条件的P2中的点一定落在一个d×2d的矩形R中。**
- 由d的意义可知，P2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×n/2=3n个候选者。**



# 最接近点对问题




**证明:** 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)×(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)×(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

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

distance(u,v)<d。这与d的意义相矛盾。

# 最接近点对问题

- 为了确切地知道要检查哪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者。对P1中每一点最多只要检查P2中排好序的相继6个点。

# 最接近点对问题

```
public static double cpair2(S)
{
    n=
    i
1.  m=S中
    //构
    S1
    S2=
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
   序;
   点列;
   检查Y中
   多6个)
   时，Y
   间内
   * 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;
}

**复杂度分析**

第2步用了2T(n/2)时间

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

**T(n)=O(nlogn)**