递归与分治策略

学习要点

- 理解递归的概念
- 掌握设计有效算法的分治策略
- 通过下面的范例学习分治策略设计技巧
  1) 二分搜索技术；
  2) 大整数乘法；
  3) Strassen矩阵乘法；
  4) 合并排序和快速排序；
  5) 线性时间选择；
  6) 最接近点对问题；

哈尔滨工程大学

递归的概念

递归

❑ 递归的概念

- 直接或间接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

- 算法是面向问题的，是解决问题的方法或过程。递归算法中，多次使用同样的方法，解决同样的问题，只不过输入规模不同。

递归函数（阶乘函数）

- 阶乘函数

边界条件

```
int fractorial (int n)
    {
        If(n==0)   return 1;
        return n*fractorial(n-1);
    }
```

非递归定义的初始值

较小自变量的函数值表示较大自变量的函数值

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。

递归函数（斐波那契数列）

- Fibonacci数列
  - 无穷数列1，1，2，3，5，8，13，21，34，55，……
  - 递归定义为

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

需要两个非递归定义的初始值

用两个较小的自变量定义一个较大自变量的函数值

整数划分问题

- 定义
  - 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$n_1 \geqslant n_2 \geqslant \ldots \geqslant n_k \geqslant 1$，$k \geqslant 1$。
  - 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

- 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

(1) q(n, 1)=1, n≥1;　*m=1*
最大加数n_1不大于1，任何正整数n只有一种划分形式，即 $n=\overbrace{1+1+\cdots+1}^{n}$

(2) q(n, m)=q(n, n),　*m>n;*
最大加数n_1实际上不能大于n。因此，q(1, m)=1

(3) q(n, n)=1+q(n, n−1); *m=n*
正整数n的划分由n_1=n的划分(1个)和n_1≤n−1的划分组成。

(4) q(n, m)=q(n−m, m)+ q(n, m−1), *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 (n_1<m)的划分组成。

m	n−m

整数划分问题

- 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

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

正整数n的划分数p(n)=q(n, n)

整数划分问题

❐ 例子

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

$$q(6,3) = q(6,2) + q(3,3)$$

$$= \underline{q(6,1) + q(4,2)} + \underline{1 + q(3,2)}$$

$$= q(6,1) + \underline{q(4,1) + q(2,2)} + 1 + \underline{q(3,1) + q(1,2)}$$

$$= q(6,1) + q(4,1) + \underline{1 + q(2,1)} + 1 + q(3,1) + q(1,2)$$

$$= 1 + 1 + 1 + 1 + 1 + 1 + 1 = 7$$

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

- 程序

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

```
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 1+q(n,n-1)
    return q(n,m-1)+1(n-m,m)
}
```

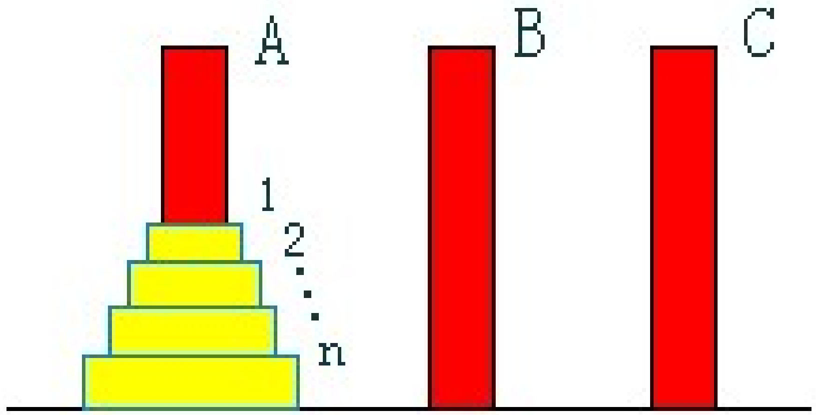
Hanoi塔问题

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；

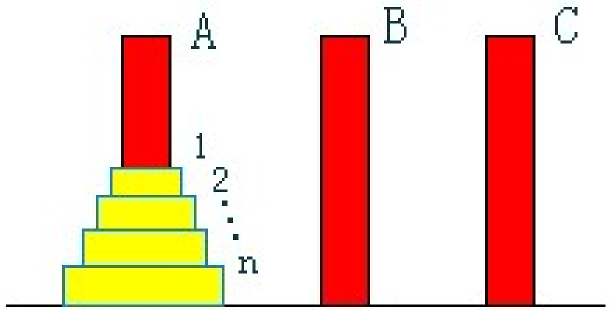
规则3：在满足移动规则1和2的前提下，可将圆盘移至a, b, c中任一塔座上。



Hanoi塔问题

- 印度教的主神梵天创造世界时，做了三根金刚石的柱子，并在其中的一根柱子上按照大小顺序依次放置了64个黄金圆盘。
- 梵天神告诉侍奉他的婆罗门（祭司），要借助一根柱子做中介，来把这64个圆盘一起移动到另一根柱子上；规则和上面说的一样
- 梵天大神说了，只要你们能实现最终的目标，世界就会在一个闪电中毁灭。
- 据说，这个婆罗门和他的后人从此就开始一刻不停的挪圆盘，以愚公移山的精神，为世界的最终毁灭贡献自己的力量。
- 让他们先挪一会！

Hanoi塔问题



- 简单解法

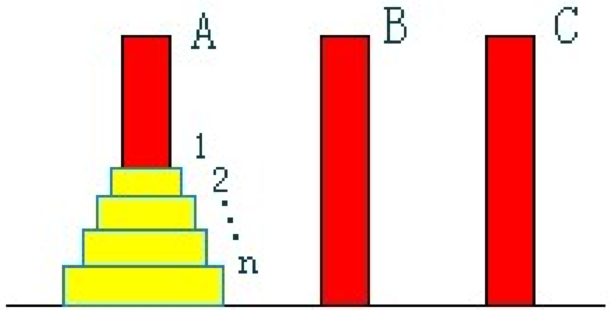
（1）将ABC排成一个三角形，A-B-C-A构成顺时针循环；

（2）在移动过程中，奇数次移动，则将最小的圆盘移到顺时针下一塔座；

（3） 若偶数次移动，则保持最小的圆盘不动，而在其他两个塔座之间，将较小的圆盘移动到另一塔座。

方法可以证明正确，但很难明白其中道理。

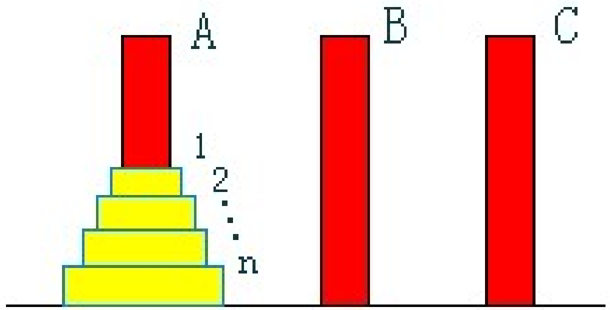
Hanoi塔问题



- 要将n个圆盘按规则从A移动到B，只需
  - 将n－1个较小圆盘按规则从A移动到C
  - 将最大的圆盘从A移动到B
  - 将n-1个较小圆盘从C移动到B
- n个圆盘的移动问题可以转换为
  - 两次n-1个圆盘移动的问题＋一个单一圆盘移动
  - 假设h（n）为n个圆盘的移动次数，那么

$$h(n) = 2 * h(n-1) + 1$$

Hanoi塔问题




```
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);
     }
  }
```

hanoi(n, A, B, C)表示将n个圆盘按规则从A移动到B，移动的过程中用C作为辅助塔座

move(A, B)表示将A上剩余的单一圆盘从A移动到B

Hanoi塔问题

h(n) =2* h(n-1)+1 的时间复杂度计算？

推导：$f(1) = 1 = 2^1-1$

$f(2) = 2*f(1) + 1 = 3 = 2^2-1$

$f(3) = 2*f(2) + 1 = 7 = 2^3-1$

…

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

指数增长到底有多快？

对于“梵天事件”，每个圆盘移动要一秒，要多长时间完成？

$2^{64}-1$ = 18446744073709551615秒

一个平年365天有31536000 秒，闰年366天有31622400秒，平均每年31556952秒，计算一下：

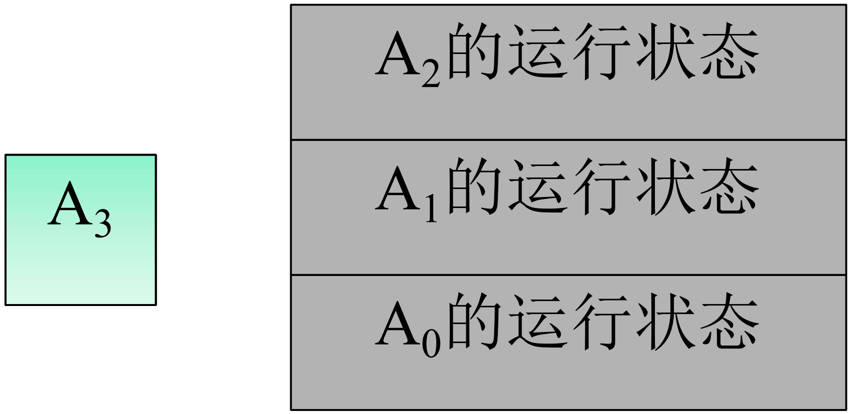
共计5845.54亿年！

函数调用的处理过程

- 函数A调用函数B时
  - 保存A的所有运行状态
  - 将实参指针、返回地址等信息传递给B
  - 为B中的变量分配存储区
  - 将控制转移到B的入口
- 从函数B返回到函数A时
  - 保存B的计算结果
  - 释放分配给B的存储区
  - 依照返回地址将控制转移到A

递归的处理过程

- 函数A调用自身
  - 分层：A_0, A_1, A_2, A_3,……
  - A_0调用A_1
  - A_1调用A_2
  - A_2调用A_3
  - A_3返回给A_2
  - A_2返回给A_1
  - A_1返回给A_0



递归的特点

- 优点
  - 结构清晰可读性强
  - 容易用数学归纳法来证明算法的正确性
- 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

消除递归

- 采用一个用户定义的栈来模拟系统的递归调用工作栈
  - 机械地模拟与递归算法效果相同，但仅仅如此没有优化
  - 根据程序特点对递归调用的工作栈进行简化，减少栈操作
- 尾递归消除
  - 递归调用只有一个，并且是放在最后，如n!
  - 对栈空间优化，当前栈只需要存储（n-1）！ ,反复利用
- 迭代法
  - 采用循环结构（相比递归的选择结构）
  - 结构复杂，效率高

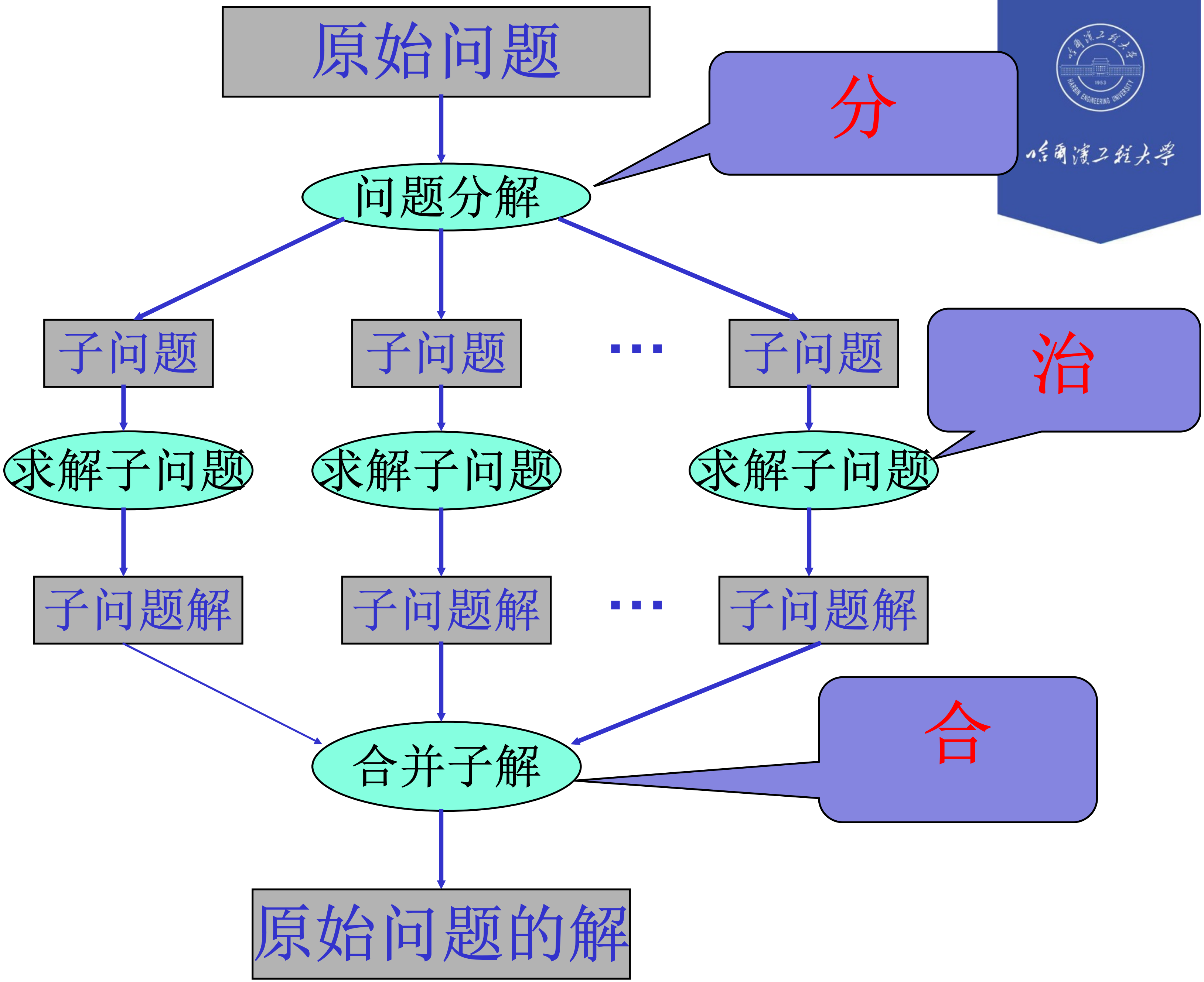
消除递归

```c
#include <stdio.h>
int main()
{
    int i,n;
    double sum=1;
    scanf("%d",&n);
    for(i=1;i<=n;i++)
        sum=sum*i;
    printf("%d!=%d",n,sum);
    printf("\n");
    return 0;
}
```

分治策略

分治策略

- 分
  - 将要求解的较大规模的问题分割成k个更小规模的子问题。
  - 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。
- 治
  - 求解各个子问题
- 合
  - 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。


原始问题
分
问题分解
子问题
子问题
…
子问题
治
求解子问题
求解子问题
求解子问题
子问题解
子问题解
…
子问题解
合
合并子解
原始问题的解

分治策略

□ 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；

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。

分治策略

□ 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：

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，此特征反映了递归思想的应用

分治策略

□ 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：

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；

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**。

分治策略

□ 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：

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。

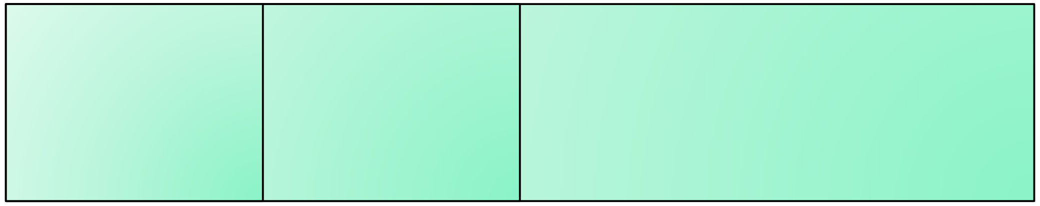
这条特征涉及到分治法的效率，如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然也可用分治法，但一般用**动态规划**较好。

分治策略

- 如何分？
  - 应该把原问题划分为多少个子问题？
  - 每个子问题的规模是否相同？
  - 从大量实践中发现，最好使子问题的规模大致相同
  - 许多问题可以取k=2，基于平衡子问题的思想，几乎总是比子问题规模不等要好。

合并排序算法

- 输入：a[1, …, n]
- 输出：a[1, …, n] 满足 a[0]≤ a[1] ≤… ≤ a[n]
- 基本思想:
  - 将待排序元素分为大小相等的两个子集合
  - 分别对两个子集合排序
  - 将排好序的子集合合并为最终结果



合并排序算法

```
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
      }
  }
```

Mergesort 进行分裂直到只有一个元素。

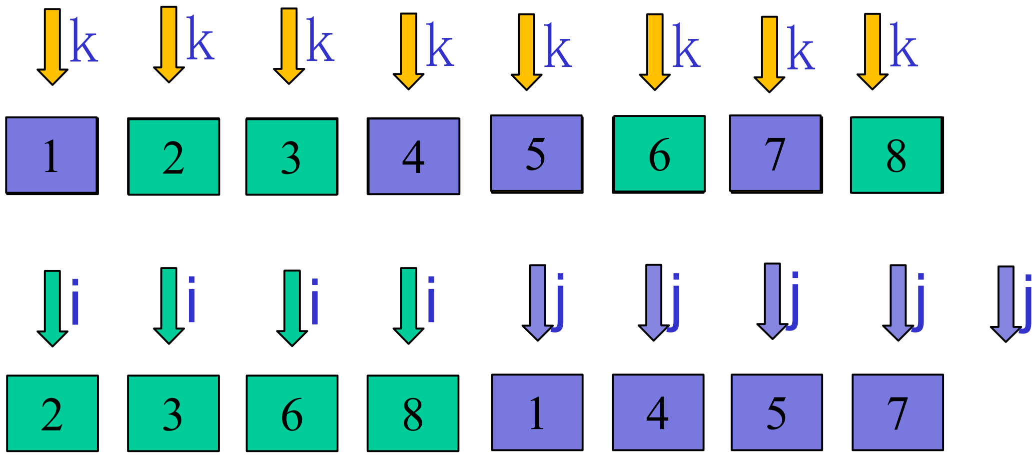
调用merge函数合并两个有序数组（只有一个元素也是有序）

Merge(a, b, left, i, right): 将排好序的a[left, …, i]和a[i+1, …, right]按顺序合并到数组b中，然后再copy回数组a。

合并排序算法

❐ 合并过程 （Merge）

- 对两个*已经排序好的*数组(n=8)，如何将他归并成一个数组
- 需要额外o（n）空间建立临时数组
- 三个索引, i、j、k来追踪数组位置



合并排序算法

```
void Merge(Tpye c[], Tpye d[], int l, int m, int r)
{
    //合并c[l:m]和c[m+1:r]到d[l:r]
    int i = l, j=m+1, k = l;
    while(i<=m && j<=r)
    {
        if(c[i] > c[j])
            d[k++] = c[i++];
        else
            d[k++] = c[j++];
    }
    if(i >= m)
       for(q=j; q<=r; q++)
        d[k++] = c[q];
    else
       for(q=i; q<=m; q++)
        d[k++] = c[q];
}
```

合并排序算法

消除递归

- 首先将a中相邻的元素两两配对
- 用合并算法将它们排序，构成n/2个长度为2的排好序的数组
- 再用合并算法将它们排序成n/4个长度4的排好序数组…

```
void MergeSort(Type a[], int n){
    int i=1;
    while(i<n){
    对a进行一边扫描，将a中大小为i的相邻数组合并(Merge);
    }
}
```

合并排序算法

□ 运行例子

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]

合并排序算法

- 时间复杂性$T(n)$
  - Merge和Copy可以在O(n)时间内完成

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

$$T(n)=O(n\log n)$$

分治策略的时间复杂性

- 假设分治方法将原问题分解为k个规模为n/m的子问题来求解，则

$$T(n) = kT(n/m) + f(n)$$

  - 其中，$f(n)$为将原问题分解为k个子问题及将k个子问题的解合并为原问题的解所需要的时间

1.5章 递归方程的渐进性

二分搜索算法

❑ 解决查找元素问题

- 输入：已经按升序排好序的数组a[0,…,n-1]和某个元素x
- 输出：x在a中的位置

问题特点：

- ✓ 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。

二分搜索算法

- 分治法解决方案

比较x和a的中间元素a[mid]

1. 若x=a[mid]，则x在L中的位置就是mid；
2. 如果x<a[mid]，由于a是递增排序的，所以我们只要在a[mid]的前面查找x即可；
3. 如果x>a[i]，同理我们只要在a[mid]的后面查找x即可。
4. 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。

二分搜索算法

据此容易设计出二分搜索算法：

```
int BinarySearch(Type a[], const Type& x, int n)
{
    int l = 0; int r = n-1;
    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;// 未找到x
}
```

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)** 。

二分搜索算法

递归写法：

```
int BSearch(Type a[], const Type& x,int low,int high)
{
    int mid;
    if(low>high) return -1;
    mid=(low+high)/2;
    if(x==a[mid]) return mid;
    if(x<a[mid]) return(BSearch(a,x,low,mid-1));
    else return(BSearch(a,x,mid+1,high));
}
```

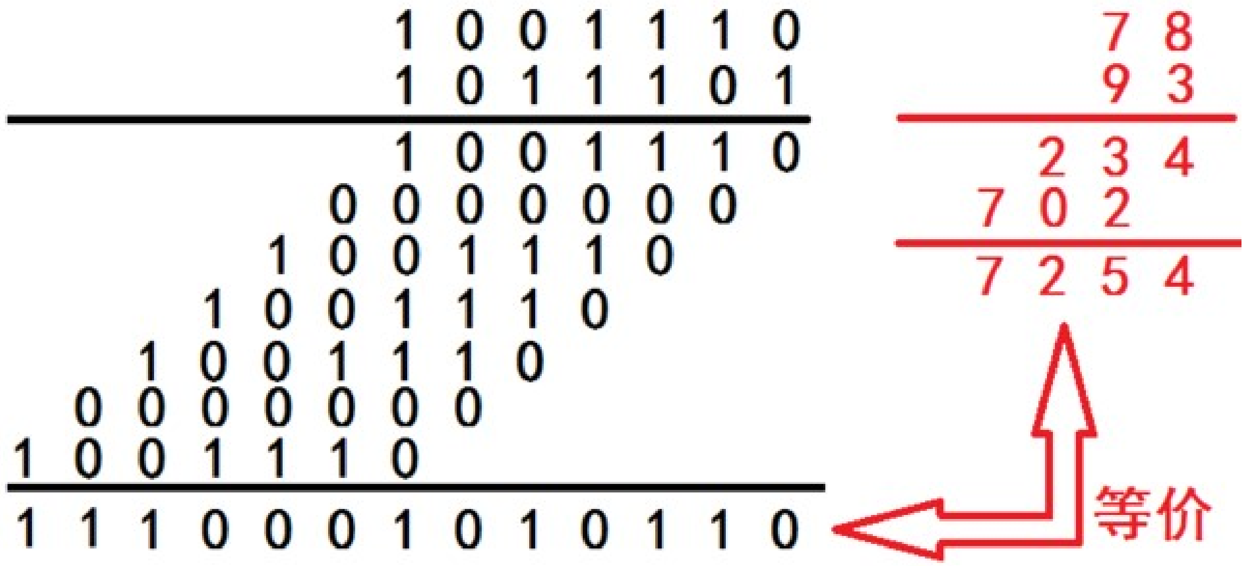
So easy?????

1.n很大怎么办？
2.数组中有很多相同大小的元素怎么办？

大整数乘法

计算两个n位整数的乘积

- 小学生方法：$O(n^2)$　　✘效率太低



大整数乘法

计算两个n位整数的乘积

◆分治法1:

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

$$X = a\,2^{n/2} + b \qquad Y = c\,2^{n/2} + d$$

$$XY = ac\,2^{n} + (ad+bc)\,2^{n/2} + bd$$

四次n/2位乘法，3次不超过2n位的加法及2次移位操作

大整数乘法

为了降低时间复杂度，必须减少乘法的次数。

- 分治法2

$$XY = ac\ 2^n + (ad+bc)\ 2^{n/2} + bd$$

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

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

3次2/n乘法，6次不超过2n位的加、减法及2次移位操作

大整数乘法

- 小学生方法：$O(n^2)$ ✘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。
- **1971年，n*logn*loglogn**
- 。。。
- **2020年，n*logn**

Strassen矩阵乘法

❑ 问题

- 输入：n×n的矩阵A和B
- 输出：C=AB

❑ 简单方法

$$c_{ij}=\sum_{k=1}^{n}a_{ik}b_{kj}$$,A的i行和B的j列的第k个元素

- 复杂性$O(n^3)$

Strassen矩阵乘法

❑ 简单分治

- 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:

复杂度分析

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

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

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

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

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

Strassen矩阵乘法

为了降低时间复杂度，必须减少乘法的次数。

- Strassen分治

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

$M_1 =$

$M_2 =$

$M_3 =$

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

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

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

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

$C_{21} = M_3 + M_4$

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

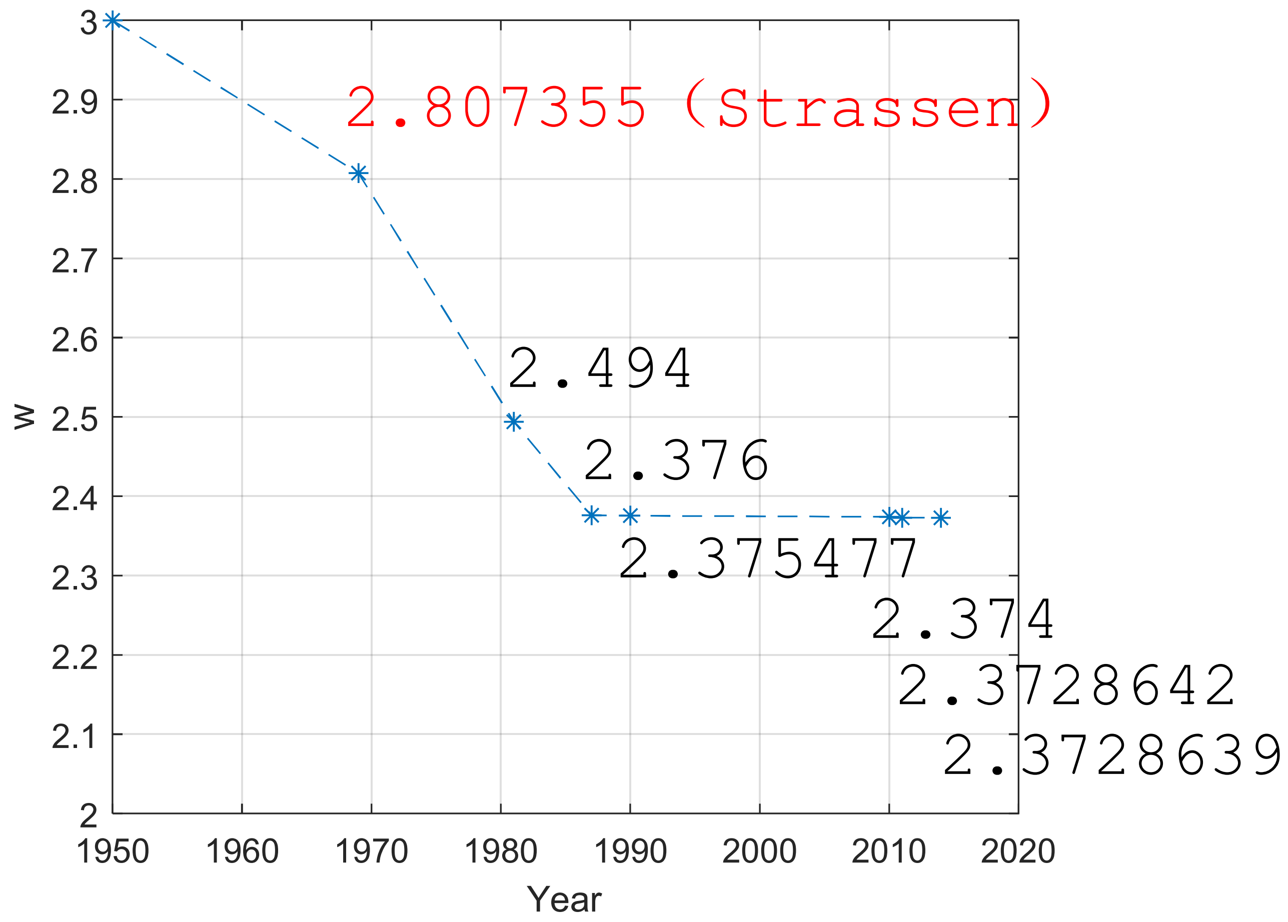
Strassen矩阵乘法

- 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。

Strassen矩阵乘法

- 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 **$O(n^{2.372})$**
- 是否能找到$O(n^2)$的算法?



快速排序

- 对数组a[p : r]进行排序
  - 分：以**a[p]=x**为基准将a[p : r]划分为3段
    - a[p : q-1]，a[q]，a[q+1 : r]
    - a[q]=x
    - a[p : q-1]中的元素小于等于a[q]
    - a[q+1 : r]中的元素大于等于a[q]
    - 找到基准数据的正确索引位置的过程。
  - 治：递归调用快速排序算法对a[p : q-1]和a[q+1 : r]排序
  - 合：递归调用过程中，就地排序，对于任意小的划分都已经排好序

快速排序

```
void QuickSort (Type a[], int p, int r)
{
    if (p<r) {
      int q = Partition(a,p,r);//将a[p : r]分成三部分
      QuickSort (a,p,q-1); //对左半段排序
      QuickSort (a,q+1,r); //对右半段排序
      }
}
```

关键：**q**是位置，**a[p]**是基准；**a[q]=a[p]**

对具有**n**个元素的数组**a[0:n-1]**进行排序只需要调用**QuickSort (a, 0, n-1)**

快速排序

int **Partition** (Type a[], int p, int r)（非常重要）

```
{
    int i = p, j = r + 1;
    Type x=a[p];//基准
    // 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] <x);//a[i] 左边都要小于x
      while (a[- -j] >x);//a[j] 右边都要大于x
      if (i >= j) break;
      Swap(a[i], a[j]);
      //当a[i] >=x， a[j] <=x 时，交换基准
    }
    a[p] = a[j];
    a[j] = x;
    return j;
}
```

{6, 7, 5, 2, 5, 8} 初始序列

{6, 7, 5, 2, 5, 8} ++i - -j

i j

{6, 7, 5, 2, 5, 8} - -j

i j

{6, 5, 5, 2, 7, 8} Swap()

i j

{6, 5, 5, 2, 7, 8} ++i - -j

j i

{2, 5, 5} 6 {7, 8} Swap()

快速排序

- 时间复杂性与划分是否对称有关
- 最坏情况
  - 划分产生的两个区域分别包含1个元素和n-1个元素
  - 每次递归都出现这种不对称划分
  - Partition计算时间为$O(n)$

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

$$T_{\max}(n) = O(n^2)$$

快速排序

最好情况

- 每次划分都产生两个大小为n/2的区域

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

$$T(n)=O(n\log n)$$

平均情况

$$T_{avg}(n)=O(n\log n)$$

- 可以证明，但相当复杂。

快速排序

❒ 改进

- 修改Partition函数，从a[p : r]中随机选择一个元素最为划分基准，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。

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

```
void RandomQuickSort (Type a[], int p, int r)
{
      if (p<r) {
        int q = RandomizedPartition(a,p,r);
        RandomQuickSort(a,p,q-1);
        RandomQuickSort(a,q+1,r);
        }
}
```

- 时间复杂性没有变化

线性时间选择

问题

- 输入：数组$a[n]$，正整数$1\leq k \leq n$
- 输出：$a[n]$中第k小的元素
  - k=1取最小元素；k=n取最大元素；k=(n+1)/2取中位数

排序法

- 先用合并排序算法对$a[n]$排序
- 取$a[k]$
- 复杂性：$O(n\log n)$

线性时间选择

- 找n个元素中的最大或最小元素
  - 时间复杂度：O(n)
- k<= n/logn 或 k>= n-n/logn,堆排序可以实现：
  - 时间复杂度：O(n)
- 分治方法：随机选择法
  - 模仿快速排序
  - 只对划分出的数组之一递归求解
  - 重点是**Partition**，以哪个元素为基准**Partition**

线性时间选择

❑ 随机选择法

- 从a[p : r]中<u>随机选择一个元素</u>将其进行划分为
  - a[p : q-1], a[q], a[q+1 : r]
  - a[p : q-1]中的元素小于等于a[q]
  - a[q+1 : r]中的元素大于等于a[q]
  - a[p : q]中元素的个数为m=q-p+1
- If (k = m)
  - 返回a[q],第m小的元素
- If (k < m)
  - 用随机选择法选取数组a[p : q-1]中第k小的元素
- If (k > m)
  - 用随机选择法选取a[q+1 : r]中第k-m小的元素

线性时间选择

```
RandomizedSelect (Type a[],int p,int r,int k)
{
    if (p==r) return a[p];
    int q = RandomizedPartition(a,p,r);
    m=q-p+1;
    if (k==m) return a[q];
    else if (k<m) return RandomizedSelect(a, p, q-1, k)
    else return RandomizedSelect(a, q+1, r, k-m);
```

- 在最坏情况下，比如找最小的元素总是在最大的元素处划分，算法**RandomizedSelect**需要**$O(n^2)$**计算时间
- 但可以证明，算法**RandomizedSelect**由于划分基准随机，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

```
RandomizedSelect (Type a[],int p,int r,int k)
{
    if (p==r) return a[p];
    int q = RandomizedPartition(a,p,r);
    if (k==m) return a[q];
    else if (k<m) return RandomizedSelect(a,p,q-1,k);
    else return RandomizedSelect(a,q+1,r,k-m);
```

如何在最坏情况下，算法复杂度达到O(n)

- 在最坏情况下，比如找最小的元素总是在最大的元素处划分，算法**RandomizedSelect**需要**O(n²)**计算时间
- 但可以证明，算法**RandomizedSelect**由于划分基准随机，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

如果能保证划分后得到的 2 个子数组*都至少*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。

- ✓如果，ε= 1/10 ，算法递归调用所产生的子数组的长度至多为原来的9/10。
- ✓那么，在最坏情况下，算法所需的计算时间T(n)满足递归式$T(n) \le T(9n/10)+O(n)$ 。
- ✓由此可得，**$T(n)=O(n)$**。

线性时间选择

- 改进的选择算法（取数组 a 中第 k 小的元素）
  - 将数组a划分为 n /5个组，每组 5 个元素。将每组的 5 个元素排好序，取出每组的中位数，共n/5个
  - 递归调用改进的选择算法取这n/5个元素的中位数 x
  - 用x来划分数组a得到（前后分别小于和大于基准）
    - a[p : q-1], a[q], a[q+1 : r]
  - If (k = m)
    - 返回a[q]
  - If (k < m)
    - 用选择算法选取数组a[p : q-1]中第k小的元素
  - If (k > m)
    - 用选择算法选取a[q+1 : r]中第k-m小的元素

线性时间选择

➢实例：找出中位数（改进的选择算法）

8，33，17，51，57，49，35，11，25，37，14，3，2，13，52，12，6，29，32，54，5，16，22，23，7

✓ **A[1..25]**

✓ 中位数$k=\lceil 25/2 \rceil=13$

线性时间选择

- 实例：找出中位数（改进的选择算法）

8，33，17，51，57，49，35，11，25，37，14，3，2，13，52，12，6，29，32，54，5，16，22，23，7

- ✓ A[1..25]
- ✓ 中位数 $k=\lceil 25/2 \rceil=13$

线性时间选择

哈尔滨工程大学

- 实例：找出中位数（改进的选择算法）

8，33，17，51，57，<u>49，35，11，25，37</u>，14，3，2，13，52，12，6，29，32，54，5，16，22，23，7

- ✓ A[1..25]
- ✓ 中位数k=⌈25/2⌉=13

线性时间选择

- 实例：找出中位数（改进的选择算法）

8，33，17，51，57，49，35，11，25，37，14，3，2，
13，52，12，6，29，32，54，5，16，22，23，7

- ✓ **A[1..25]**
- ✓ 中位数$k=\lceil 25/2\rceil=13$

线性时间选择

- 实例：找出中位数（改进的选择算法）

8，33，17，51，57，<u>49，35，11，25，37</u>，14，3，2，
13，52，<u>12，6，29，32，54</u>，5，16，22，23，7

- A[1..25]
- 中位数k=$\lceil 25/2 \rceil$=13

线性时间选择

➢实例：找出中位数（改进的选择算法）

8，33，17，51，57，49，35，11，25，37，14，3，2，13，52，12，6，29，32，54，5，16，22，23，7

- ✓ A[1..25]
- ✓ 中位数k=⌈25/2⌉=13

线性时间选择

➢实例：找出中位数（改进的选择算法, **k=13**）

8，	33，	17，	51，	57，
49，	35，	11，	25，	37，
14，	3，	2，	13，	52，
12，	6，	29，	32，	54，
5，	16，	22，	23，	7

8，	17，	33，	51，	57，
11，	25，	35，	37，	49，
2，	3，	13，	14，	52，
6，	12，	29，	32，	54，
5 ，	7 ，	16，	22，	23

线性时间选择

➢实例：找出中位数（改进的选择算法, **k=13**）

8，	33，	17，	51，	57，
49，	35，	11，	25，	37，
14，	3，	2，	13，	52，
12，	6，	29，	32，	54，
5，	16，	22，	23，	7

8，	17，	33，	51，	57，
11，	25，	35，	37，	49，
2，	3，	13，	14，	52，
6，	12，	29，	32，	54，
5，	7，	16，	22，	23

13, 16, 29, 33, 35

线性时间选择

- 实例：找出中位数（改进的选择算法, **k=13**）

8，	33，	17，	51，	57，
49，	35，	11，	25，	37，
14，	3，	2，	13，	52，
12，	6，	29，	32，	54，
5，	16，	22，	23，	7

⇒

8，	17，	33，	51，	57，
11，	25，	35，	37，	49，
2，	3，	13，	14，	52，
6，	12，	29，	32，	54，
5 ，	7 ，	16，	22，	23

13, 16, 29, 33, 35

线性时间选择

➢实例：找出中位数（改进的选择算法, **k=13**）

5，16，22，23，7，8，17，11，25，14，3，2，13，

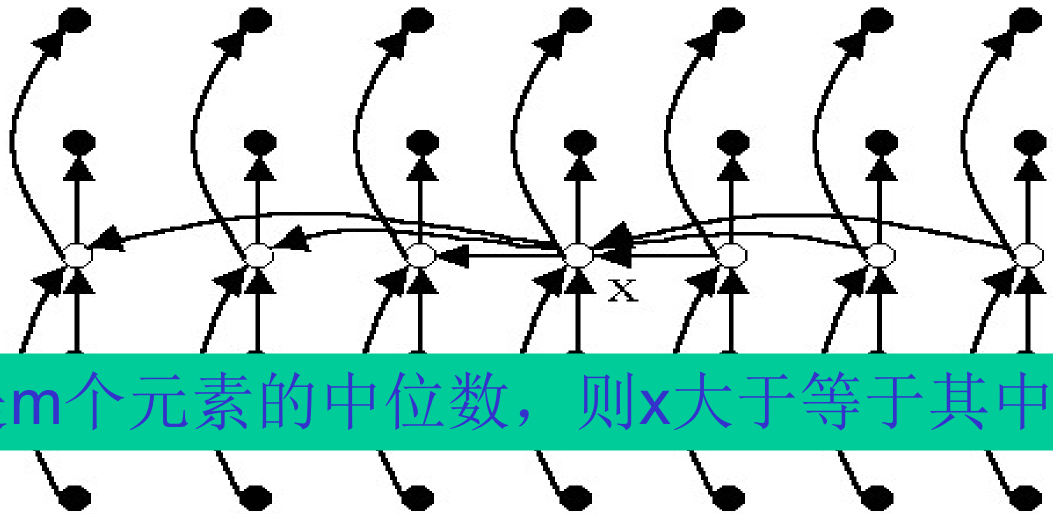
12，6，29，52，37，51，57，49，35，33，32，54

✓ 以**29**作为划分点，重新划分数组

✓与 k 值做比较

13, 16, 29, 33, 35

线性时间选择



如果x是m个元素的中位数，则x大于等于其中的$\left\lfloor \frac{m-1}{2} \right\rfloor$个元素

中位数小于**x**的组至少有 $\left\lfloor \frac{\lceil n/5 \rceil - 1}{2} \right\rfloor = \left\lfloor \frac{n-5}{10} \right\rfloor$ 个

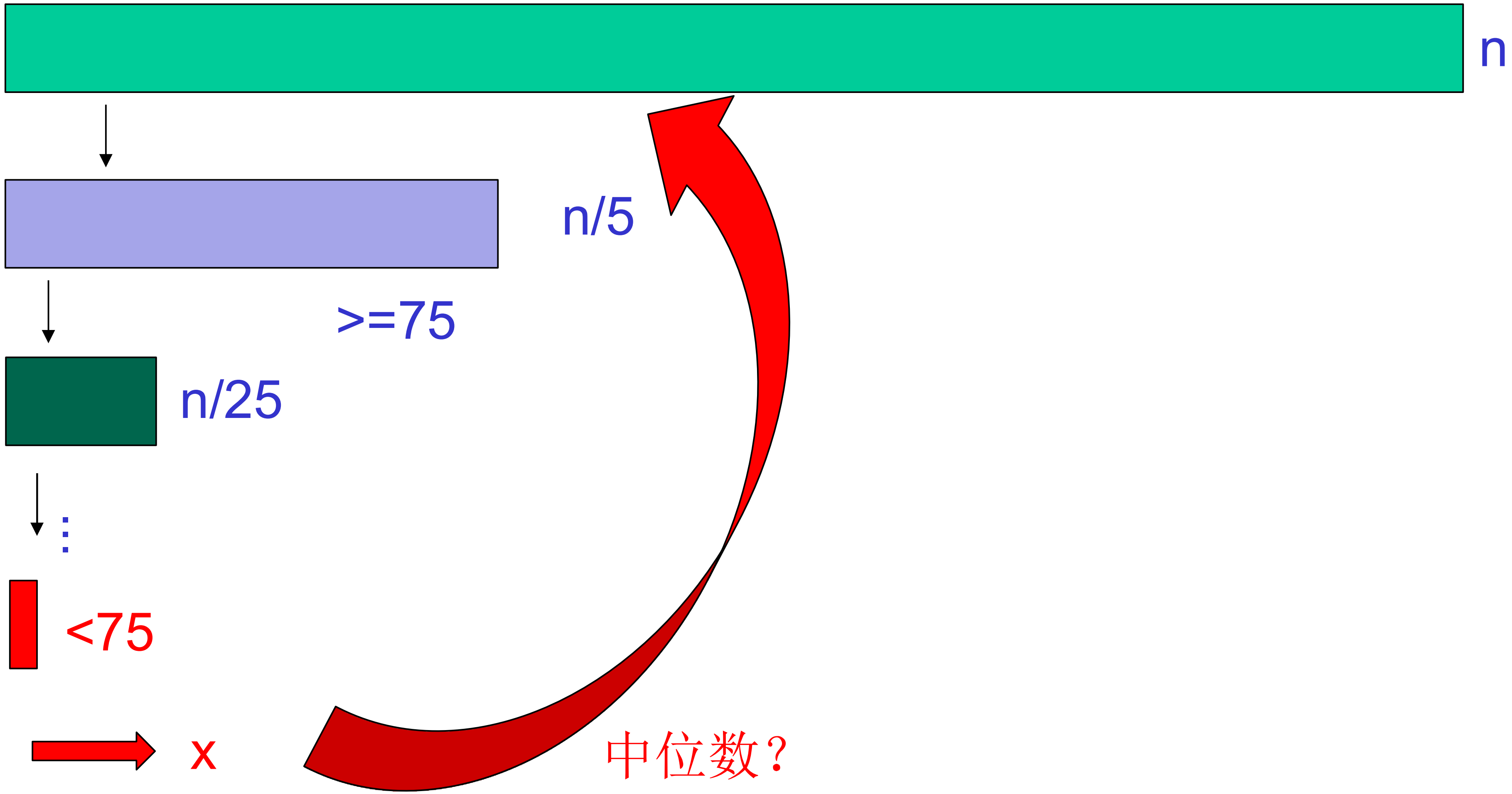
这些组中每组至少有**3**个元素小于**x**

$\therefore$至少有 $3\left\lfloor \frac{n-5}{10} \right\rfloor$ 个元素小于**x**

同理，至少有 $3\left\lfloor \frac{n-5}{10} \right\rfloor$ 大于**x**

当**n≥75**时，$3\left\lfloor \frac{n-5}{10} \right\rfloor \geq \frac{n}{4}$

```
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++ )
     //分组排序后，将中位数找到，都放在a[p: p+(r-p-4)/5]
        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;
    //找中位数的中位数，r-p-4即上面所说的n-5
    Type x = Select(a, p, p+(r-p-4)/5, (r-p-4)/10);
    int q=Partition(a,p,r, x),
    m=q-p+1;
    if (k==m) return a[q];
    else if (k<m) return Select(a, p, q-1, k)
    else return Select(a, q+1, r, k-m);
}
```


n
n/5
>=75
n/25
<75
x
中位数？

线性时间选择

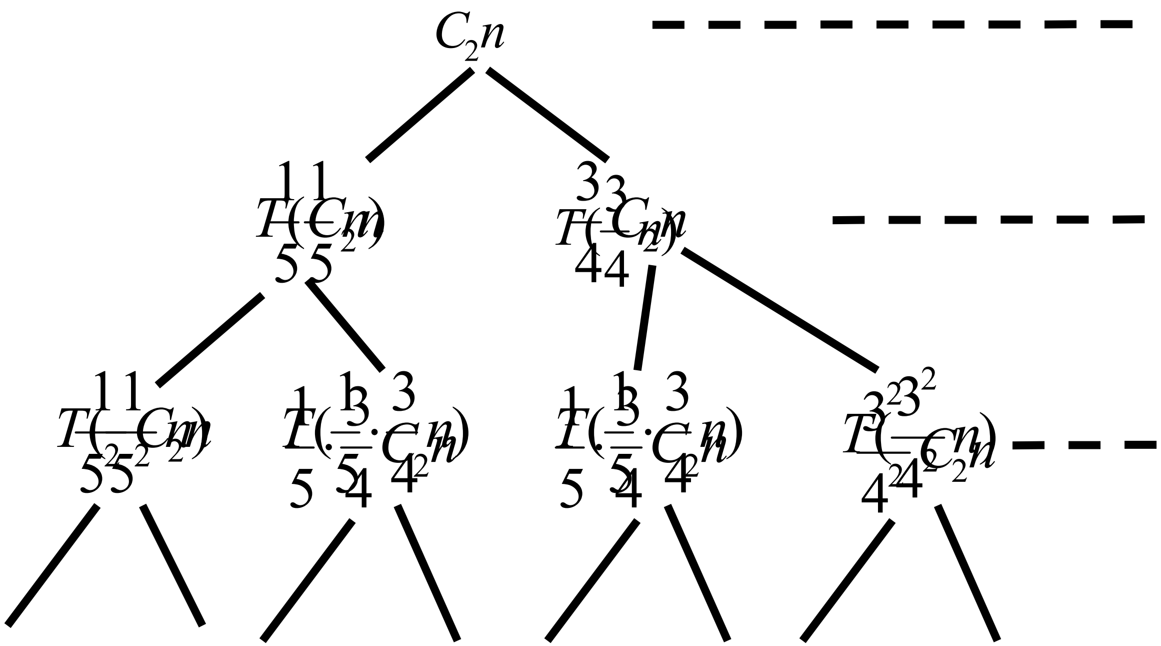
分别取中位数 | 中位数排序 | 递归调用

❑ 改进的选择算法（复杂度如何分析？）

✓n<75时，算法计算时间不超过常数C1
✓n>=75时，分三部分：
(1) 算法以中位数的中位数x对a[p:r]进行划分，需要O(n)时间。For循环共执行n/5次（i最大n/5），每次需要O(1)，共O(n)时间。
(2) 找到中位数的中位数共对n/5个元素进行递归调用，共至多T(n/5)
(3) 以x为基准划分的两个子数组分别至多包含3n/4，无论对哪个子数组进行递归调用，都至多T(3n/4)

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

$$T(n)=C_2 n+T(n/5)+T(3n/4)$$



$$C_2 n\left(\frac{1}{5}+\frac{3}{4}\right)^0$$

$$C_2 n\left(\frac{1}{5}+\frac{3}{4}\right)^1$$

$$C_2 n\left(\frac{1}{5}+\frac{3}{4}\right)^2$$

$$T(n)\le C_2 n\sum_{i=0}^{\infty}\left(\frac{1}{5}+\frac{3}{4}\right)^i=C_2 n\sum_{i=0}^{\infty}\left(\frac{19}{20}\right)^i=20C_2 n$$

线性时间选择

❑ 改进的选择算法

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

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

改进的选择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，找出距离最小的点对
- 算法应用
  - 常用于空中交通的计算机自动控制系统，也是计算机几何学研究的基本问题之一
  - 假设在一片金属上钻n 个大小一样的洞，如果洞太近，金属可能会断。若知道任意两个洞的最小距离，可估计金属断裂的概率。这种最小距离问题实际上也就是距离最近的点对问题。

最接近点对问题

- 简单暴力方法
  - 对任意点对，计算两点之间的距离
  - 找出距离最小的点对
  - $O(n^2)$

- 问题的时间复杂性下界：$\Omega(n \log n)$

最接近点对问题（一维）

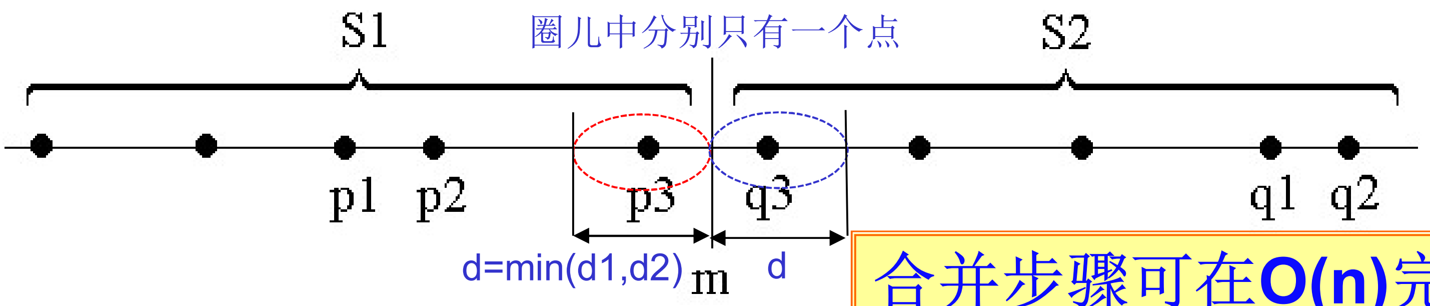
- 一维情形
  - S中的n个点退化为x轴上的n个实数 x1,x2,…,xn。最接近点对即为这n个实数中相差最小的2个实数。
- 简单方法
  - 将S中的点按坐标排好序，用一次线性扫描就可以找出最接近点对
  - 时间复杂性：$O(n \log n)$
  - 排序方法不能推广到二维情形！

最接近点对问题（一维）

❑ 分治方法

- 用x轴上某个点m将S划分为2个子集S1和S2，使得S1={x≤m}；S2={x>m}。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。线性时间选择，O(n)
- 对于S中的任意两个点a和b，至多存在三种情况：

1. a,b均在 S1，假设最接近点对d1=|p1-p2|
2. a,b均在 S2，假设最接近点对d2=|q1-q2|
3. a ,b 分别在 S1和S2，假设最接近点对d3=|p3-q3|，此时p3必然是S1中x坐标最大的点，同时q3是S2中x坐标最小的点

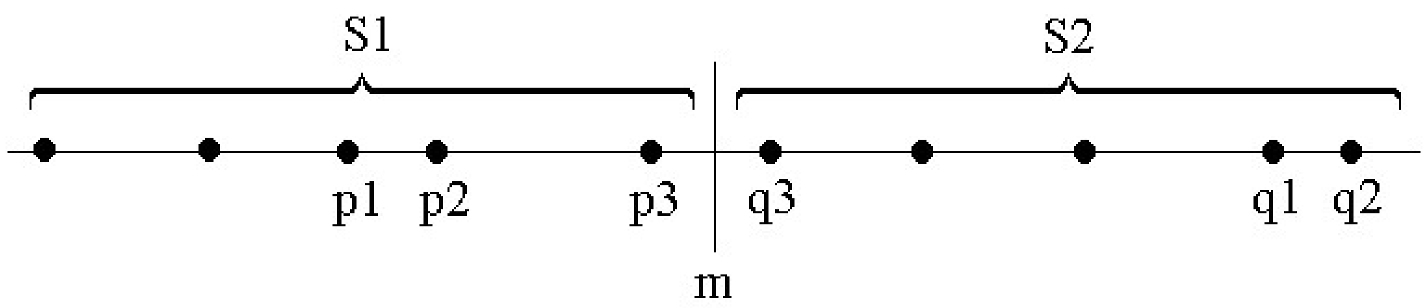


合并步骤可在O(n)完成

最接近点对问题（一维）

分治方法

- 以中位数分割，递归地在S1和S2上找出其最接近点对 $d1=|p1-p2|$和$d2=|q1-q2|$
- 取S1中坐标最大的点p3，S2中坐标最小的点q3，$d3=|p3-q3|$
- $d=\min\{d1, d2, d3\}$



时间复杂性

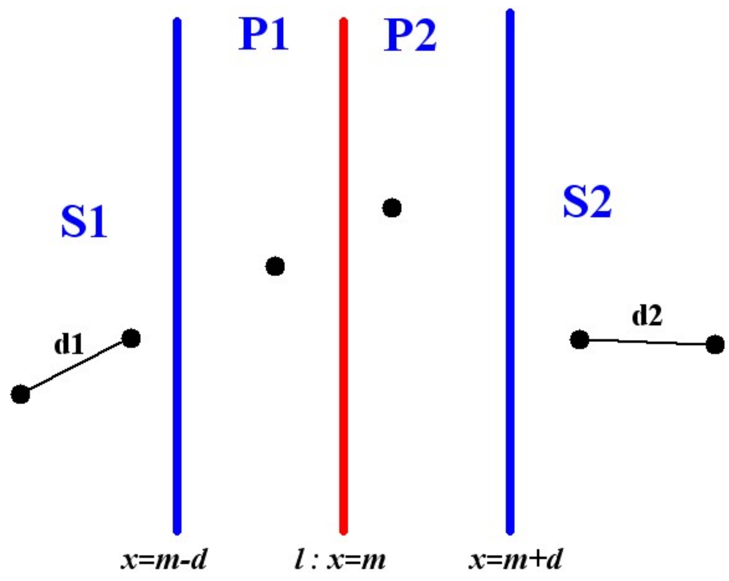
$T(n)=2T(n/2)+O(n)$

$T(n)=O(n \log n)$

最接近点对（二维）

二维情形

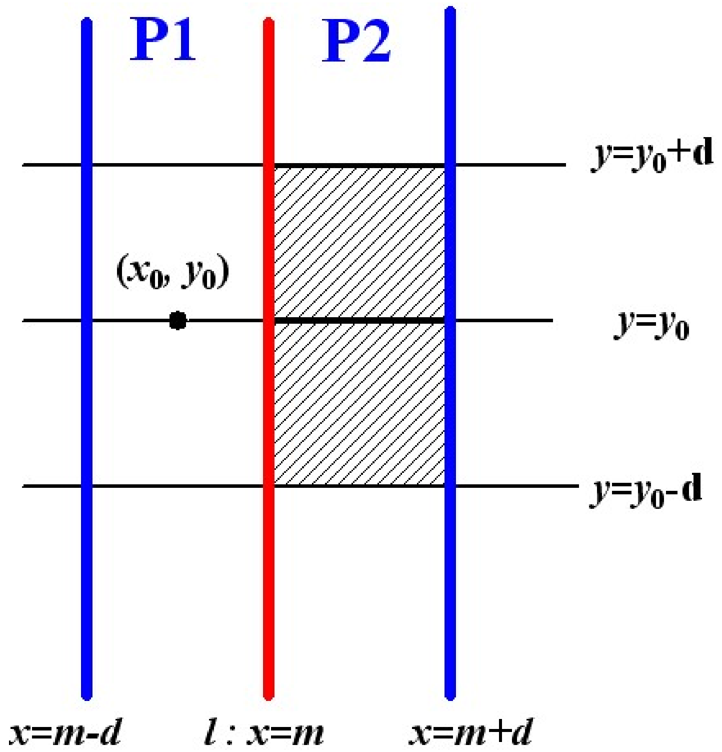
- 选取l: x=m作为分割线，请S分割为S1和S2。其中m取S中各点x坐标的中位数(O(n))
- 递归地在S1和S2中找出其最小距离d1和d2，并设d=min{d1, d2}
- P1={(x, y)∈S1 | m-d<x≤m}, P2={(x, y)∈S2 | m<x≤m+d}
- S中的最接近点对或者是d，或者是某个{p,q}，其中p∈P1且q∈P2. 否则{p,q} ∈S1 or S2
- 能否在线性时间内找到**p,q**？如果可以，合并就可以**O(n)!**



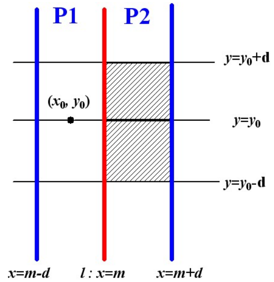
最接近点对（二维）

- 线性时间找到p，q

考虑P1中任意一点p=(x0, y0)，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。

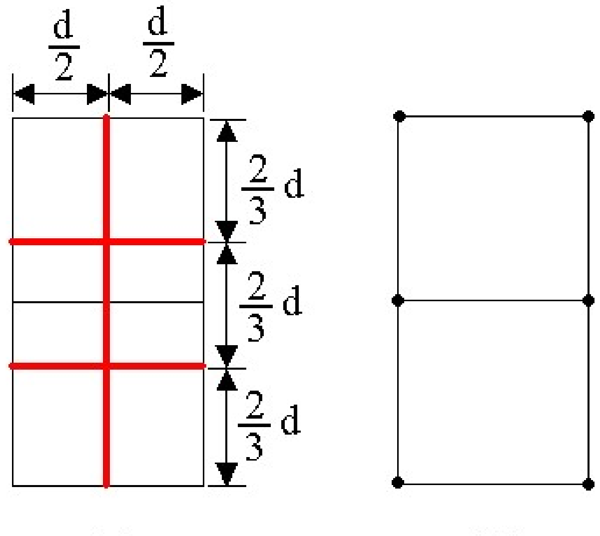


最接近点对（二维）



- 由d的意义可知，P2中任何2个点的距离都不小于d。由此可以推出矩形R中最多只有6个P2中的点

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。
任意一个矩形中两点距离的最大值为5d/6
任意一个矩形中至多含有一个P2中的点
R中至多含有6个P2中的点



(a) (b)

- 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（二维）

- 对于P1中的某个点p，具体考察P2中哪 6 个点？
- 对于P1中的点$p=(x_0, y_0)$
  - 考察P2中的点$\{(x, y)\in P2 \mid y_0-d\leq y \leq y_0+d\}$
    - 这样的点最多有 6 个
  - 计算p与这些点的最小距离
- 若将P1和P2中所有点按其y坐标排好序，则对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（二维）

```
double cpair2 (S)
{
   //S中的点已经按x,y坐标排好序。O(nlogn)
1、m=S中各点x间坐标的中位数;  构造S1={p∈S|x(p)<=m}, S2={p∈S|x(p)>m};
2、d1= cpair2 (S1);  d2= cpair2 (S2);
3、d= min (d1,d2);
4、设P1是S1中距垂直分割线l的距离在d之内的所有点组成的集合;
   P2是S2中距分割线l的距离在d之内所有点组成的集合;
   //P1和P2中点已经按依其y坐标值排序;
5、对于P1中每个点p检查P2中与其y坐标距离在d之内的点(最多6个)，计算最小距离;
   当P1中的扫描指针逐次向上移动时，P2中的扫描指针可在宽为2d的区间内移动;
   设d0是按这种扫描方式找到的点对间的最小距离;
6、return min(d,d0);
}
```

复杂度分析

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

T(n)=O(nlogn)

总结

- 理解递归的概念。
- 掌握设计有效算法的分治策略。
- 通过下面的范例学习分治策略设计技巧。
  1) 二分搜索技术；
  2) 大整数乘法；
  3) Strassen矩阵乘法；
  4) 合并排序和快速排序；
  5) 线性时间选择；
  6) 最接近点对问题；