数据结构与算法

第六章-2 分治算法与排序

裴文杰

计算机科学与技术学院 教授

本讲内容

3.1 Divide-and-Conquer(分治算法)

3.2 Merge Sort（归并排序）

3.3 Quick Sort（快速排序）

3.4 排序问题的下界

Divide-and-Conquer 技术

- Divide-and-Conquer算法的设计
- Divide-and-Conquer算法的分析

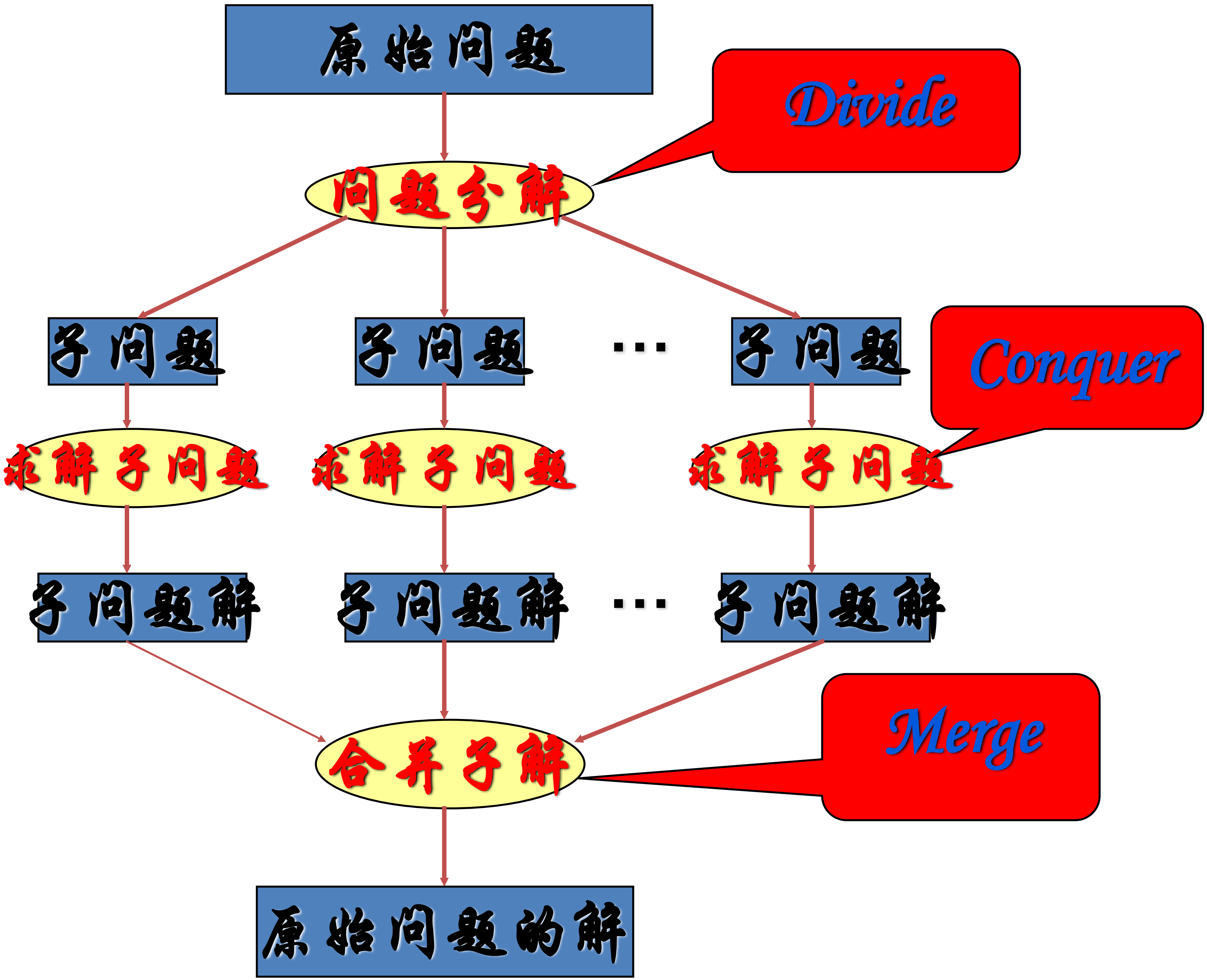
Divide-and-Conquer 技术

- Divide-and-Conquer算法设计过程分为三个阶段
  - Divide： 整个问题划分为多个子问题
  - Conquer：求解各子问题(递归调用设计的算法)
  - Combine：合并子问题的解，形成原始问题的解

如何划分和构造子问题是关键！

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

1. 划分合理：**子问题的形式和原问题一致**
2. 尽可能减少时间复杂性：a尽可能小，b尽可能大
3. 子问题尽量划分均衡


原始问题
Divide
问题分解
子问题
子问题
…
子问题
Conquer
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
Merge
合并子解
原始问题的解

Divide-and-Conquer 技术

- **Divide-and-Conquer 算法分析**
  - 分析过程
    - 建立递归方程
    - 求解算法复杂度
  - 递归方程的建立方法
    - 设输入大小为n，$T(n)$为时间复杂性
    - 当$n<c$, $T(n)=\theta(1)$

Divide-and-Conquer 技术

- **Divide-and-Conquer 算法分析**
  - ❖ 递归方程的建立方法
    - 设输入大小为n，$T(n)$为时间复杂性
    - 当$n<c$, $T(n)=\Theta(1)$
  - **Divide阶段的时间复杂性**
    - 划分问题为a个子问题。
    - 每个子问题大小为n/b。
    - 划分时间可直接得到=$D(n)$
  - **Conquer阶段的时间复杂性**
    - 递归调用
    - Conquer时间= $aT(n/b)$
  - **Combine阶段的时间复杂性**
    - 时间可以直接得到=$C(n)$

Divide-and-Conquer 技术

- **Divide-and-Conquer 算法分析**

  - 总之
    - $T(n)=\Theta(1)$ if n<c
    - $T(n)=aT(n/b)+D(n)+C(n)$ *otherwise*
  - 求解递归方程 $T(n)$
    - 使用第二章的方法

Divide-and-Conquer 技术

- 分治算法实例
  - 求Max和Min问题
  - 大数乘法
  - Chessboard Cover
  - 中位数问题

实例问题一： 求max与min问题

- 问题定义

输入： 数组$A[1,\ldots,n]$
输出： A中的max和min

该算法的核心操作：
比较大小

通常，直接扫描需要$2n$-2次比较操作。
有没有更快的算法？

分治算法？
我们给出仅需$\lceil 3n/2-2 \rceil$次比较操作的算法。

实例问题一：求max与min问题

- 算法一

基于任务不同将问题划分成两个子问题

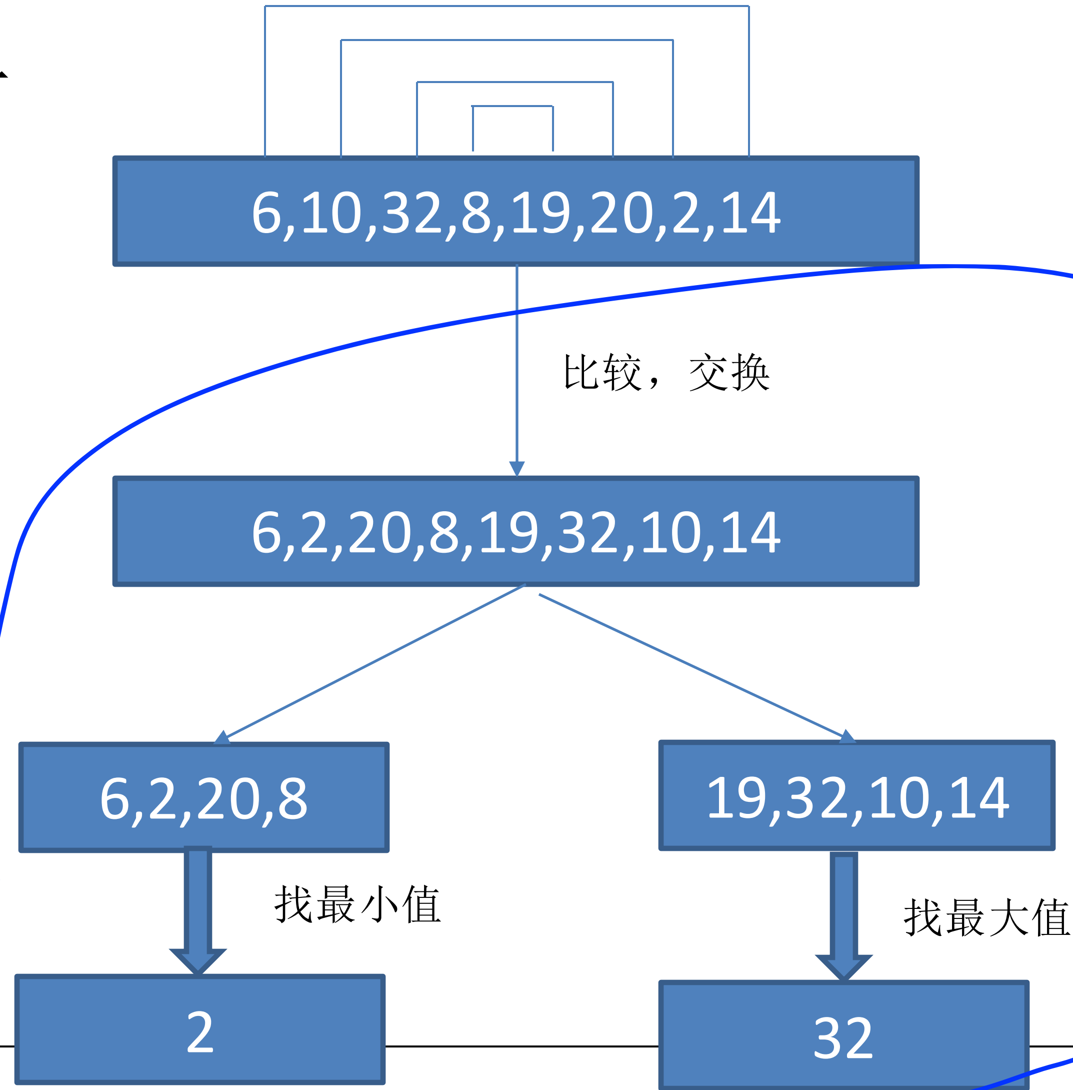
1）一个子问题求解max

2）另一个子问题求解min

将$A[i]$与$A[n-i+1]$比较，$i=1,2,\ldots,n/2$
较小的元素放在前面，较大的元素放在后面，
最小元素出现在$A[1,2,\ldots\lceil n/2\rceil]$中，最大元素$A[\lceil n/2\rceil,\ldots,n]$中

实例问题一： 求max与min问题

- 算法一



实例问题一：求max与min问题

Max-min(A)

Input: 数组$A[1,\ldots,n]$

Output: 数组$A[1,\ldots,n]$中的max和min

1. For $i \leftarrow 1$ To $n/2$ Do
2. IF $A[i] > A[n-i+1]$ THEN swap($A[i]$,$A[n-i+1]$);
3. max $\leftarrow A[n]$; min $\leftarrow A[1]$;
4. For $i \leftarrow 2$ To $\lceil n/2 \rceil$ Do
5. IF $A[i] <$ min THEN min $\leftarrow A[i]$;
6. IF $A[n-i+1] >$ max THEN max $\leftarrow A[n-i+1]$;
7. print max, min;

算法复杂度：
$\lceil 3n/2-2 \rceil$次比较操作

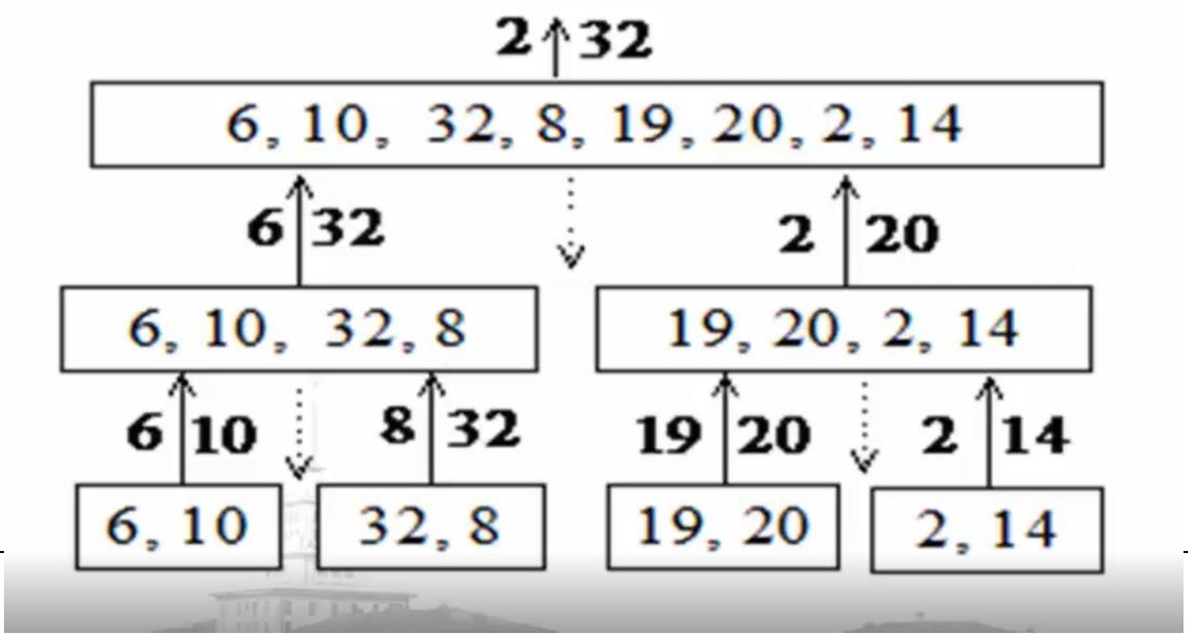
实例问题一： 求max与min问题

- 算法二

Divide:　将n个元素的序列分解为各具有n/2个元素的两个子序列

Conquer:　递归地选出两个子序列的max和min元素

Combine:　合并两个子序列的答案



实例问题一： 求max与min问题

- 算法二

```
Max-min
Input:  数组A[1,…,n]
Output: (x, y) , A中的最小元素和最大元素
  1. Max-min(1, n)

过程: Max-min(low, high)
  1. IF high-low = 1
  2.       IF  A[low] < A[high] THEN  return (A[low], A[high])
  3.        ELSE return (A[high], A[low])
  4. ELSE
  5.        mid ← (low+high)/2
  6.        (x1,y1) ← Max-min(low,mid)
  7.         (x2,y2) ← Max-min(mid+1,high)
  8.        x ← min{x1,x2}
  9.        y ← max{y1,y2}
  10.       return (x,y)
```

标准递归伪代码

实例问题一： 求max与min问题

- 算法二
  - ❖ 算法复杂度

$$T(1)=0$$
$$T(2)=1$$
$$\begin{aligned}T(n)&=2T(n/2)+2\\&=2^2T(n/2^2)+2^2+2\\&=\ldots\\&=2^{k-1}T(2)+2^{k-1}+2^{k-2}+\ldots+2^2+2\\&=2^{k-1}+2^k-2\\&=n/2+n-2\\&=3n/2-2\end{aligned}$$

$n=2^k$

Master定理？ $\theta(n)$


实例问题二
大数乘法

实例问题二： 大数乘法

- 问题定义

输入： n位二进制整数X和Y
输出： X和Y的乘积

直接相乘很耗时。
分治算法？如何构造子问题？

实例问题二：大数乘法

- 简单分治算法

	n/2位	n/2位
X=	A	B

	n/2位	n/2位
Y=	C	D

$$XY = (A2^{n/2} + B)(C2^{n/2} + D)$$
$$= AC2^{n} + (AD+BC)2^{n/2} + BD$$

算法

1. 划分产生A,B,C,D;
2. 计算n/2位乘法AC、AD、BC、BD;
3. 计算BC+AD;
4. AC左移n位，(BC+AD)左移n/2位;
5. 计算XY。

算法复杂性:

$T(n)=4T(n/2)+\theta(n)$，使用Master定理：$T(n)=O(n^2)$

实例问题二：大数乘法

- 改进分治算法

n/2位 n/2位　　　　n/2位 n/2位

X= | A | B |　　　　Y= | C | D |

$$XY = (A2^{n/2} + B)(C2^{n/2} + D)$$
$$= AC2^{n} + (AD+BC)2^{n/2} + BD$$
$$= AC2^{n} + ((A-B)(D-C)+AC+BD)2^{n/2} + BD$$

减少子问题数量

算法

1. 计算A-B和D-C；
2. 计算n/2位乘法AC、BD、(A-B)(C-D)；
3. 计算(A-B)(D-C)+BC+AD；
4. AC左移n位，((A-B)(D-C)+BC+AD)左移n/2位；
5. 计算XY

算法复杂性：

$T(n)=3T(n/2)+\theta(n)$

实例问题二： 大数乘法

- 改进分治算法
  - 建立递归方程

$$T(n)=\Theta(1) \qquad \text{if } n=1$$

$$T(n)=3T(n/2) + \Theta(n) \qquad \text{if } n>1$$

  - 使用Master定理

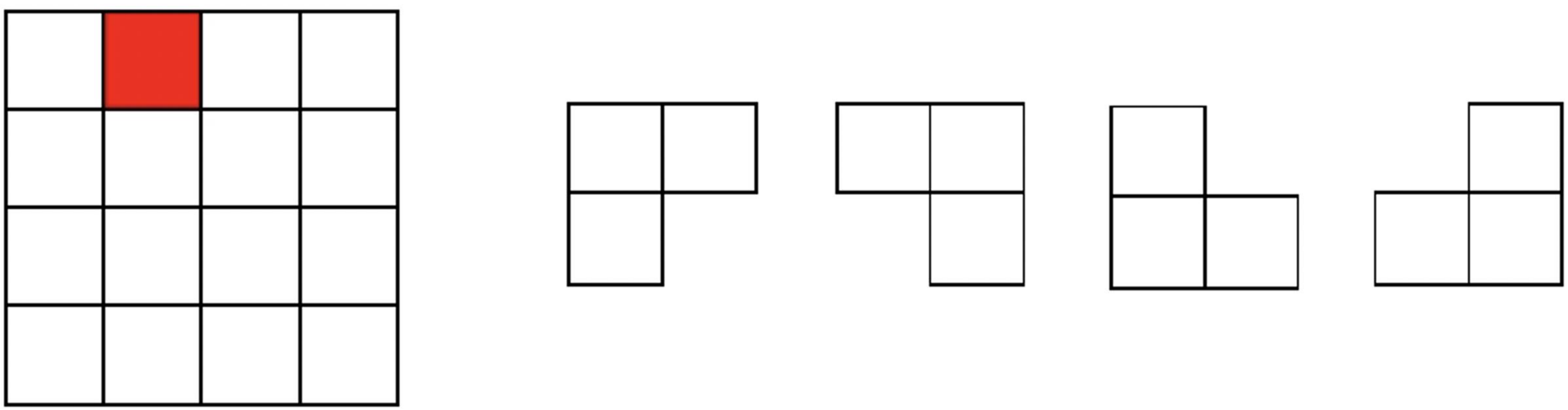
$$T(n)=O(n^{\log 3})=O(n^{1.59})$$

例： 棋盘覆盖问题

实例问题三：Chessboard Cover

- **问题定义**

在一个$2^k \times 2^k$的棋盘上， 只有一个格子是不同的，称为“奇异格”。棋盘覆盖问题，要求采用下列四种L形卡片来覆盖除去“奇异格”的整个棋盘，并且没有重叠。

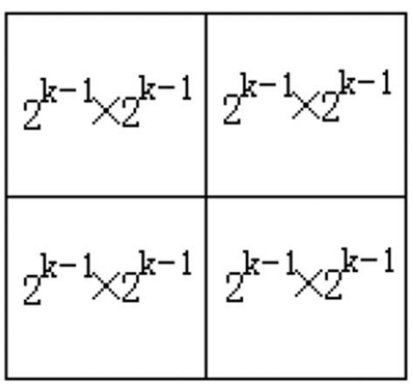


实例问题三：Chessboard Cover

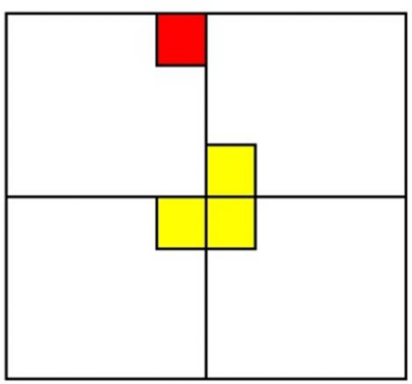
- 分治算法

采用分治算法，难点在于如何构造子问题

- ❖ 当k>0, 将$2^k\times2^k$棋盘格分成四个 $2^{k-1}\times2^{k-1}$子棋盘；
  - 那么那个“奇异格”必然在其中一个子棋盘中，而其他三个子棋盘不包含“奇异格”：子问题形式与原问题不一致
- ❖ 将L形卡片放置在不包含“奇异格”的三个子棋盘的交界处；
  - 这样我们得到四个棋盘覆盖子问题。
- ❖ 递归求解直到我们得到1×1大小的子棋盘。



(a)


(b)

Complexity

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

$$T(k)=\Theta(4^k)$$

实例问题四

中位数问题

实例问题四：中位数选取问题

- **问题定义**

Input: 由n个数构成的多重集合X 　元素可以有重复

Output: $x \in X$使得 $-1 \le |\{y \in X \mid y < x\}| - |\{y \in X \mid y > x\}| \le 1$

4 3 5 8 1 9 2 6 7

先排序，再取中位数？O(nlogn)，可以更快吗？

实例问题四：中位数选取问题

- 中位数选取问题的复杂度

[Blum et al. *STOC*'72 & *JCSS*'73]

A "shining" paper by five authors:

Manuel Blum (Turing Award 1995)

Robert W. Floyd (Turing Award 1978)

Vaughan R. Pratt

Ronald L. Rivest (Turing Award 2002)

Robert E. Tarjan (Turing Award 1986)

从n个数中选取中位数需要的比较操作的次数介于 **$1.5n$**到 **$5.43n$**之间

实例问题四：中位数选取问题

- 上界
  - $3n + o(n)$ by Schonhage, Paterson, and Pippenger (*JCSS* 1975).
  - $2.95n$ by Dor and Zwick (*SODA* 1995, *SIAM Journal on Computing* 1999).

- 下界
  - $2n+o(n)$ by Bent and John (*STOC* 1985)
  - $(2+2^{-80})n$ by Dor and Zwick (*FOCS* 1996, *SIAM Journal on Discrete Math* 2001).

实例问题四：中位数选取问题

- 线性时间选择
  - 本节讨论如何在$O(n)$时间内从n个不同的数中选取第i大的元素
  - 中位数问题也就解决了，因为选取中位数即选择第n/2大的元素

Input: n个(不同)数构成的集合X,整数i,其中$1 \le i \le n$

Output: $x \in X$使得X中恰有i-1个元素小于x

实例问题四：中位数选取问题

- 分治算法
  - 如何构造子问题？
    - 将集合X等分成两部分，然后分别求取中位数？怎么合并？

找到一个数x，将X分成两部分：

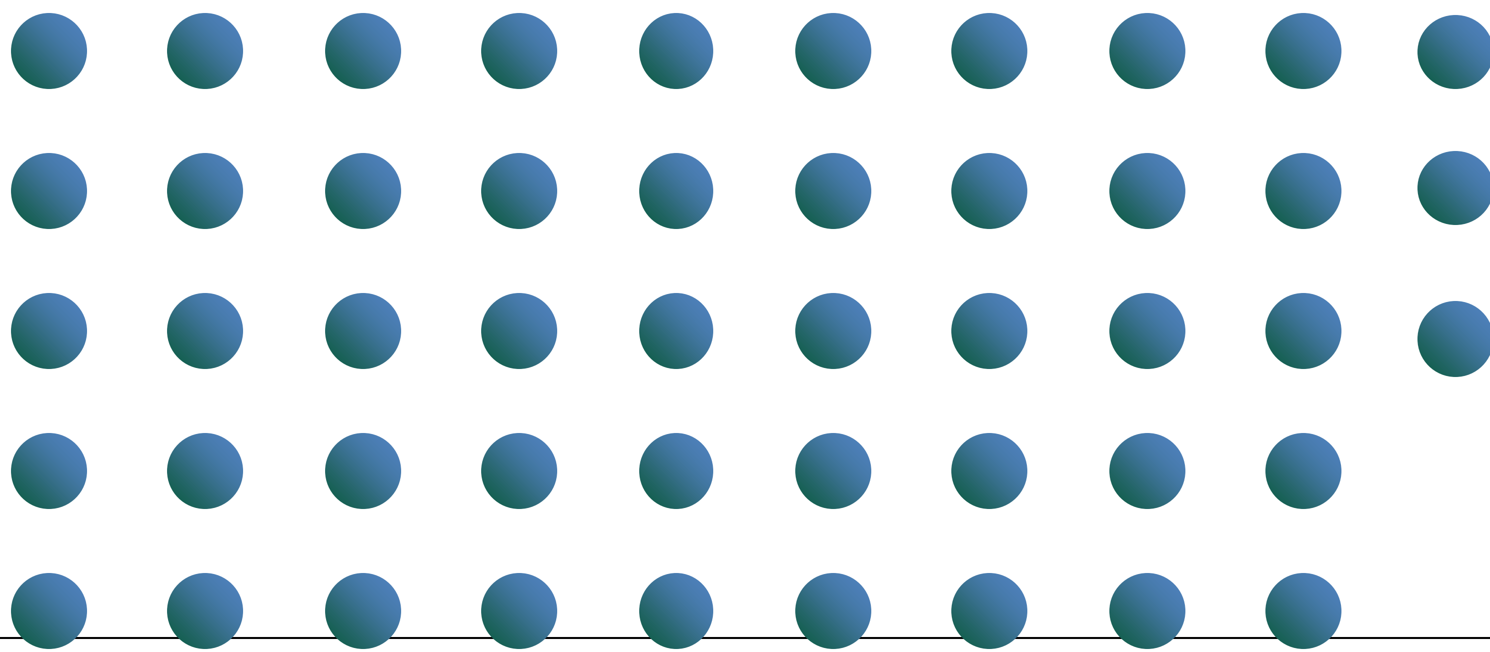
- **小于x的数组成一个子集X_1，大于x的数组成一个子集X_2；**
- **如果$|X_1|>\frac{1}{2}|X|$，那么中位数在X_1中，否则在X_2中；**
- **于是在X_1或者X_2中寻找中位数构成子问题，通过递归求解。**

关键：x将X划分越均衡，子问题规模越小，如何确定（估计）合适的x？

实例问题四：中位数选取问题

- 求解步骤

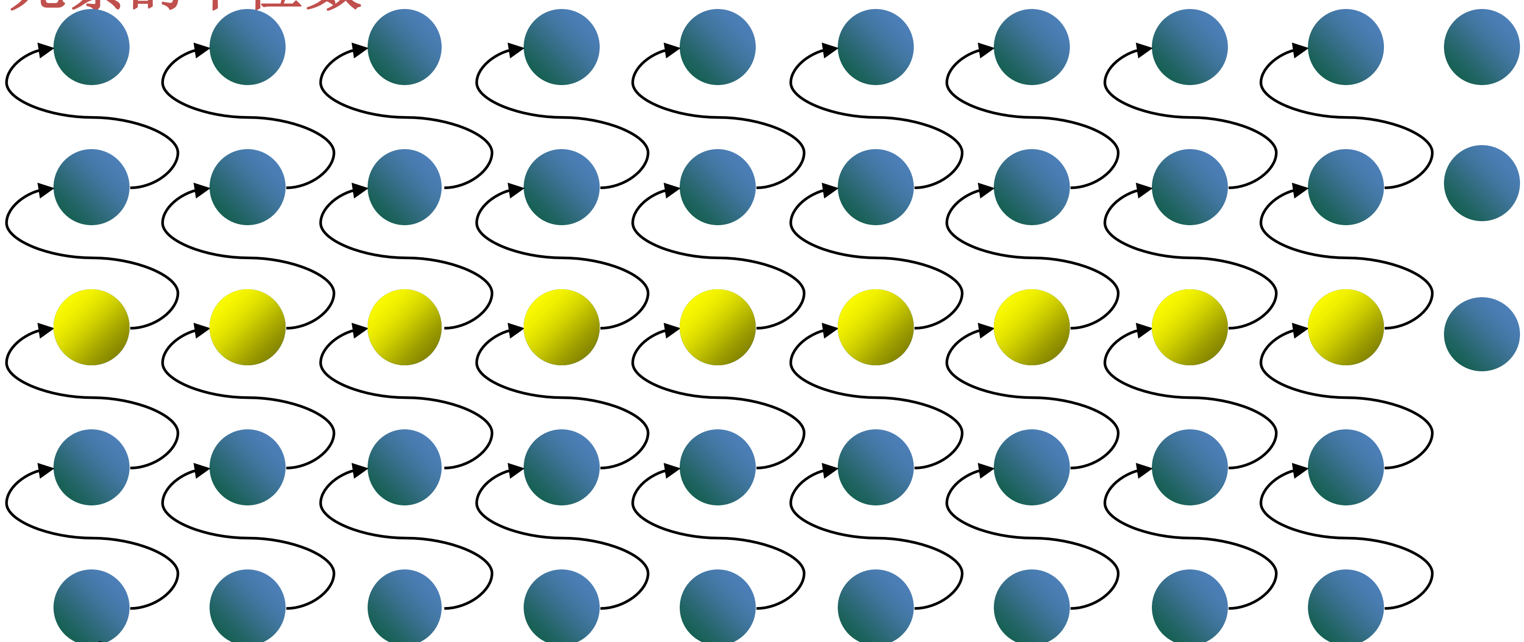
第一步：分组，每组5个数
最后一组可能少于5个数



实例问题四：中位数选取问题

- **求解步骤**

  第二步：将每组数分别用插入排序（InsertSort），选出每组元素的中位数

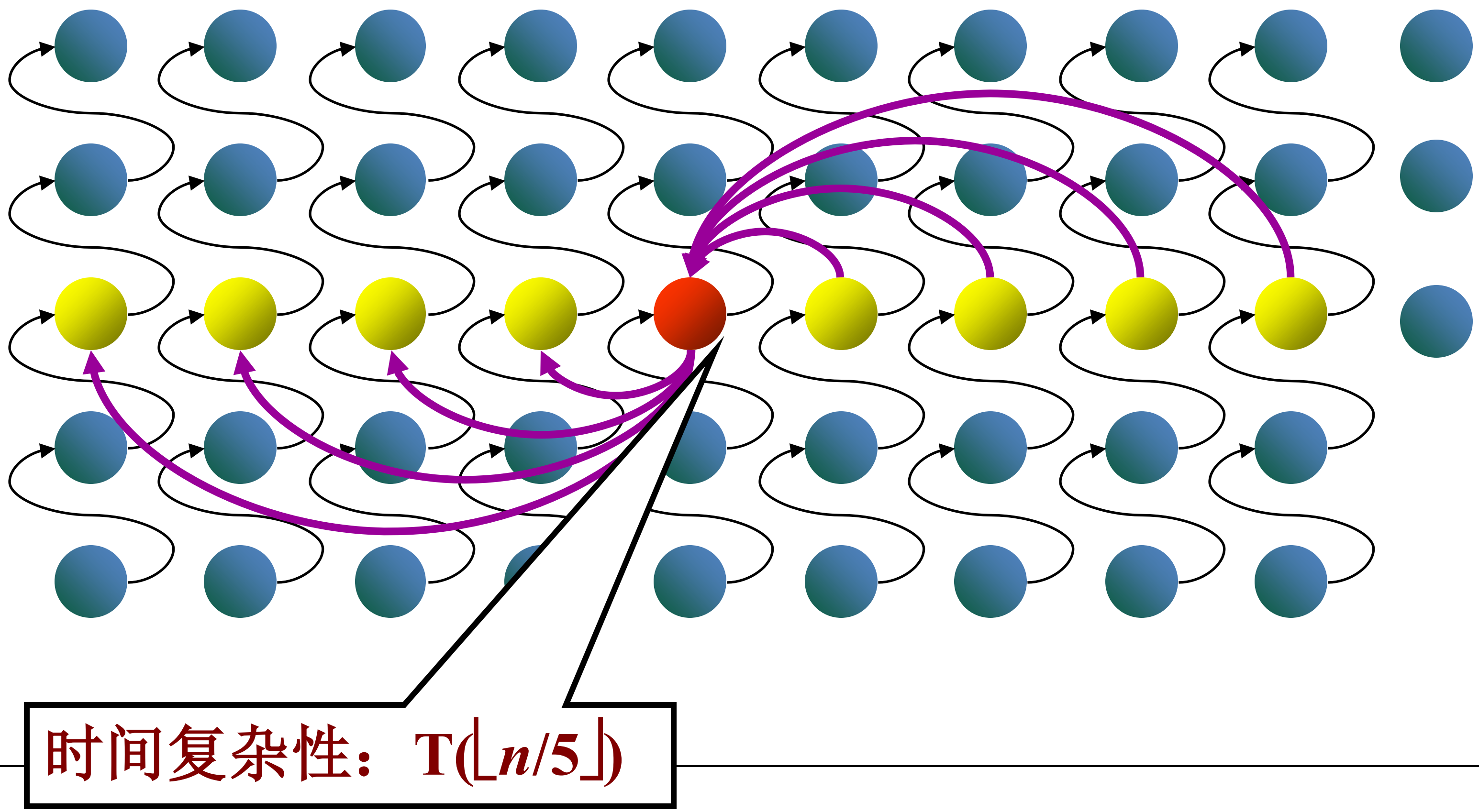


排序每组数时，比较操作的次数为5(5-1)/2=10次
总共需要10*$\lceil n/5 \rceil$次比较操作

实例问题四：中位数选取问题

- **求解步骤**

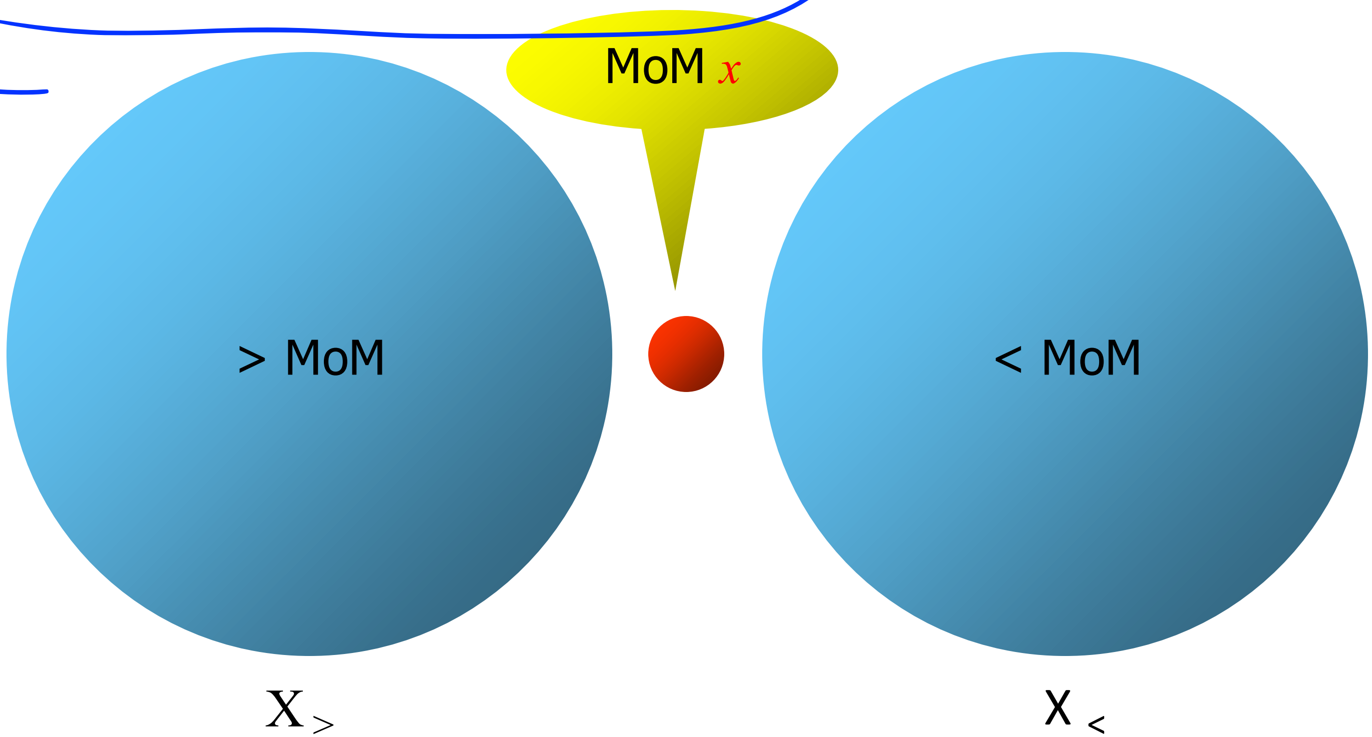
第三步：递归调用算法求得这些中位数的中位数(**MoM**)



实例问题四：中位数选取问题

- **求解步骤**

第四步：用 MoM 完成划分



时间复杂性$O(n)$

实例问题四：中位数选取问题

- 求解步骤

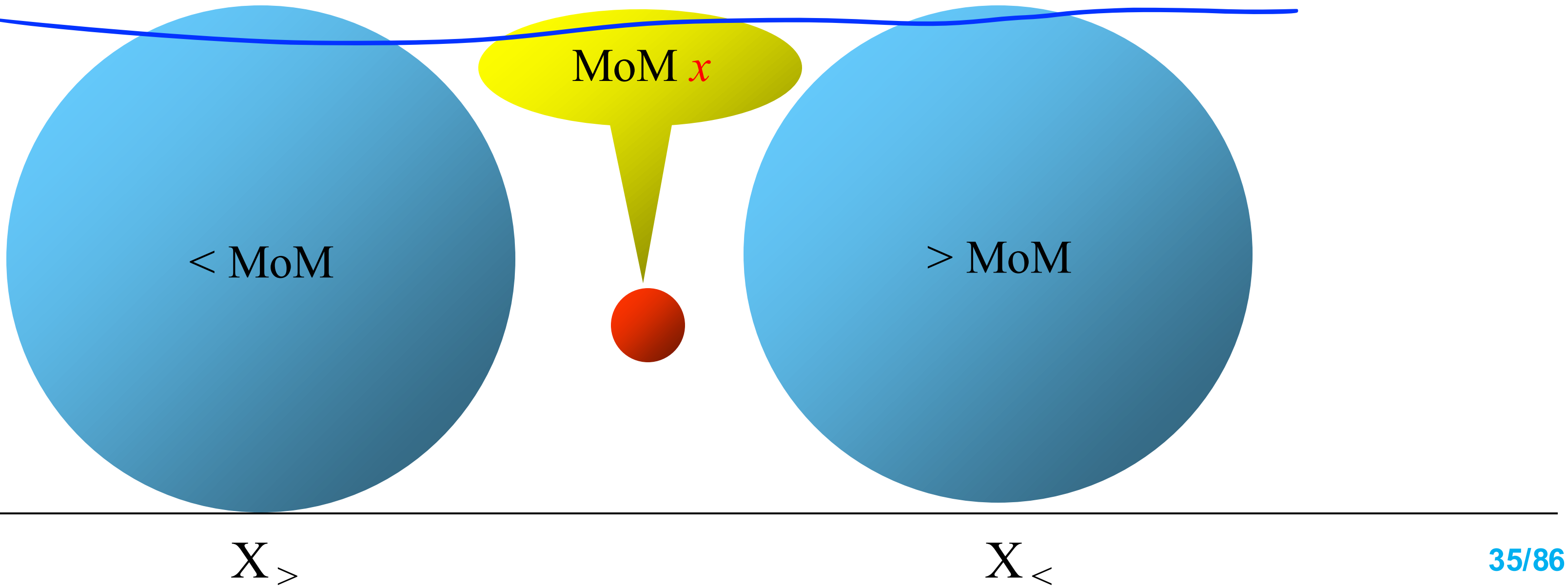
第五步：递归

设x是中位数的中位数(MoM), 划分完成后其下标为*k*

如果$i=k$,则返回x

如果$i<k$,则在第一个部分递归选取第i大的数

如果$i>k$,则在第三个部分递归选取第$(i-k)$大的数



实例问题四：中位数选取问题

算法Select(A, i)

Input: 数组$A[1:n]$, $1\le i \le n$

Output: $A[1:n]$中的第i-大的数

```
for j←1  to  n/5                                           第一步
    InsertSort(A[(j-1)*5+1 : (j-1)*5+5]);                  第二步
    swap(A[j], A[(j-1)*5+3]);
x ←Select(A[1: n/5],  n/10 );                              第三步
k ←partition(A[1:n], x);                                   第四步
if        k=i  then   return x;
else if  k>i  then   return Select(A[1:k-1], i);           第五步
else                 return Select(A[k+1:n], i-k);
```

A[1:k-1]是小于x的元素集合

实例问题四：中位数选取问题

算法Select(A, i)

Input: 数组$A[1:n]$, $1\leq i \leq n$

Output: $A[1:n]$中的第i-大的数

```
for j←1 to n/5
    InsertSort(A[(j-1)*5+1 : (j-1)*5+5]);
    swap(A[j], A[(j-1)*5+3]);
x ←Select(A[1: n/5], n/10 );
k ←partition(A[1:n], x);
if        k=i  then  return x;
else if  k>i  then  return Select(A[1:k-1], i);
else                 return Select(A[k+1:n], i-k);
```

$10*\lceil n/5 \rceil = O(n)$

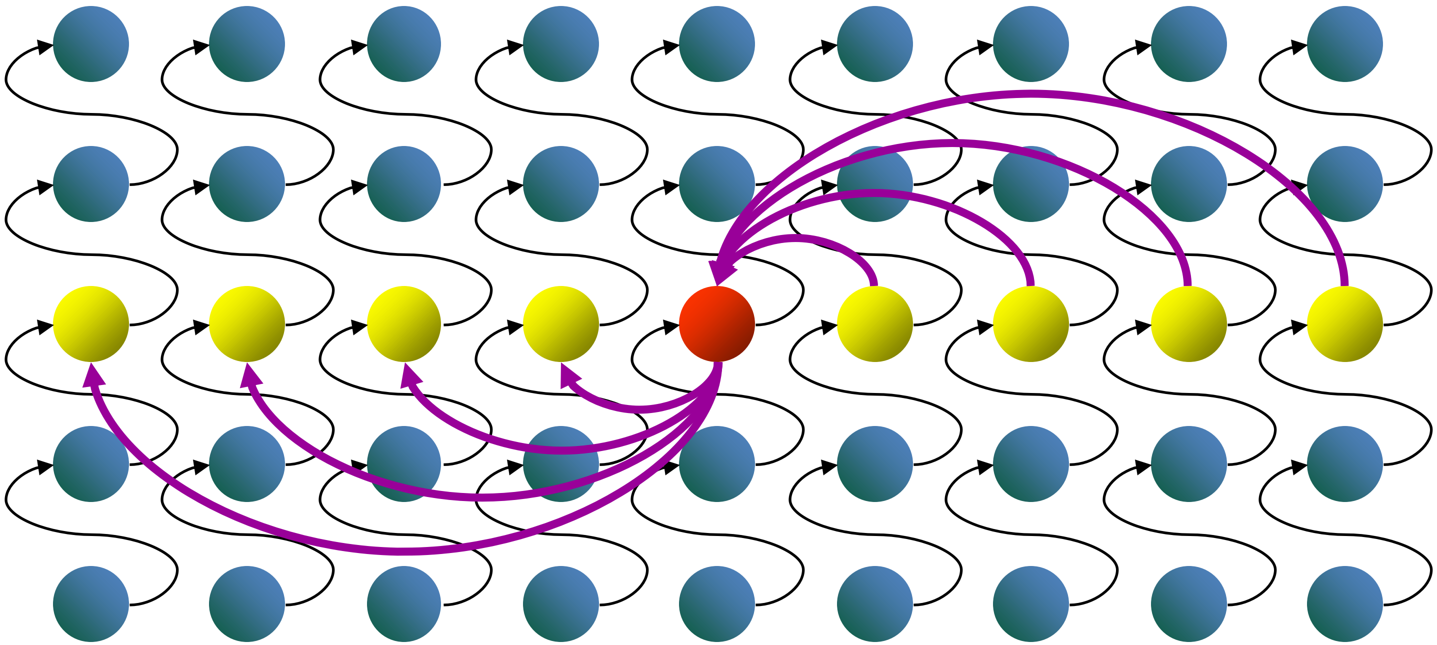
$T(\lfloor n/5 \rfloor)$

$O(n)$

???

实例问题四：中位数选取问题

观察第五步的处理过程

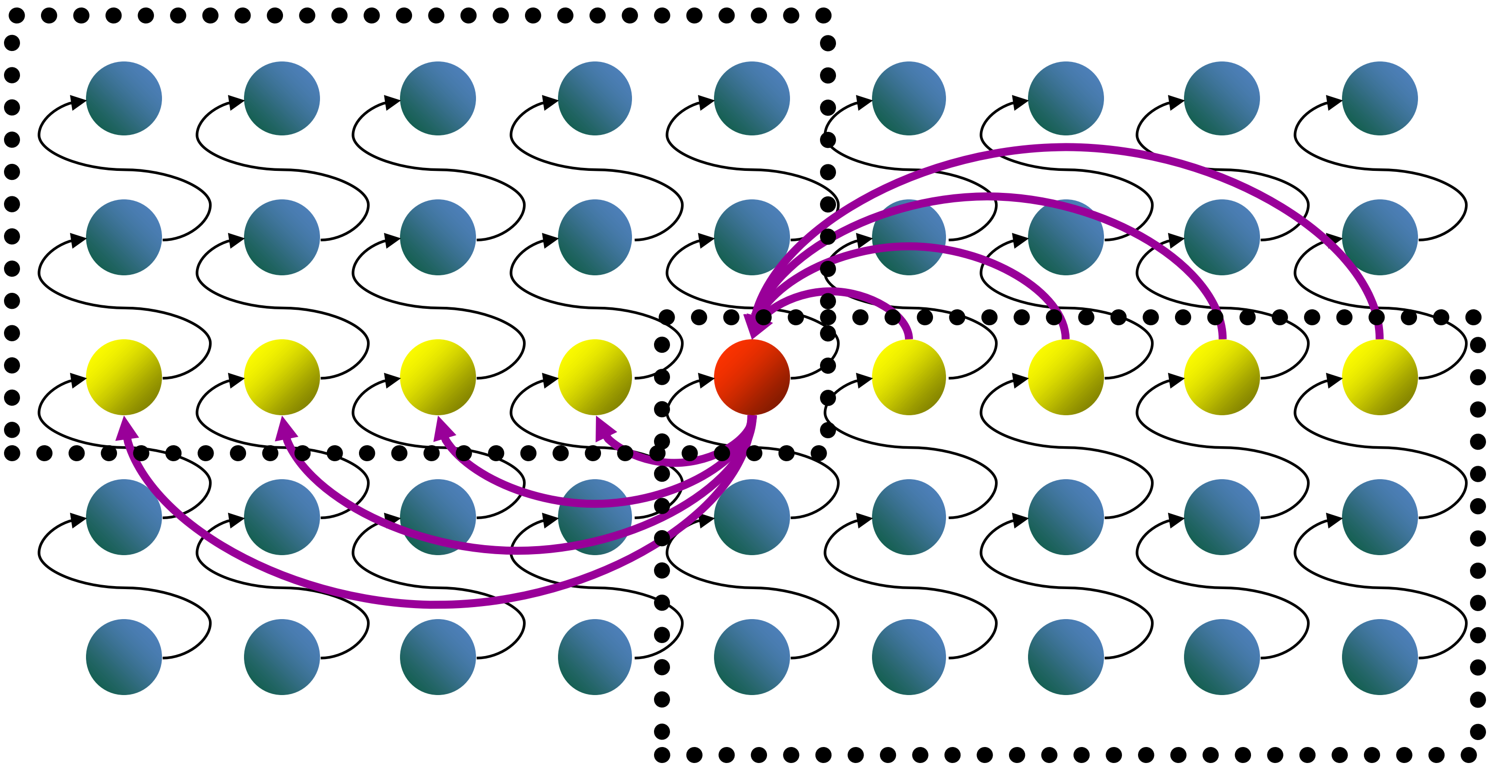


关键问题在于由MoM分成的两部分集合的均衡程度，越均衡，时间复杂性越低。

实例问题四：中位数选取问题

第五步至少删除了$\lfloor 3n/10 \rfloor$个数，准确地是，$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$

因为至少有 $\frac{3n}{10} - 6$元素大于x，和小于x，因此递归调用至多作用于 $\frac{7n}{10} + 6$个元素。



"-2"：除去当n不能被5整除时产生的元素少于5的那个组，和包含x的那个组

则第五步递归产生的时间开销不超过$T(7n/10+6)$

实例问题四：中位数选取问题

- 时间复杂性

$$T(n) \le \begin{cases} \Theta(1) & \text{if } n \le C \text{（C>70）} \\ T\lfloor n/5 \rfloor + T(7n/10+6) + \Theta(n) & \text{if } n > C \end{cases}$$

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

采用数学归纳法证明。
证明过程详见算法导论9.3节。
C的取值跟证明过程有关。

本讲内容

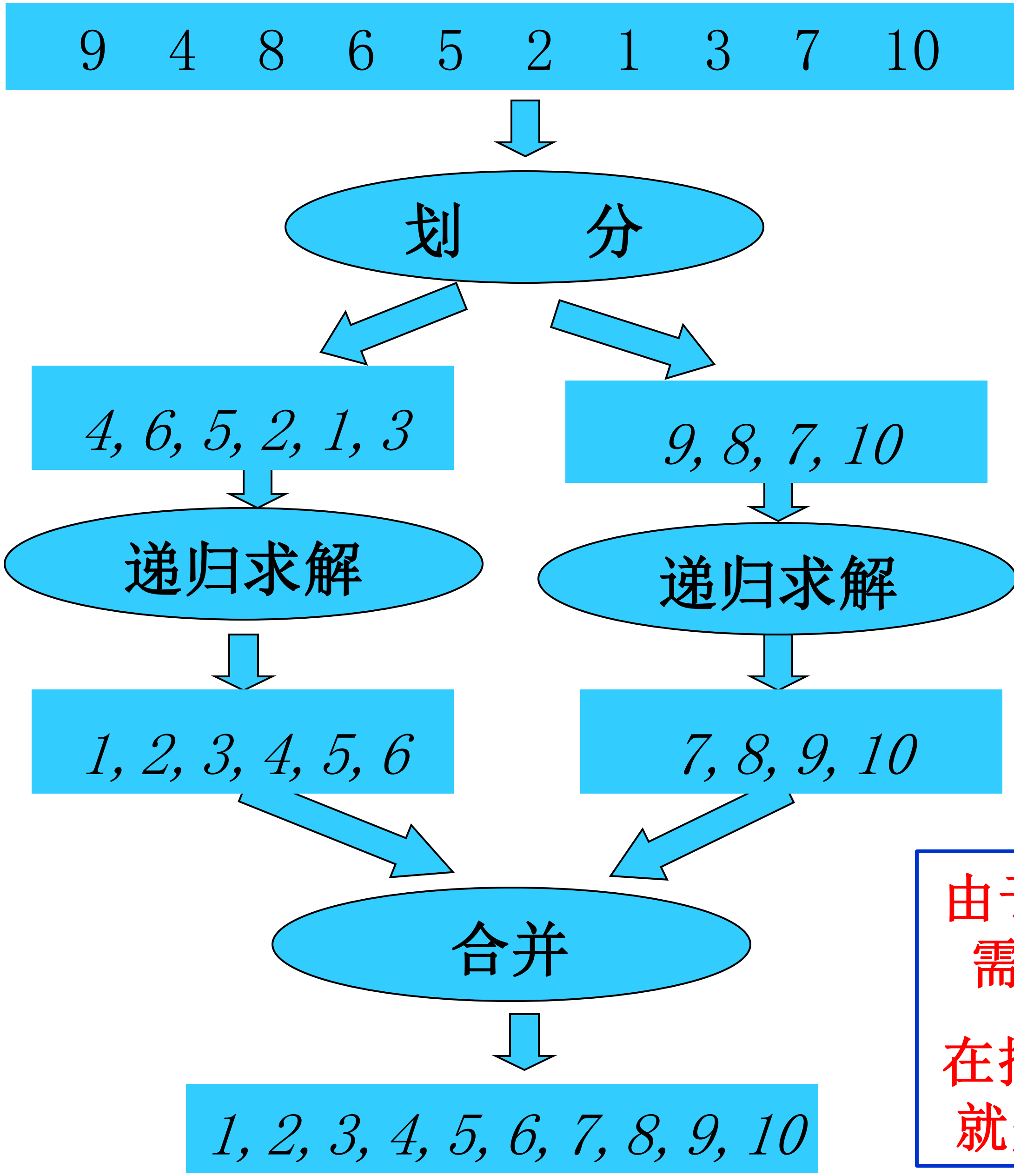
3.1　Divide-and-Conquer(分治算法)

3.2　Merge Sort（归并排序）

3.3　Quick Sort（快速排序）

3.4　排序问题的下界

基于分治思想的排序算法



划分的策略

1. 选择一个位置将数组划分成两个部分
   Mergesort （归并排序）

2. 选择一个划分标准x, 根据元素与x的大小关系来划分
   Quicksort (快速排序)

合并策略

不同的划分策略对应不同的合并策略

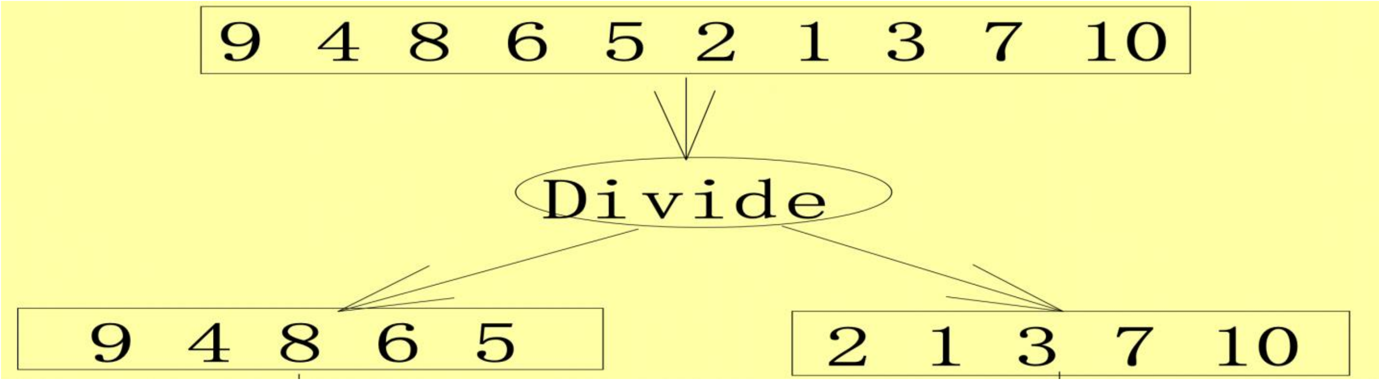
由于分治思想涉及到递归调用，需要关心子问题的最一般形式

在排序问题中，子问题一般形式就是将$A[i, \cdots, j]$中的元素排序

Merge Sort（归并排序）

- **Divide**
  - ❖ 对于序列 $A[i,\ldots,j]$, 选择一个划分位置k：
    - 第一个子问题是$A[1,\ldots,k]$
    - 第二个子问题是$A[k+1,\ldots,j]$
  - ❖ 为使得两个问题的大小大致相当，**k**可以如下产生：

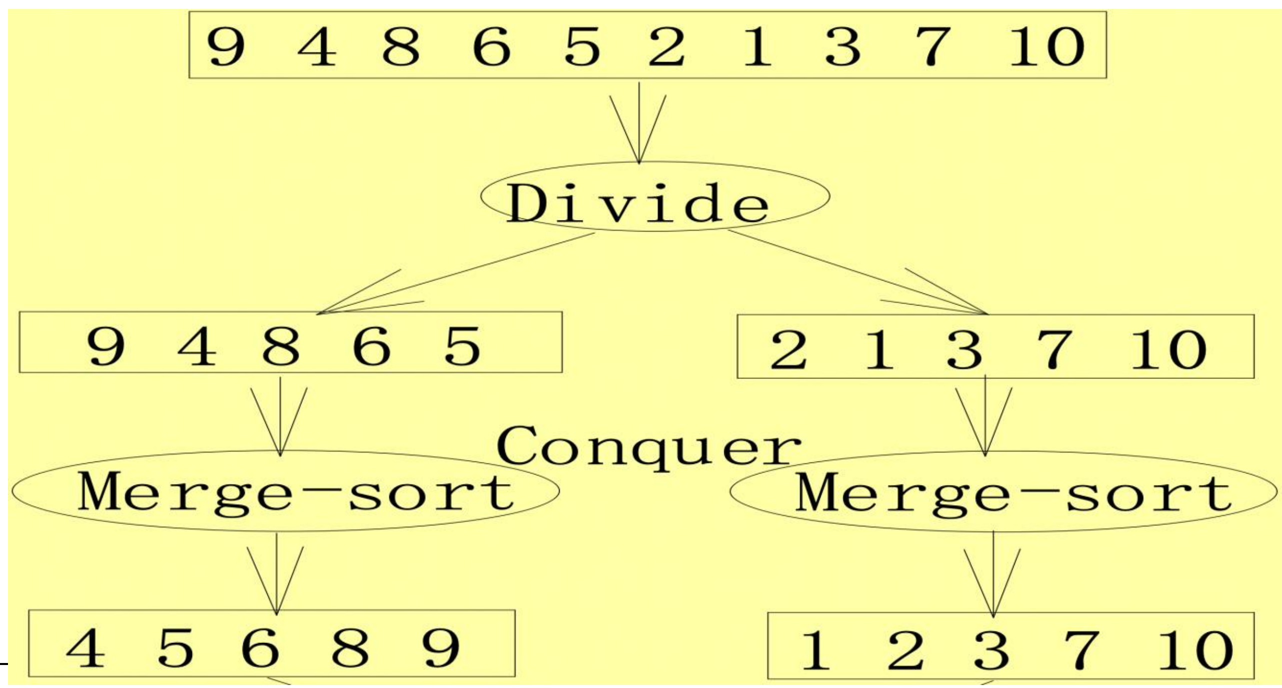
$$k = (i+j)/2$$



Merge Sort（归并排序）

- **Conquer**
  - ❖ 递归求解就是算法的递归调用
    - Mergesort(A,i,k)； //求解第一个子问题
    - Mergesort(A,k+1,j)； //求解第二个子问题

Divide没有时间消耗；
但Combine并不直接，需要设计算法



Merge Sort（归并排序）

重新定义一个数组B，与输入数组A相同大小，用来临时存储合并后的排序结果。
注意：归并排序用了额外的存储空间O(n)

- **Combine （序列i:k和序列k+1:j）**

combine: 将两个有序序列合并成一个有序序列

1. $l \leftarrow i$; $h \leftarrow k+1$; $t=i$ //设置两个指针，分别指向两个序列开头

2. While $l \le k$ & $h \le j$ **Do** //注意while条件：当两个子问题都有剩余元素时

    IF $A[l] < A[h]$ **THEN** $B[t] \leftarrow A[l]$; $l \leftarrow l+1$; $t \leftarrow t+1$; //B为和A一样大的数组，用于临时保存排序结果

    ELSE $B[t] \leftarrow A[h]$; $h \leftarrow h+1$; $t \leftarrow t+1$;

3. IF $l \le k$ **THEH** //第一个子问题有剩余元素

    For $v \leftarrow l$ **To** k **Do**

        $B[t] \leftarrow A[v]$; $t \leftarrow t+1$;

4. IF $h \le j$ **THEN** //第二个子问题有剩余元素

    For $v \leftarrow h$ **To** j **Do**

        $B[t] \leftarrow A[v]$; $t \leftarrow t+1$;

4, 5, 6, 8, 9   **1，2，3，7, 10**

combine

1,2,3,4,5,6,7,8,9,10

Merge-sort算法

MergeSort(A,i,j)
Input: $A[i,\ldots,j]$
Output:排序后的$A[i,\ldots,j]$

1. $k \leftarrow (i+j)/2;$
2. $MergeSort(A,i,k);$
3. $MergeSort(A,k+1,j);$
4. $l \leftarrow i;$ $h \leftarrow k+1;$ $t = i$ //设置指针
5. While $l \leq k$ & $h \leq j$ Do
6. IF $A[l] < A[h]$ THEN $B[t] \leftarrow A[l]; l \leftarrow l+1; t \leftarrow t+1;$
7. ELSE $B[t] \leftarrow A[h]; h \leftarrow h+1; t \leftarrow t+1;$
8. IF $l \leq k$ THEH //第一个子问题有剩余元素
9. For $v \leftarrow l$ To k Do
10. $B[t] \leftarrow A[v]; t \leftarrow t+1;$
11. IF $h \leq j$ THEN //第二个子问题有剩余元素
12. For $v \leftarrow h$ To j Do
13. $B[t] \leftarrow A[v]; t \leftarrow t+1;$
14. For $v \leftarrow i$ To j Do //将归并后的数据复制到A中
15. $A[v] \leftarrow B[v];$

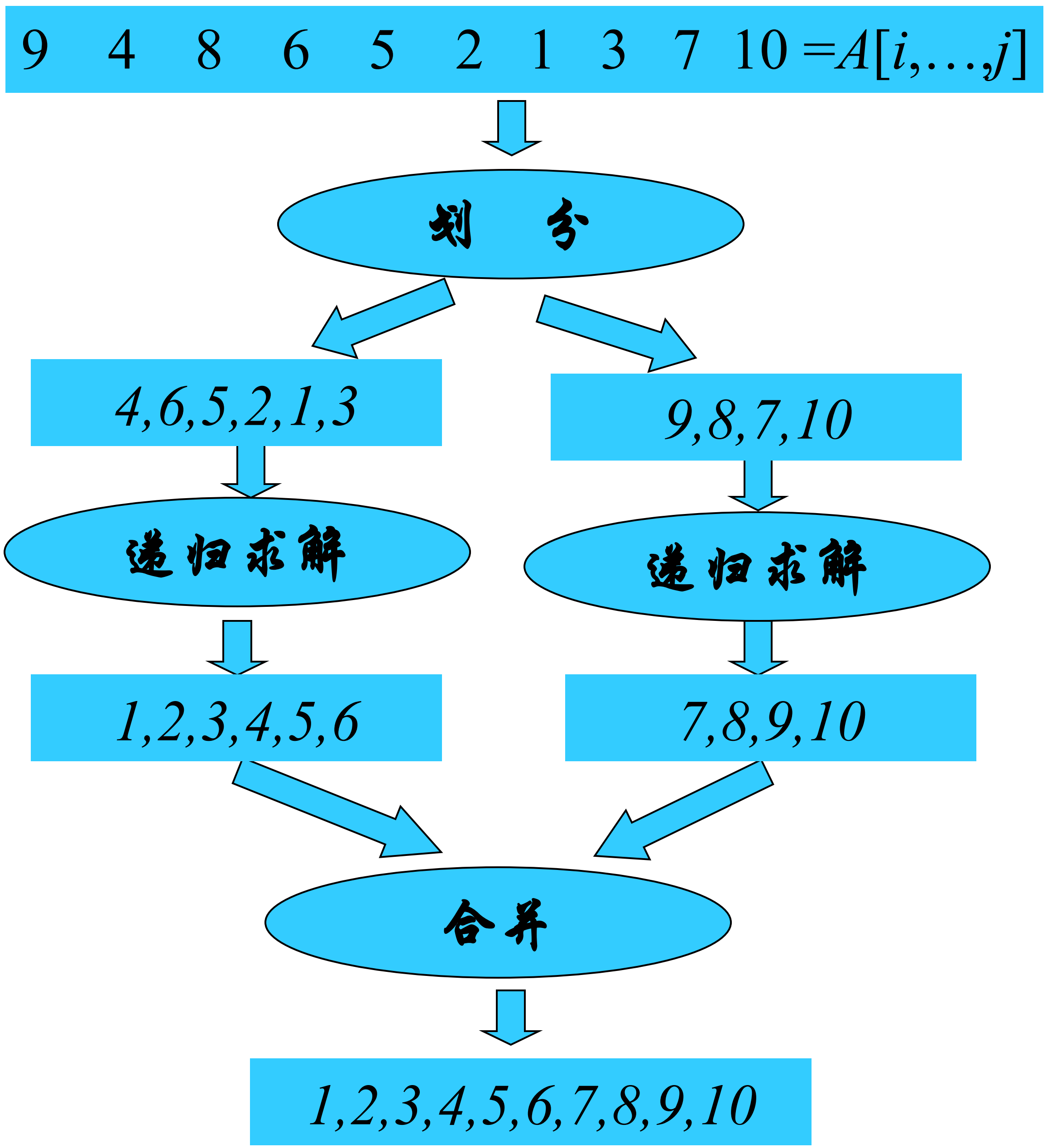
$T(n)=2T(n/2)+O(n)$
$T(n)=O(n\log n)$

Divide没有时间消耗
Combine算法复杂性 $\Theta(n)$

本讲内容

3.1 Divide-and-Conquer(分治算法)
3.2 Merge Sort（归并排序）
3.3 Quick Sort（快速排序）
3.4 排序问题的下界

Quick Sort (or Partition Sort)



基本思想

Divide：

确定一个划分标准x，将数组中小于x的元素放到数组的前面部分，大于x的元素放到数组的后面部分，并返回一个划分k；

Conquer：

递归调用算法将$A[i,\ldots,k]$和$A[k+1,\ldots,j]$求解。

Combine：

无操作

Divide需要时间消耗，而Combine没有时间消耗。

Quick Sort (快速排序)

QuickSort(*A*,*i*,*j*)

Input: $A[i,\ldots,j]$, x

Output: 排序后的$A[i,\ldots,j]$

1. **If (i<j)**
2. 选择划分元素x;
3. k=**Partition**(A,i,j,x); //用x完成划分
4. **QuickSort**(A,i,k); //递归求解子问题
5. **QuickSort**($A,k+1,j$);

无需额外的合并操作为：原址排序

在排序算法中，如果输入数组中仅有常数个元素需要在排序过程中存储在数组之外，则称排序算法是原址的。
插入排序，快速排序都是原址排序，归并排序不是原址排序。

Quick Sort (快速排序)

- **Divide: 关键步骤 (Partition)**

9	4	8	6	5	2	1	3	7	10	$=A[i,\ldots,j]$
9	4	8	6	5	2	1	3	7	10	$=A[i,\ldots,j]$
3	4	8	6	5	2	1	9	7	10	$=A[i,\ldots,j]$
3	4	1	6	5	2	8	9	7	10	$=A[i,\ldots,j]$
3	4	1	6	5	2	8	9	7	10	$=A[i,\ldots,j]$

3 4 1 6 5 2 $=A[i,\ldots,high]$ 8 9 7 10 $=A[high+1,\ldots,j]$

Divide：

确定一个划分标准x（可以随机，或者指定，或者按照一定规则（例如第一个元素）等），将数组中小于x的元素放到数组的前面部分，大于x的元素放到数组的后面部分，并返回一个划分k；

问：如何不利用额外存储空间进行原址分割操作？

指定划分标准 $x=7$

指针low从低区中找出应放到x之后的第一个元素，

指针high从高区中找出应放到x之前的第一个元素，

如果low<high, 则交换low和high标记的两个元素的位置；

否则划分位置就是high。

Quick Sort (快速排序)

Partition(A,i,j,x)

Input: $A[i,\ldots,j]$, x

Output: 划分位置k使得$A[i,\ldots,k]$中的元素均小于x且$A[k+1,\ldots,j]$中的元素均大于（等于）x

1. $low \leftarrow i$; $high \leftarrow j$;
2. While($low < high$) Do
3. swap($A[low]$, $A[high]$);
4. While($A[low] < x$) Do
5. $low \leftarrow low+1$;
6. While($A[high] >= x$) Do
7. $high \leftarrow high-1$;
8. return($high$)

快速排序也通过交换元素调整数据次序，类似冒泡排序。

《算法导论》7.1节提供了另外一种思路。

Partition算法复杂性 $\Theta(n)$

Quick Sort (快速排序)

QuickSort(A,i,j)
Input: $A[i,\ldots,j]$, x
Output: 排序后的$A[i,\ldots,j]$

1. $x \leftarrow A[i]$; //以确定的策略选择x: *例如选择第一个元素*
2. k=Partition(A,i,j,x); //用x完成划分
3. QuickSort(A,i,k); //递归求解子问题
4. QuickSort($A,k+1,j$);

不需合并操作

Partition(A,i,j,x)

1. $low \leftarrow i$; $high \leftarrow j$;
2. While($low < high$) Do
3. swap($A[low]$, $A[high]$);
4. While($A[low] < x$) Do
5. $low \leftarrow low+1$;
6. While($A[high] >= x$) Do
7. $high \leftarrow high-1$;
8. return($high$)

Quick Sort (快速排序)

稍微改进（不要求掌握）：划分元素x划分一次后，就不会再参与比较了。

当用数列中一个元素作为划分元素时，使得划分位置k为划分元素，*A*[*i*,…,*k-1*]中的元素均小于*x*且*A*[*k*+1,…,*j*]中的元素均大于等于*x*，这样递归时，只需递归[i, k-1]和[k+1, j]即可。

QuickSort(*A*,*i*,*j*)

Input: *A*[*i*,…,*j*], *x*

Output: 排序后的*A*[*i*,…,*j*]

```
x←A[i];                          //以确定的策略选择x：例如第一个元素
k=Partition(A,i,j,x);            //用x完成划分
QuickSort(A,i,k-1);              //递归求解子问题
QuickSort(A,k+1,j);
```

Partition(*A*,*i*,*j*,*x*)

```
low←i+1 ;  high ←j;
While( low< high ) Do
        swap(A[low], A[high]);
        While( A[high] >=x & low <= high)  Do
                high←high-1;
        While( A[low] < x &  low < high)  Do
                low←low+1;
Swap(A[i], A[high]); //交换位于划分位置的元素和划分元素（第一个元素）
 return(high)
```

Quick Sort (快速排序)

7 4 8 6 5 2 1 3 7 10 $=A[i,\ldots,j]$

7 4 8 6 5 2 1 3 7 10 $=A[i,\ldots,j]$

划分标准 $A[i]=7$

7 4 3 6 5 2 1 8 7 10 $=A[i,\ldots,j]$

7 4 3 6 5 2 1 8 7 10 $=A[i,\ldots,j]$

1 4 3 6 5 2 7 8 7 10 $=A[i,\ldots,j]$

交换划分元素（第一个）和划分位置的元素

1 4 3 6 5 2 $=A[i,\ldots,k-1]$ 8 7 10 $=A[k+1,\ldots,j]$

Partition(A,i,j,x)

1. $low \leftarrow i+1$；$high \leftarrow j$;
2. While($low < high$) Do
3. swap($A[low]$, $A[high]$);
4. While($A[high] >= x$ & $low <= high$) Do
5. $high \leftarrow high-1$;
6. While($A[low] < x$ & $low < high$) Do
7. $low \leftarrow low+1$;
8. ~~Swap(A[i], A[high]);~~
9. return($high$)

稍微改进:
当用数列中一个元素作为划分元素时，使得划分位置k为划分元素，$A[i,\ldots,k-1]$中的元素均小于x且$A[k+1,\ldots,j]$中的元素均大于等于x

Quick Sort (快速排序)

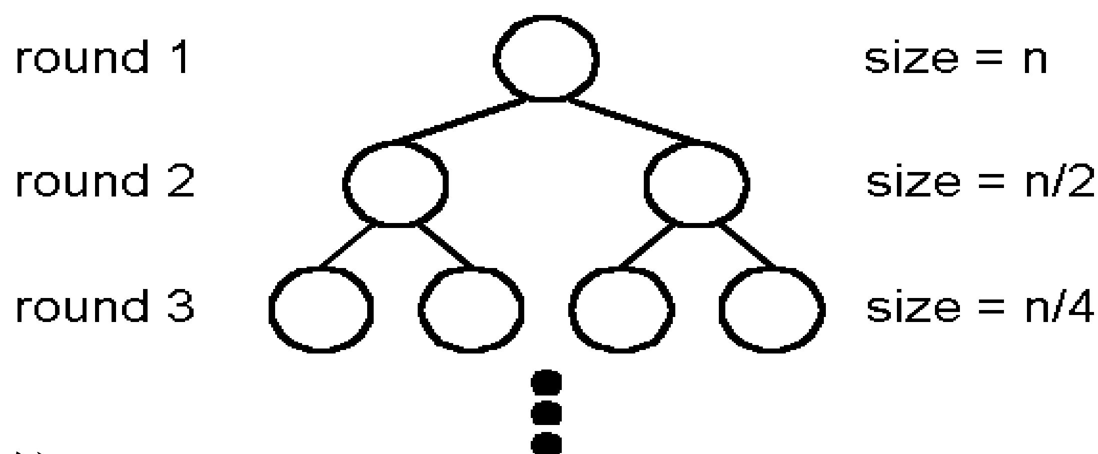
- 算法复杂性分析
  - 运行时间依赖于划分是否平衡。
    - 如果划分平衡，性能接近于归并排序；
    - 如果划分不平衡，性能接近于插入排序。

Partition（Divide）算法复杂性 Θ(n)
Combine没有时间消耗

最好情况 ：$\Theta(n\ log\ n)$

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

数组被分为大致相等的两个部分.



需要$log_2 n$ 轮.
每轮需要$\Theta(n)$次比较

Quick Sort (快速排序)

- **算法复杂性分析**

最坏情况 ： $\Theta(n^2)$

对于分治算法，子问题划分是否均衡至关重要。

每一轮用最大或者最小元素划分

$$T(n) = T(n-1) + \Theta(n)$$

用递归树方法求解：

$T(n) = cn + c(n\text{-}1)+\ldots+c = cn(n\text{-}1)/2 = \Theta(n^2)$

Quick Sort (快速排序)

- 算法复杂性分析

平均情况：$\Theta(n \log n)$

s　　n-s

包括分割点

划分点s可能是任意一个点

$$T(n) = \underset{1\le s\le n}{\mathrm{Avg}}(T(s)+T(n-s))+cn$$

$$= \frac{1}{n}\sum_{s=1}^{n}(T(s)+T(n-s))+cn$$

$$= \frac{1}{n}(T(1)+T(n-1)+T(2)+T(n-2)+\cdots+T(n)+T(0))+cn,\ T(0)=0$$

$$= \frac{1}{n}(2T(1)+2T(2)+\cdots+2T(n-1)+T(n))+cn$$

Quick Sort (快速排序)

- 算法复杂性分析

$$(n-1)T(n) = 2T(1)+2T(2)+\cdots+2T(n-1) + cn^2\cdots\cdots(1)$$

$$(n-2)T(n-1)=2T(1)+2T(2)+\cdots+2T(n-2)+c(n-1)^2\cdots(2)$$

$$(1) - (2)$$

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

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

$$\frac{T(n)}{n} = \frac{T(n-1)}{n-1} + c(\frac{1}{n} + \frac{1}{n-1})$$

$$=c(\frac{1}{n} + \frac{1}{n-1})+c(\frac{1}{n-1} + \frac{1}{n-2})+\cdots+c(\frac{1}{2}+1)+T(1),\ T(1) = 0$$

$$=c(\frac{1}{n} + \frac{1}{n-1}+\ldots+\frac{1}{2})+c(\frac{1}{n-1} + \frac{1}{n-2}+\ldots+1)$$

Quick Sort (快速排序)

- 算法复杂性分析
  - 调和级数

$$H_n = \sum_{k=0}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = O(\log n)$$

$$\frac{T(n)}{n} = c(H_n - 1) + cH_{n-1}$$

$$= c(2H_n - \frac{1}{n} - 1)$$

$$\Rightarrow T(n) = 2\, c\, n\, H_n - c(n+1)$$

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

Quick Sort (快速排序)

通过随机选取划分元素来实现每个元素是等概率的作为划分元素，从而实现所有划分是等概率的假设。

- 随机快速排序

```
RandomizedQuickSort(A,i,j) //随机选择划分元素
Input:    A[i,…,j], x
Output: 排序后的A[i,…,j]
    temp ←rand(i,j);                    //产生i，j之间的随机数
    x←A[temp];                          //以确定的策略选择x
    k=Partition(A,i,j,x);               //用x完成划分
    RandomizedQuickSort(A,i,k);              //递归求解子问题
    RandomizedQuickSort(A,k+1,j);

Partition(A,i,j,x)
    low←i ; high ←j;
    While( low< high ) Do
        swap(A[low], A[high]);
        While( A[low] < x ) Do
            low←low+1;
        While( A[high] >= x ) Do
            high←high-1;
    return(high)
```

Quick Sort (快速排序)

稍微改进后的版本（不要求掌握）：
划分元素x划分一次后，就不会再参与比较了

- 随机快速排序

RandomizedQuickSort(A,i,j)
Input: $A[i,\ldots,j]$, x
Output: 排序后的$A[i,\ldots,j]$

1. $temp \leftarrow$ rand(i,j); //产生i，j之间的随机数
2. $x \leftarrow A[temp]$; //以确定的策略选择x
3. Swap(A[i], A[temp]);
4. k=Partition(A,i,j,x); //用x完成划分
5. RandomizedQuickSort(A,i,k-1); //递归求解子问题
6. RandomizedQuickSort(A,k+1$,j$);

Partition(A,i,j,x)

1. $low \leftarrow i+1$; $high \leftarrow j$;
2. While($low < high$) Do
3. swap($A[low]$, $A[high]$);
4. While($A[high] >= x$ & $low <= high$) Do
5. $high \leftarrow high$-1;
6. While($A[low] < x$ & $low < high$) Do
7. $low \leftarrow low$+1;
8. Swap(A[i], A[high]);
9. return($high$)

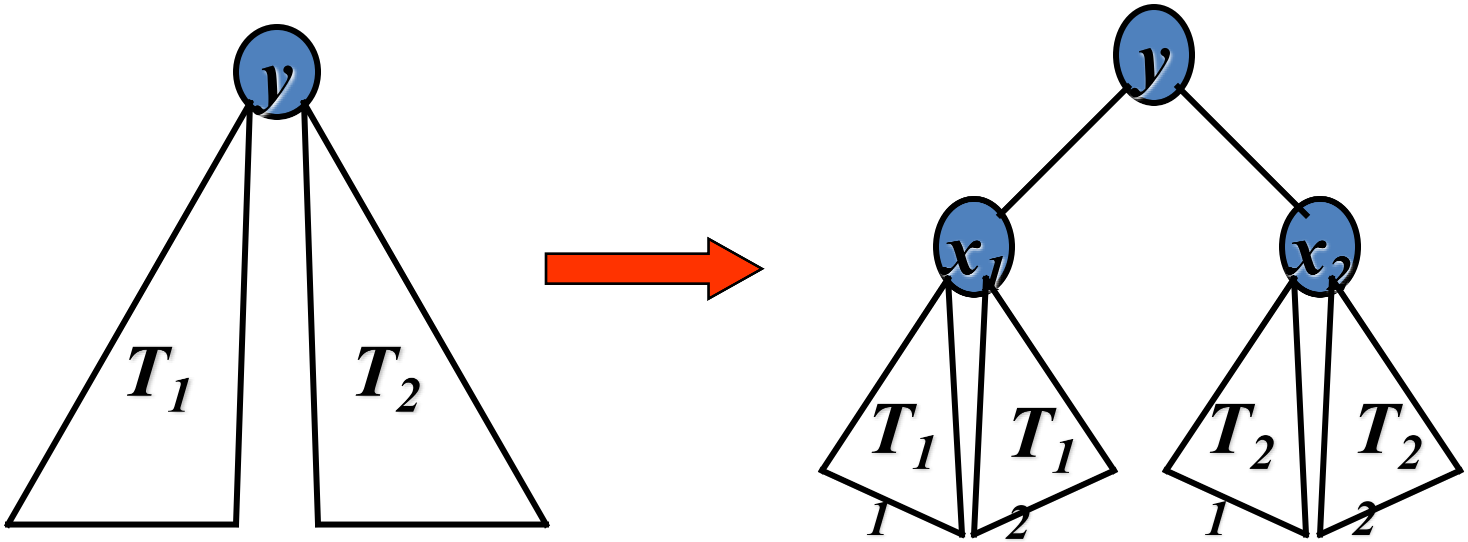
Quick Sort (快速排序)

- 随机快速排序
  - 平均（期望）运行时间
    - $S_{(i)}$表示S中第i小的元素

      例如, $S_{(1)}$和$S_{(n)}$分别是最小和最大元素
    - 随机变量X_{ij}定义如下:

      X_{ij}: $S_{(i)}$和$S_{(j)}$在运行中被比较的次数, 无比较为0
    - 算法的比较次数为 $\sum_{i=1}^{n}\sum_{j>i}X_{ij}$
    - 算法的平均复杂性为 $E[\sum_{i=1}^{n}\sum_{j>i}X_{ij}]=\sum_{i=1}^{n}\sum_{j>i}E[X_{ij}]$

- 我们可以用树表示算法的计算过程
  - 根表示当前的划分元素(主元)
  - 两个子树代表 被主元划分后的两个集合



- 我们可以观察到如下事实:
  - 一个子树的根必须与其子树的所有节点比较
  - 不同子树中的节点不可能比较
  - 任意两个节点至多比较一次：根（划分元素）划分一次后，就不会再参与比较了。

采用改进后的版本

Quick Sort (快速排序)

- 随机快速排序
  - 平均（期望）运行时间

•计算$E[X_{ij}]$

- 设p_{ij}为$S_{(i)}$和$S_{(j)}$在运行中比较的概率, 则

$$E[X_{ij}]=p_{ij}\times 1+(1-p_{ij})\times 0=p_{ij}$$

关键问题:求解p_{ij}

Quick Sort (快速排序)

- 随机快速排序
  - 平均（期望）运行时间
    - 当$S_{(i)}, S_{(i+1)}, \ldots, S_{(j)}$在同一子树时，$S_{(i)}$和$S_{(j)}$才可能比较
    - 只有$S_{(i)}$或$S_{(j)}$先于$S(i+1), \ldots, S(j-1)$被选为划分点时，$S_{(i)}$和$S_{(j)}$才可能比较
    - $S_{(i)}, S_{(i+1)}, \ldots, S_{(j)}$等可能地被选为划分点，所以$S_{(i)}$和$S_{(j)}$进行比较的概率是：$2/(j-i+1)$，即

$$p_{ij}=2/(j-i+1)$$

注意$S_{(i)}$是排好序的序列！

为什么不能选择比[i, j]更大的区间？

因为选了[i, j]区间以外的元素作为划分元素，$S_{(i)}$和$S_{(j)}$仍会分在一棵子树，仍有可能会进行比较。

只有当$S_{(i)}$和$S_{(j)}$分属于区间两端，无论选择区间内哪个元素作为划分点，此次划分后一定会分开，上述计算概率的方式才适用。

Quick Sort (快速排序)

- 随机快速排序
  - 平均（期望）运行时间
- **现在我们有**

$$\sum_{i=1}^{n}\sum_{j>i}E[X_{ij}]=\sum_{i=1}^{n-1}\sum_{j>i}p_{ij}=\sum_{i=1}^{n-1}\sum_{j>i}\frac{2}{j-i+1}$$

调和级数：$H_n=\sum_{k=0}^{n}\frac{1}{k}=1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}=O(\log n)$

$$\le\sum_{i=1}^{n-1}\sum_{k=1}^{n-i}\frac{2}{k}\le 2\sum_{i=1}^{n-1}\sum_{k=1}^{n}\frac{1}{k}=2nH_n=O(n\log n)$$

定理．随机快速排序算法的期望时间复杂性为O(*n*log*n*)

本讲内容

3.1 Divide-and-Conquer(分治算法)
3.2 Merge Sort（归并排序）
3.3 Quick Sort（快速排序）
3.4 排序问题的下界

排序问题的下界

- 问题的下界是解决该问题的所有算法中所需要的最小时间复杂性。
  - 最坏情况下界
  - 平均情况下界
- 如果一个问题的最高下界是 $\Omega(n \log n)$，而当前最好算法的时间复杂性是 $O(n^2)$.
  - 我们可以寻找一个更高的下界.
  - 我们可以设计更好的算法.
  - 下界和算法都是可以(理论上)改进的.
- 如果一个问题的下界是$\Omega(n \log n)$ 且算法的时间复杂性是 $O(n \log n)$, 那么这个算法是最优的。

排序问题的下界

- 最坏情况下界

比如对（1，2，3）3个元素有6种排列

a_1	a_2	a_3
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

排序问题的下界

- 最坏情况下界
  - 如果用插入排序
    - 输入：(2, 3, 1)

      (1) $a_1 : a_2$

      (2) $a_2 : a_3$, $a_2 \leftrightarrow a_3$

      (3) $a_1 : a_2$, $a_1 \leftrightarrow a_2$
    - 输入：(2, 1, 3)

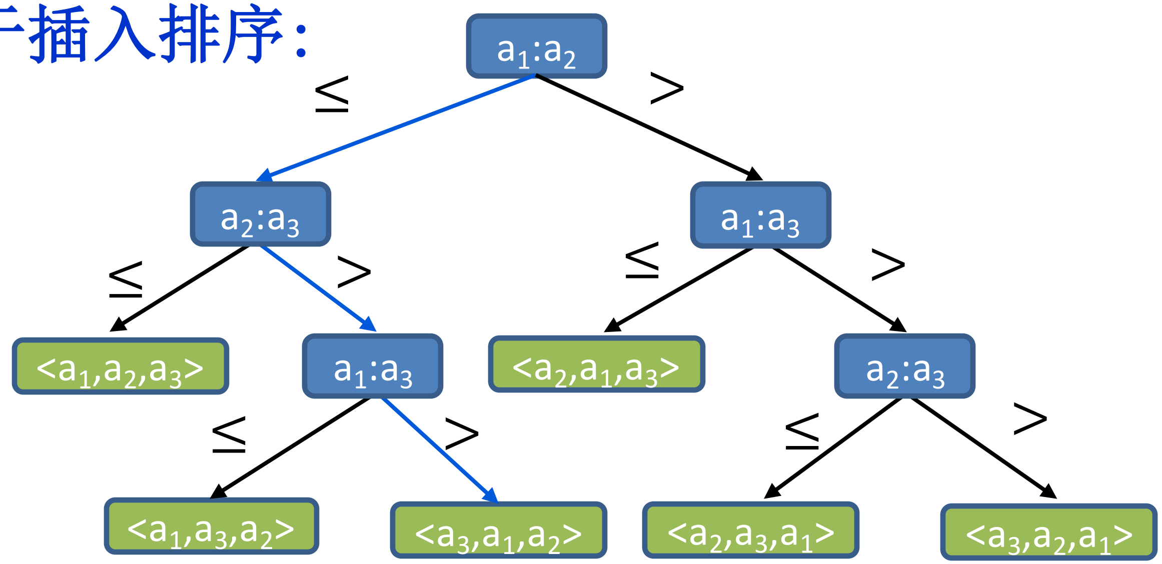
      (1) $a_1 : a_2$, $a_1 \leftrightarrow a_2$

      (2) $a_2 : a_3$

```
Insertion-sort(A)
Input:  A[1,......,n]=n个数
output: A[1,......,n]=n个sorted数
FOR j=2  To n  Do
    key←A[j];
    i←j-1
    WHILE i>0 AND A[i]>key  Do
        A[i+1]←A[i];
        i←i-1;
    A[i+1]←key;
```

排序问题的下界

- 最坏情况下界
  - 将排序过程抽象为一颗决策树，排序算法的执行对应于一条从树根到叶节点的路径。
  - 每个叶节点代表一种可能的排序结果序列。
  - 每个内部节点代表一次比较
  - 对于插入排序：



排序问题的下界

- 最坏情况下界
  - 将一种排序算法的排序过程抽象为一颗决策树
  - 在决策树中，从根节点到任意一个叶节点之间的最长简单路径的长度，即该决策树的高度，是该排序算法中最坏情况下的比较次数

- 一种排序算法对应一棵二叉决策树，为了找到排序的下界，我们需要找到所有可能的二叉树的最小高度.
- 有n!种不同排列
  二叉决策树有n!个叶子结点.
- 平衡树高度最小
  $\lceil \log(n!) \rceil = \Omega(n \log n)$
  排序的下界是: $\Omega(n \log n)$

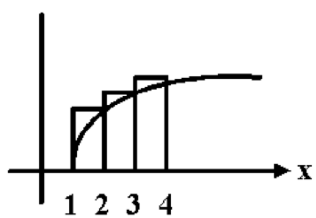
平衡二叉树:
它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。

排序问题的下界

- 最坏情况下界
  - ❖ 将一种排序算法的排序过程抽象为一颗决策树
  - ❖ 在决策树中，从根节点到任意一个叶节点之间的最长简单路径的长度，即该决策树的高度，是该排序算法中最坏情况下的比较次数
  - ❖ 方法一：

如果f(k)单调递增，则 $\int_{m-1}^{n} f(x)dx \le \sum_{k=m}^{n} f(k) \le \int_{m}^{n+1} f(x)dx$

$$
\begin{aligned}
\log(n!) &= \log(n(n-1)\cdots 1) \\
&= \log 2 + \log 3 + \cdots + \log n \\
&> \int_1^n \log x dx \\
&= \log e \int_1^n \ln x dx \\
&= \log e[x \ln x - x]_1^n \\
&= \log e(n \ln n - n + 1) \\
&= n \log n - n \log e + 1.44 \\
&\square\ n \log n - 1.44n \\
&= \Omega(n \log n)
\end{aligned}
$$



注意：log以2为底

排序问题的下界

- 最坏情况下界
  - 将一种排序算法的排序过程抽象为一颗决策树
  - 在决策树中，从根节点到任意一个叶节点之间的最长简单路径的长度，即该决策树的高度，是该排序算法中最坏情况下的比较次数
  - 方法2：Stirling近似公式

斯特林近似公式，书中公式3.18

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

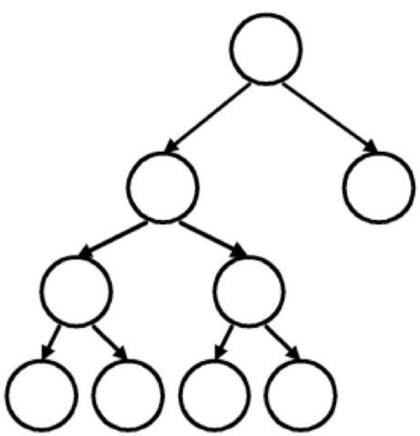
$$\log(n!) \approx \log\left(\sqrt{2\pi}\right) + \frac{1}{2}\log n + n\log\left(\frac{n}{e}\right) \approx n\log n \approx \Omega(n\log n)$$

n	n!	S_n
1	1	0.922
2	2	1.919
3	6	5.825
4	24	23.447
5	120	118.02
6	720	707.39
10	3,628,800	3,598,600
20	2.433×10^{18}	2.423×10^{18}
100	9.333×10^{157}	9.328×10^{157}

斯特林公式近似值

排序问题的下界

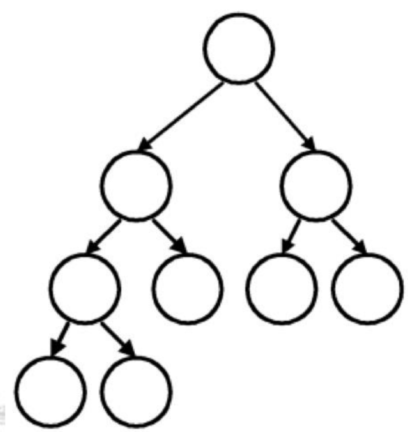
- 平均情况下界
  - ❖ 仍利用决策树
  - ❖ 排序算法的平均复杂性用从根结点到每个叶子结点的路径长度的（平均）总长度描述（共有n!个叶子节点）
  - ❖ 当树平衡时这个值最小
    - 所有叶子结点的深度是 d 或 d–1 （d为树高度）



不平衡的情况下

总长度

$= 4 \cdot 3 + 1 = 13$



平衡的情况下

总长度

$= 2 \cdot 3 + 3 \cdot 2 = 12$

排序问题的下界

- 平均情况下界
  - 当树平衡时这个值最小
  - 有c个叶子结点的平衡二叉树的深度 $d = \lceil \log c \rceil$
  - 叶子结点仅出现在d或d–1层.
    - x_1个结点在d–1层
    - x_2个结点在d层

$$X_1 + X_2 = c$$

$$X_1 + \frac{X_2}{2} = 2^{d-1}$$

d-1层节点数量

$$\Rightarrow \quad X_1 = 2^d - c$$

$$X_2 = 2c - 2^d$$

排序问题的下界

- **平均情况下界**

总长度

$$
\begin{aligned}
M &= x_1(d-1) + x_2 d \\
&= (2^d - c)(d-1) + 2(c - 2^{d-1})d \\
&= c(d-1) + 2(c - 2^{d-1}),\ \ d - 1 = \lfloor \log c \rfloor \\
&= c\lfloor \log c \rfloor + 2(c - 2^{\lfloor \log c \rfloor})
\end{aligned}
$$

$c = n!$

$$
\begin{aligned}
M &= n!\lfloor \log n! \rfloor + 2(n! - 2^{\lfloor \log n! \rfloor}) \\
M/n! &\approx \lfloor \log n! \rfloor + 2 \quad \boxed{\text{平均总长度}} \\
&= \Omega(n \log n)
\end{aligned}
$$

<u>平均情况下排序的下界是: $\Omega(n \log n)$</u>

思考题 1

给定长度为n的单调不减数列$a_0, \cdots, a_n$和一个数k，在O(logn)的时间复杂度内找到满足$a_i \geq k$ 条件的最小i，写出伪代码

二分搜索（binary search），也称折半搜索（half-interval search）、对数搜索（logarithmic search），是一种在**有序数组**中查找某一特定元素的搜索算法。

搜索过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜索过程结束；如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。如果在某一步骤数组为空，则代表找不到。这种搜索算法每一次比较都使搜索范围缩小一半。

Lowbound(A,k)

```
lowbound=0;
upbound=A.length-1;
While upbound-lowbound>0 Do
        mid=(upbound+lowbound)/2;
        IF A[mid]>=k THEN upbound=mid;
        ELSE lowbound=mid+1;
print upbound;
```

标准二分查找:

给定一个单调不减的序列 A，查找其中含有值为key的元素。

非递归版本：

BinarySearch(*A*, *key*)

```
low←0, high ←A.length-1;
While( low< high ) Do
        mid = (low+high) / 2;
        If A[mid]  == key   Then
              return  mid;
        Else If A[mid] > key   Then
              high ← mid - 1;
        Else Then
              low ← mid + 1;
Return -1；//当low>high时表示查找空间为空，没找到
```

递归版本怎么写?

递归版本：

BinarySearch(*key*, *low*, *high*)

```
If high < low Then
     return -1;
mid = (low+high) / 2;
 If A[mid]  == key   Then
       return  mid;
Else If A[mid] > key   Then
       return BinarySearch(key, low, mid-1);
Else   Then
  return BinarySearch(key, mid+1, high);
```

最坏情况:搜索到区间只剩下一个元素

$$T(n)=T(n/2)+O(1)$$

主定理

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

思考题2　平面最近点对问题

给定平面上的n个点，用O(nlogn)的时间复杂度求出距离最近的两点距离，写出伪代码。

例如：Input: n=5, A={(0,2), (6,67), (43,71), (39,107), (189,140)}

Output: 36.22

暴力求解：
依次计算所有的点对距离，然后取最小值。
复杂度：$O(n^2)$

分治算法?

思考题2　　平面最近点对问题

给定平面上的n个点，用O(nlogn)的时间复杂度求出距离最近的两点距离，写出伪代码。

例如：Input: n=5, A={(0,2), (6,67), (43,71), (39,107), (189,140)}

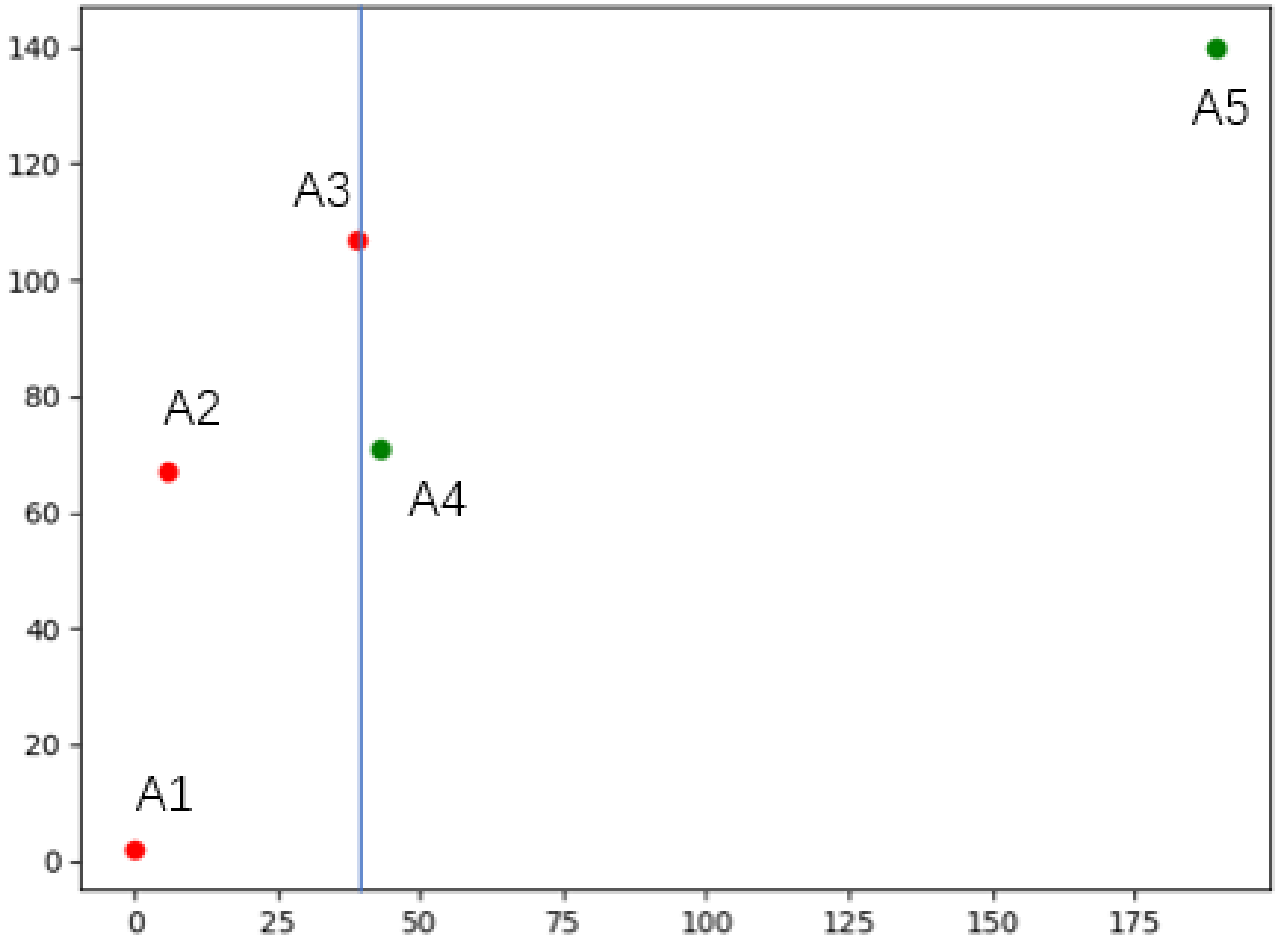
Output: 36.22

分治算法：
将点分成两个集合，分别求出两个集合中的最近点对距离，然后取最小值，这样对吗？
没有考虑两个集合中分别取一个点的最近点对。

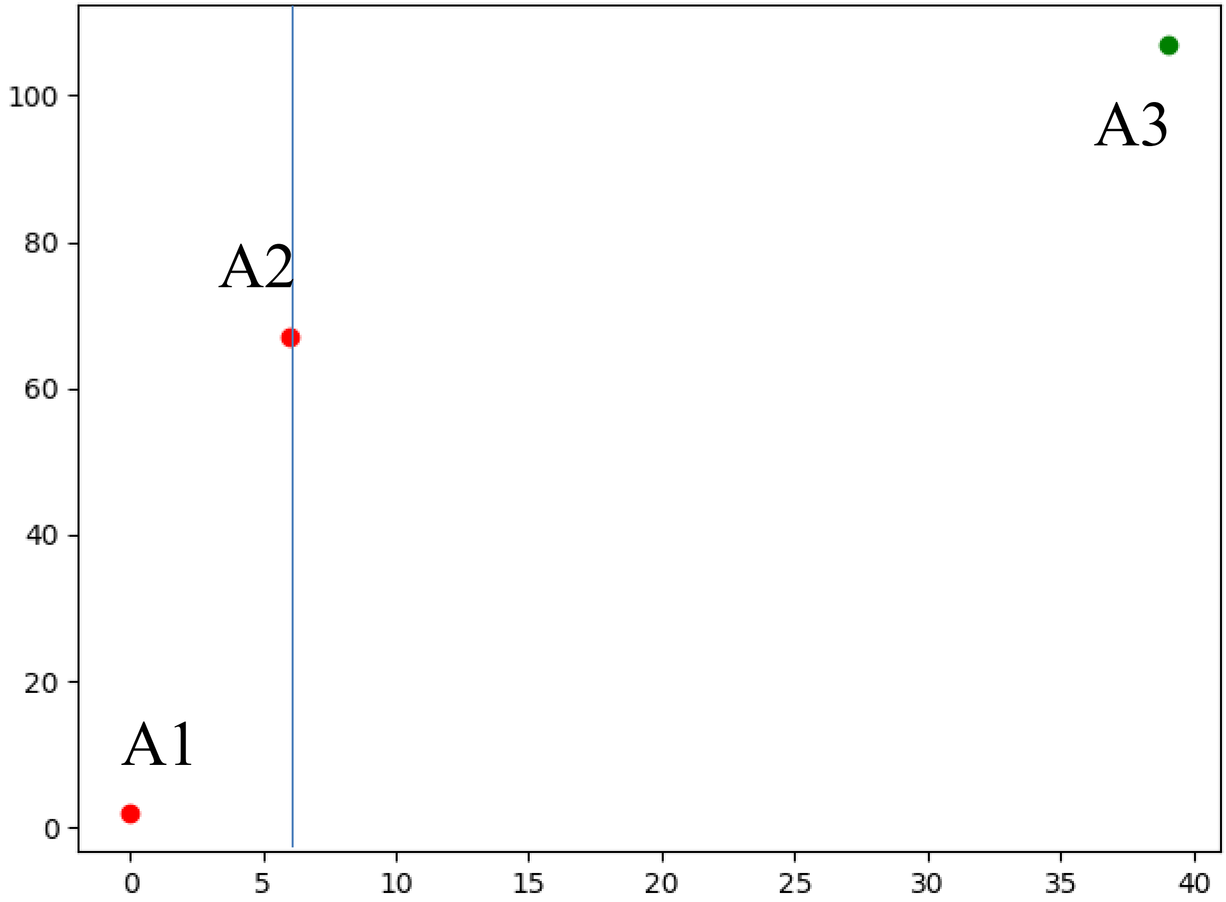
Closest_pair(A)
1. Divide A into two sets: S_1 and S_2
2. D = min(Closest_pair(S_1), Closest_pair(S_2))
3. Calculate the closest pair, in which one point from S_1 and the other point from S_2

怎么分成两个集合？

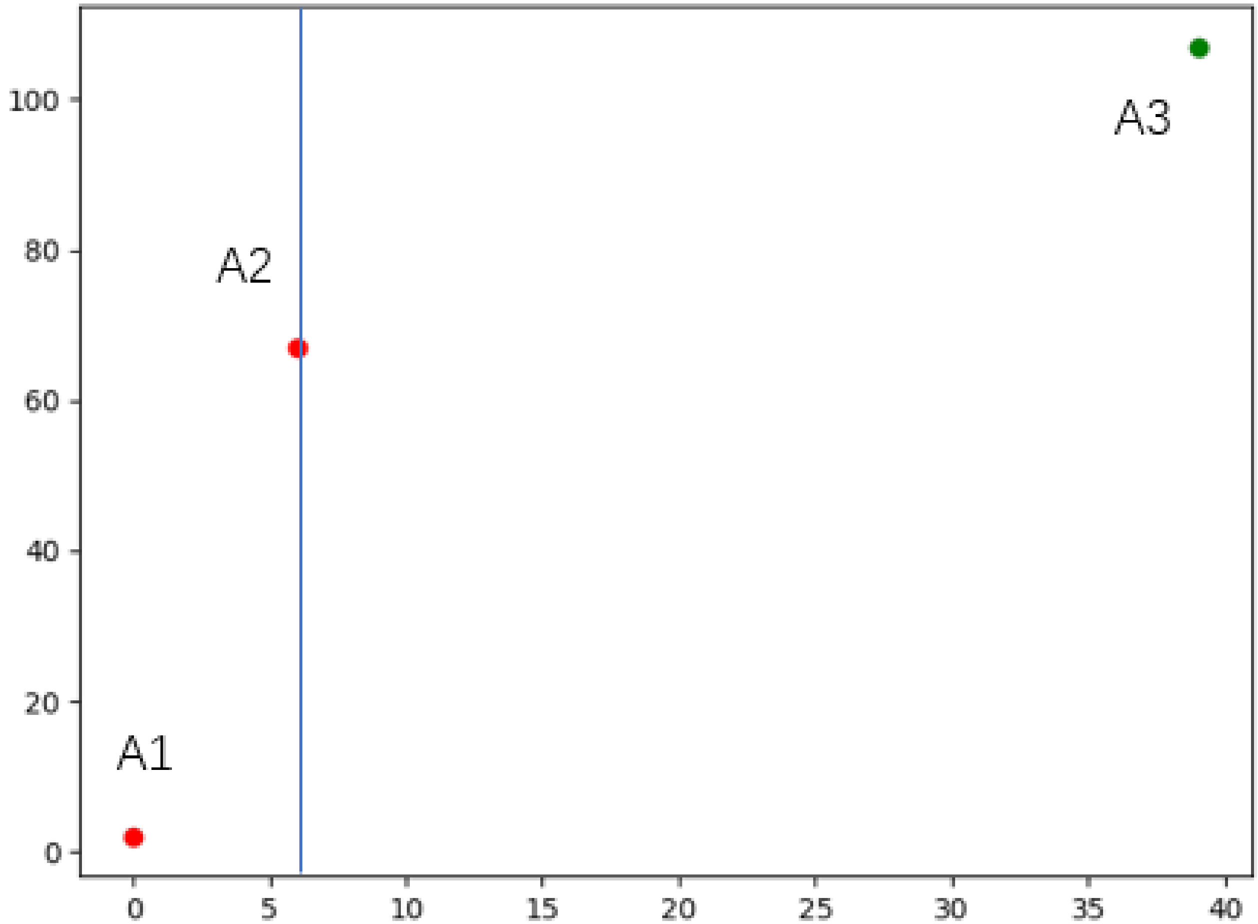
按照X坐标排序，根据x坐标中位数x_m分为左右两个部分



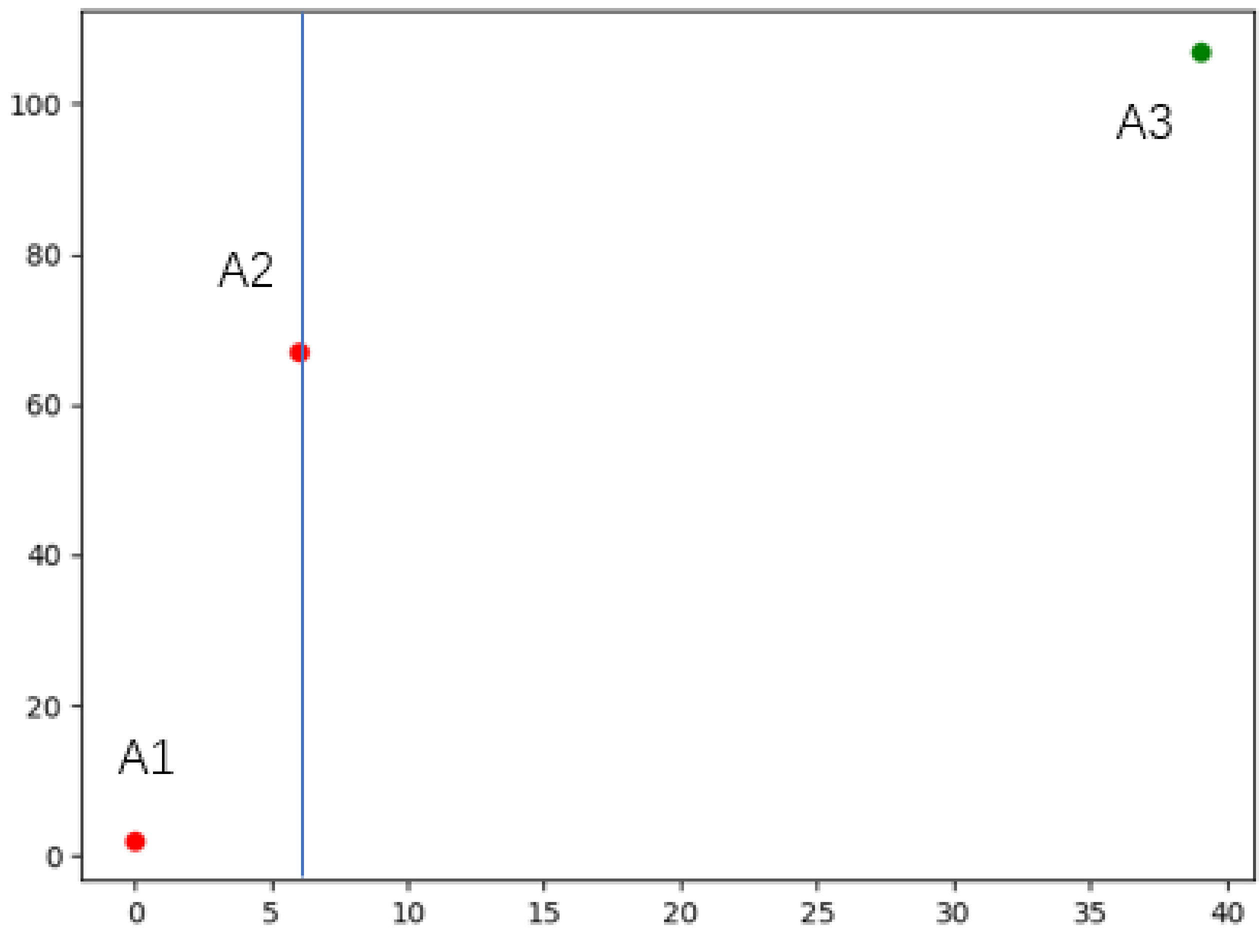
继续划分子问题，直到左右区域只剩下两个点



计算左区域的最小距离d_1



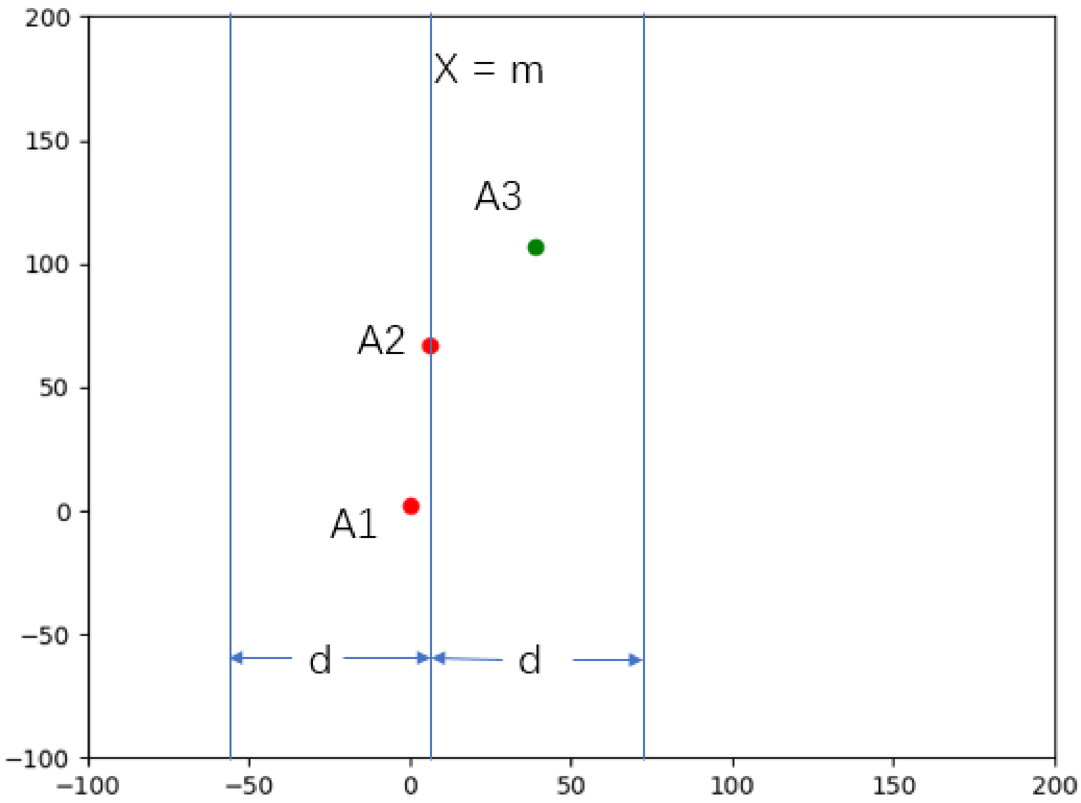
计算右区域的最小距离d_2，因此左右两个区域的最小距离$d = \min\{d_1, d_2\}$



Closest_pair(A)
1. Divide A into two sets: S_1 and S_2
2. D = min(Closest_pair(S_1), Closest_pair(S_2))
3. Calculate the closest pair, in which one point from S_1 and the other point from S_2

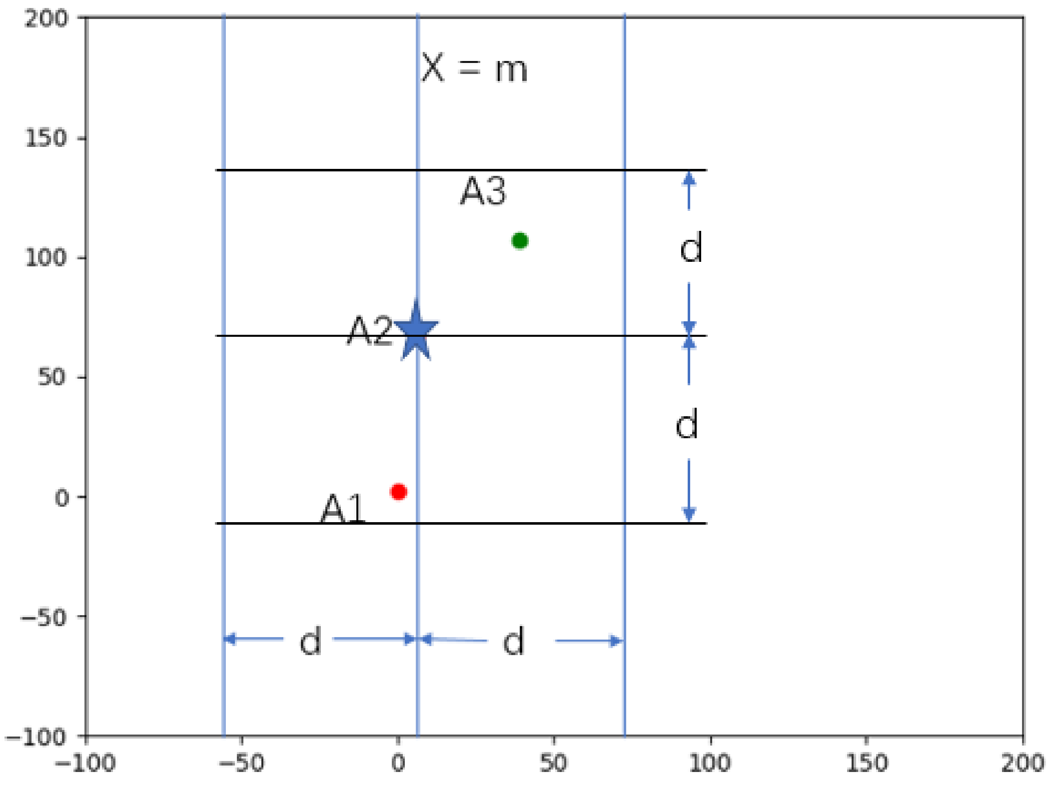
How?

根据y坐标值从小到大排序，然后从下往上寻找在 $x\in[x_m-d, x_m+d]$范围内的点. $d=\min(d_1, d_2)$



对于在 $x \in [m-d,m+d]$ 范围内的每一个点（例如 $A2(x_2,y_2)$），寻找在 $y \in [y_2-d,y_2+d]$ 范围内的点，并依次计算A2与 $y \in [y_2-d,y_2+d]$ 范围内的点的距离d’，若存在距离d’<d,则d = d’。

如果两个点的纵坐标的差的绝对值大于d，那么它们间的距离也一定大于d。



closest_pair(A)

```
If A.size<=1 return INF
sort_by_x(A)
m = A.size/2;  x= A[m].first     // 横坐标中位数
d = min(closest_pair(A[0..m]),closest_pair(A[m+1..n]))
sort_by_y(A)
B = []
For i = 1 to n
        If |A[i].first-x|>=d continue; //过滤x∉[x_m-d,x_m+d]
        for j=0 to B.size
                dx = A[i].first – B[B.size-j].first
                dy = A[i].second – B[B.size-j].second
                If dy >= d break
                d = min(d,sqrt(dx*dx+dy*dy))
        B.push(A[i])
Return d
```

B：缓冲区，依次存储按照纵坐标排好序的在$x \in [x_m-d, x_m+d]$的且已经遍历过的点。

A和B中元素都按照纵坐标y从小到大排好序，且B中元素的y值都小于此时的A[i]，因此A[i]和B中元素，从栈顶到栈底的 y差值依次增大

closest_pair(A)

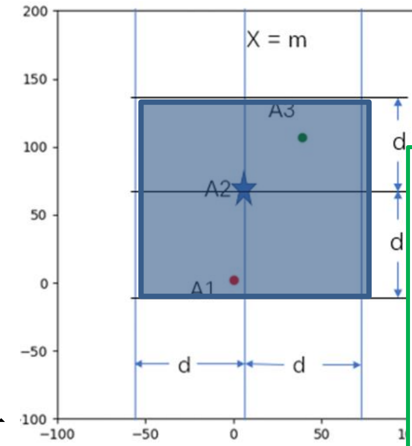
If A.size<=1 return INF

sort_by_x(A) //可以提前做好，那么只需要做一次

m = A.size/2; x= A[m].first   // 横坐标中位数

d = min(closest_pair(A[0..m]),closest_pair(A[m+1..n]))

sort_by_y(A) //可以提前做好，那么只需要做一次



因为要满足d是左右两个半区（子问题）的最小距离，所以这个阴影区内点数有限(常数个)，因此为常数时间复杂性，因此阴影框内的时间复杂性为O(n)。

如果把x和y的排序都提前排好，即放在递归程序前面，那么总的时间复杂性为O(nlogn)

时间复杂性？

```
B = []
For i = 1 to n
        If |A[i].first-x|>=d continue; //过滤x∉[xm-d,xm+d]
        for j=0 to B.size
                dx = A[i].first – B[B.size-j].first
                dy = A[i].second – B[B.size-j].second
                If dy >= d break
                d = min(d,sqrt(dx*dx+dy*dy))
        B.push(A[i])
Return d
```