

算法设计与分析

第三章 分治法

哈尔滨工业大学
李穆



有100枚硬币，其中1枚重量与众不同，是假币，略轻一些。如果用天平秤，如何找到这枚假币？



思路1：分成50组，每组2个硬币，把同组内的硬币放在天平上比较。

思路2：将硬币分成两组数量相同的硬币，放在天平两端，选择较轻的一组递归求解。

思路3：将硬币分成3组数量相同的硬币 (a、b、c)

$$\begin{cases} a, & \text{if } a < b; \\ b, & \text{if } a > b; \\ c, & \text{if } a = b. \end{cases}$$

Outlines



3.1 分治法

3.2 排序介绍

3.3 merge sort

3.4 quicksort

3.5 排序问题的下界



3.1 Divide-and-Conquer 技术

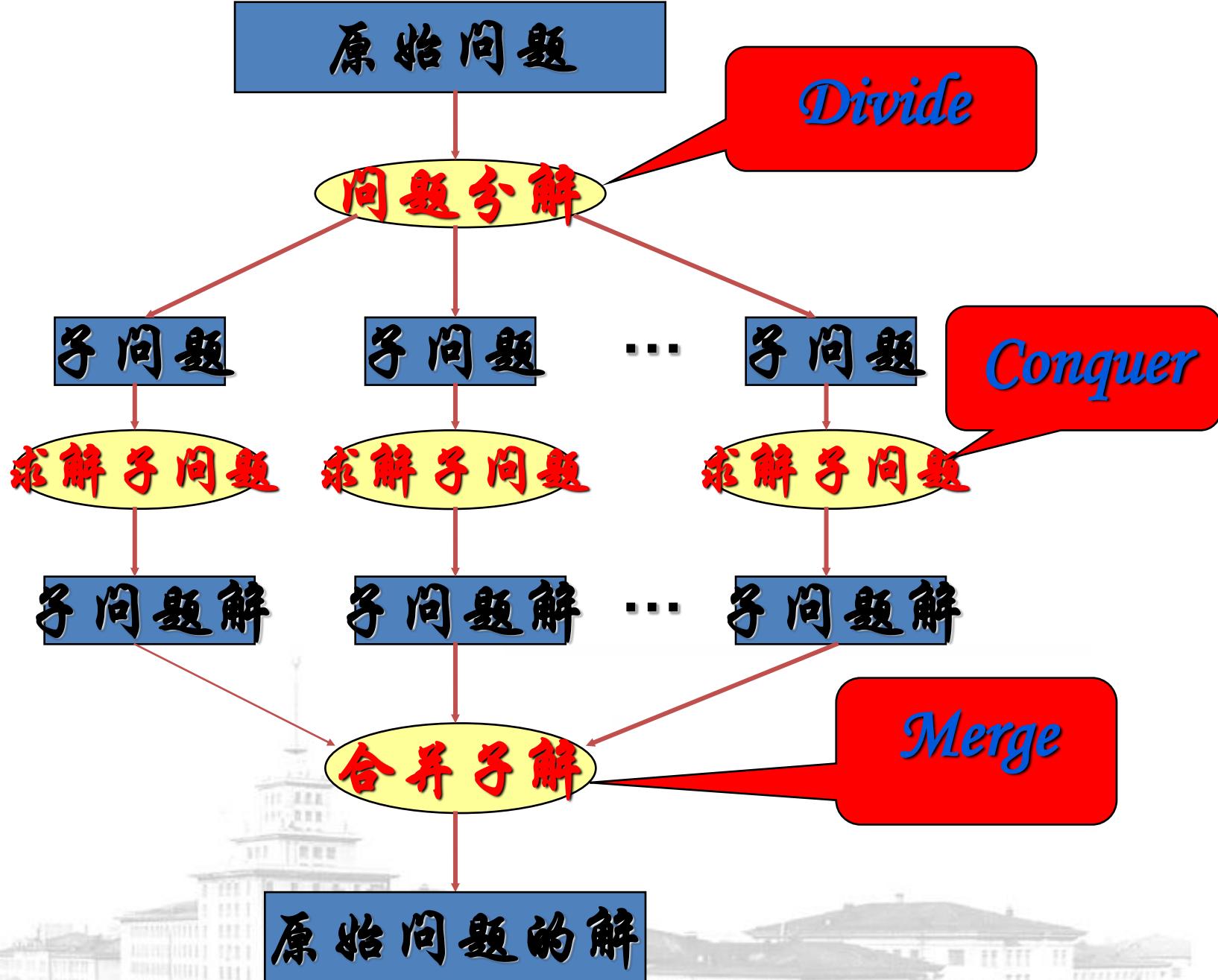


- *Divide-and-Conquer* 算法的设计
- *Divide-and-Conquer* 算法的分析



*Divide-and-Conquer*算法的设计

- 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(递归调用正设计的算法)
 - *Combine*: 合并子问题的解, 形成原始问题的解



Divide-and-Conquer 算法的分析

- 分析过程
 - 建立递归方程
 - 求解
- 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= $C(n)$



— 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



大数乘法

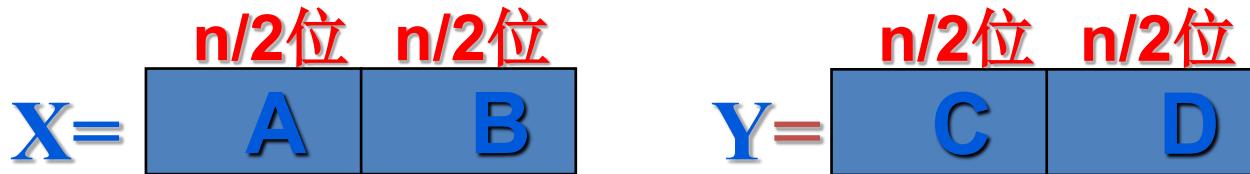


问题定义

输入：n位二进制整数X和Y
输出：X和Y的乘积



简单分治算法



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

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: $T(n)=O(n^2)$

改进分治算法



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

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， $((A-B)(D-C)+AC+BD)$ 左移 $n/2$ 位;
5. 计算 XY

算法复杂性:

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

算法的分析

- 建立递归方程

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

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

- 使用Master定理

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

例 求max与min问题

问题定义

输入： 数组 $A[1, \dots, n]$

输出： A 中的max和min

通常， 直接扫描需要 $2n-2$ 次比较操作
我们给出一个仅需 $\lceil 3n/2-2 \rceil$ 次比较操
作的算法。

基本思想



基于任务不同将问题划分为两个子问题

一个子问题求解 \max

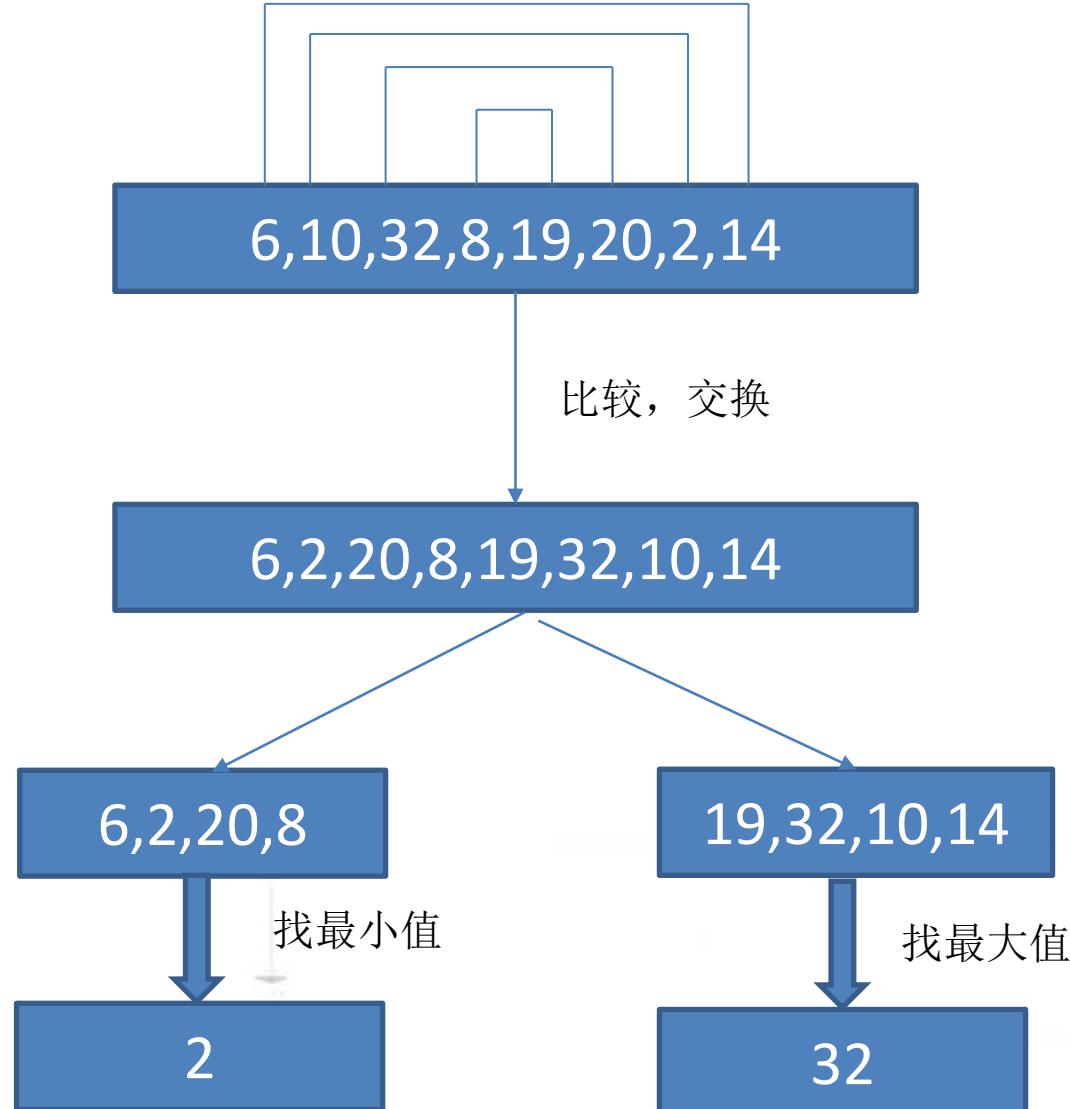
另一个子问题求解 \min

将 $A[i]$ 与 $A[n-i+1]$ 比较， $i=1,2,\dots,n/2$

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 $A[1,2,\dots,\lceil n/2 \rceil]$ 中

最大元素 $A[\lceil n/2 \rceil, \dots, n]$ 中



Max-min(A)

Input: 数组 $A[1, \dots, n]$

Output: 数组 $A[1, \dots, n]$ 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 $\max, \min;$

算法复杂性

$\lceil 3n/2 - 2 \rceil$ 次比较操作

$2 \uparrow 32$

6, 10, 32, 8, 19, 20, 2, 14

6↑
32

6, 10, 32, 8

6↑
10

6, 10

8↑
32

32, 8

2↑
20

19, 20, 2, 14

19↑
20

19, 20

2↑
14

2, 14

Max-min

Input: 数组 $A[1, \dots, n]$

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. $mid \leftarrow (low+high)/2$
6. $(x1,y1) \leftarrow \text{Max-min}(low,mid)$
7. $(x2,y2) \leftarrow \text{Max-min}(mid+1,high)$
8. $x \leftarrow \min\{x1,x2\}$
9. $y \leftarrow \max\{y1,y2\}$
10. return (x,y)



$$T(1)=0$$

$$T(2)=1$$

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

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

= ...

$$=2^{k-1}T(2)+2^{k-1}+2^{k-2}+\dots+2^2+2$$

$$n=2^k$$

$$=2^{k-1}+2^k-1$$

$$=n/2+n-1$$

$$=3n/2-1$$



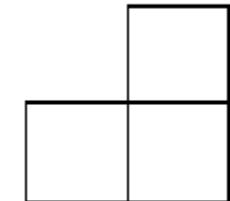
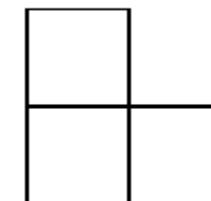
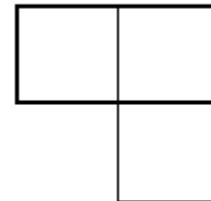
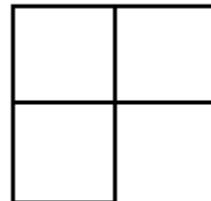
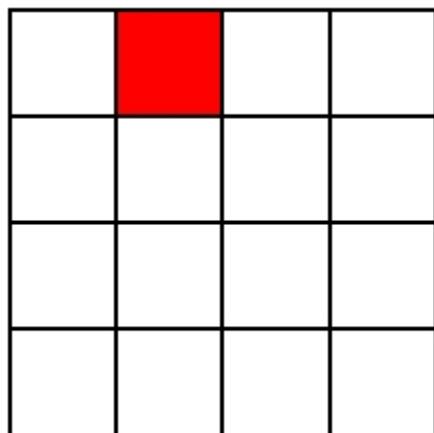
例：棋盘覆盖问题



棋盘覆盖问题 问题定义

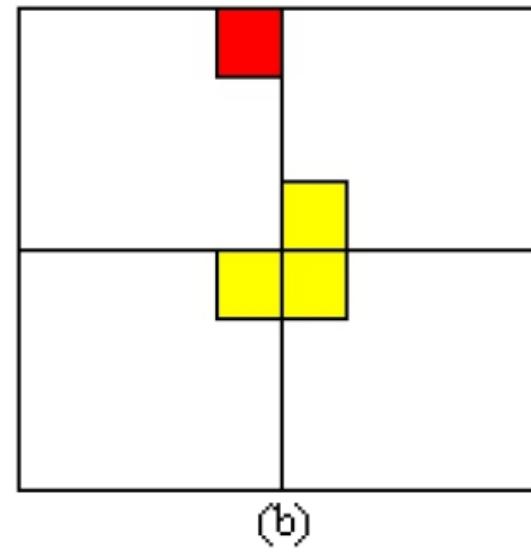
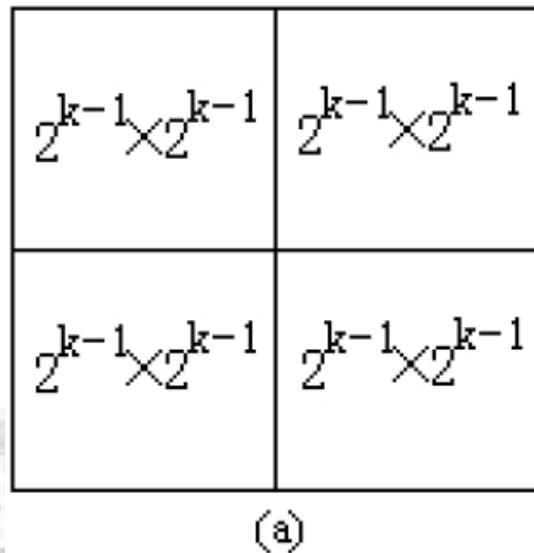
在一个 $2^k \times 2^k$ 个方格组成的棋盘中，有一个方格与其它的方格不同，称之为奇异块。

要求：若使用以下四种L型骨牌覆盖除这个奇异块的其它方格，覆盖过程中L型骨牌间不能有互相覆盖，设计算法求出覆盖方案。四个L型骨牌如下图：

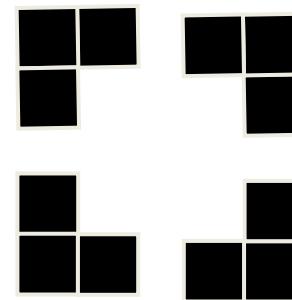
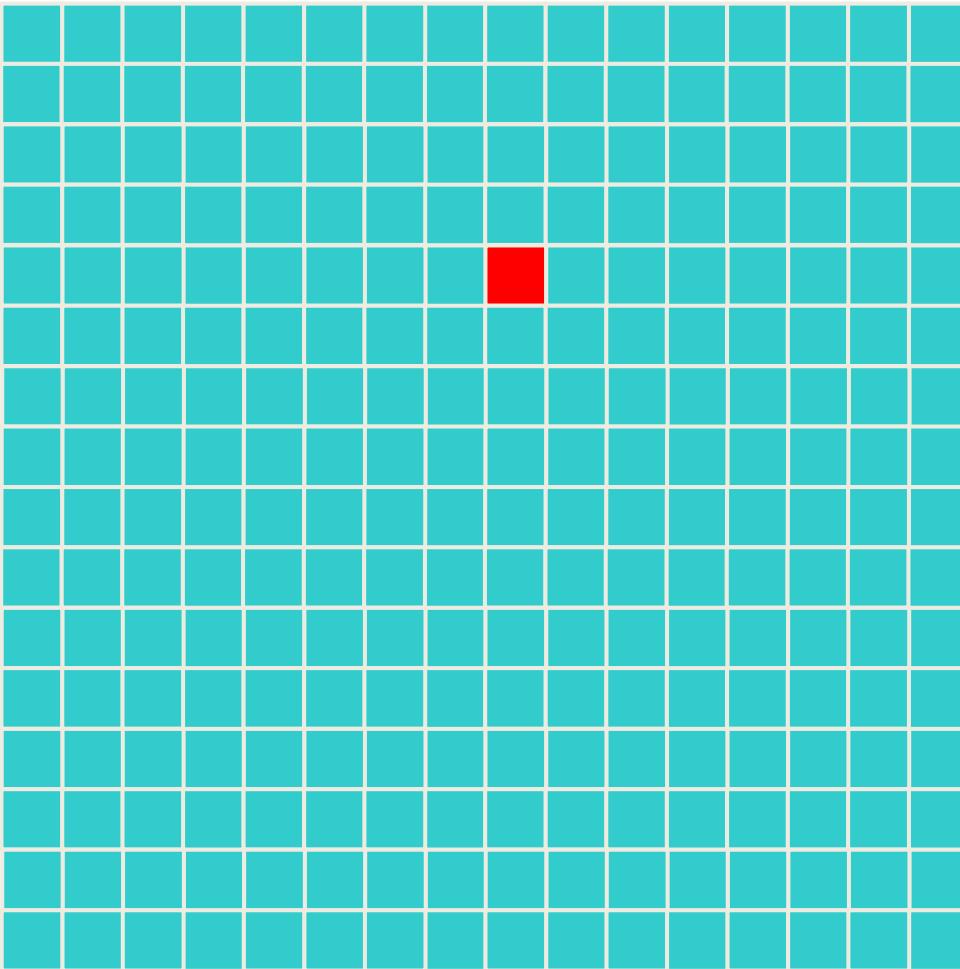


分治思想

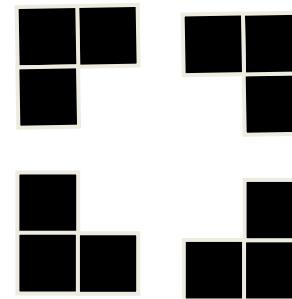
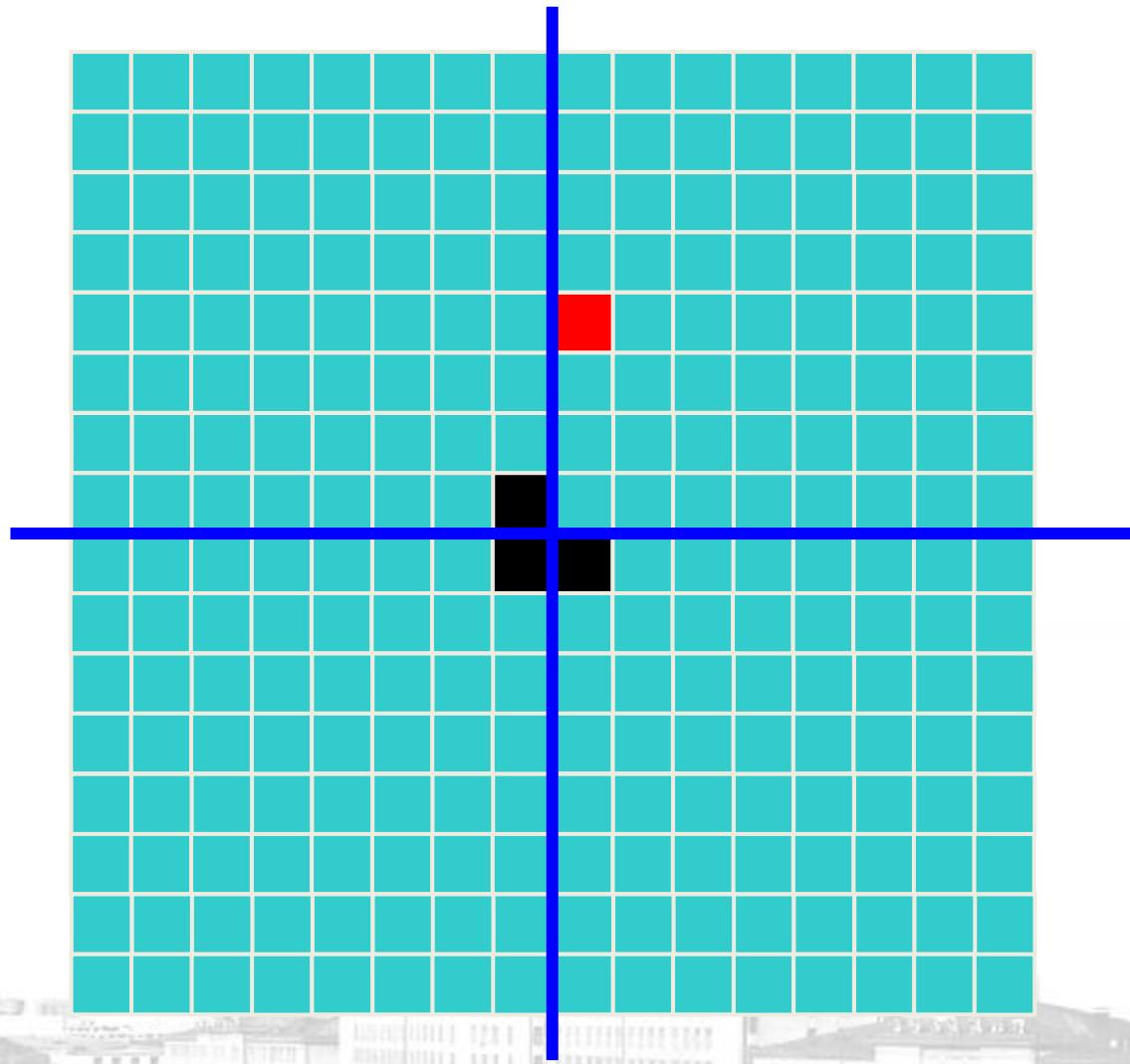
- 当 $k > 0$ 时，将 $2^k \times 2^k$ 的棋盘分成四块 $2^{(k-1)} \times 2^{(k-1)}$ 的子棋盘
特殊方格一定在其中的一个子棋盘中，其他相邻的三个子棋盘中没有特殊方格
- 使用一个合适的L型骨牌，覆盖三个没有特殊方格的子棋盘的相邻方格
我们就得到了4个 $2^{(k-1)} \times 2^{(k-1)}$ 的棋盘覆盖问题
- 继续递归处理4子棋盘，直到子棋盘中只有一个特殊方格为止



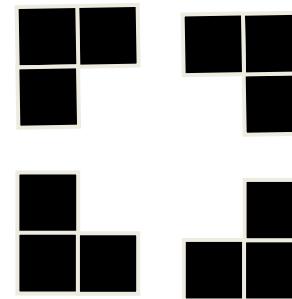
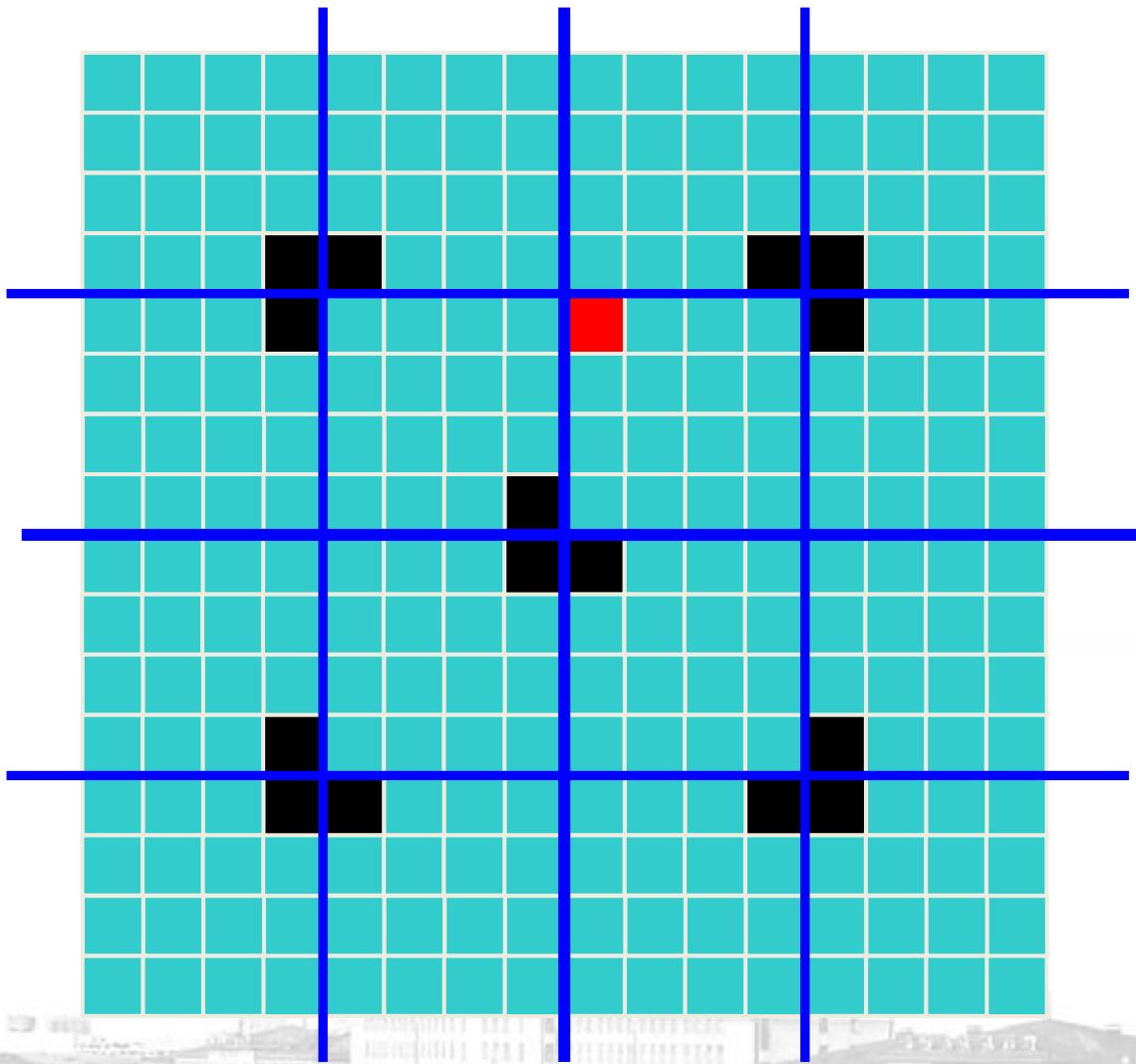
详细过程图解



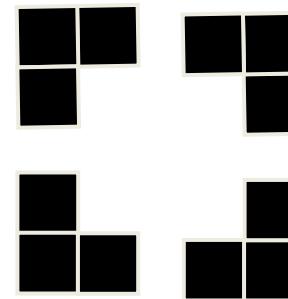
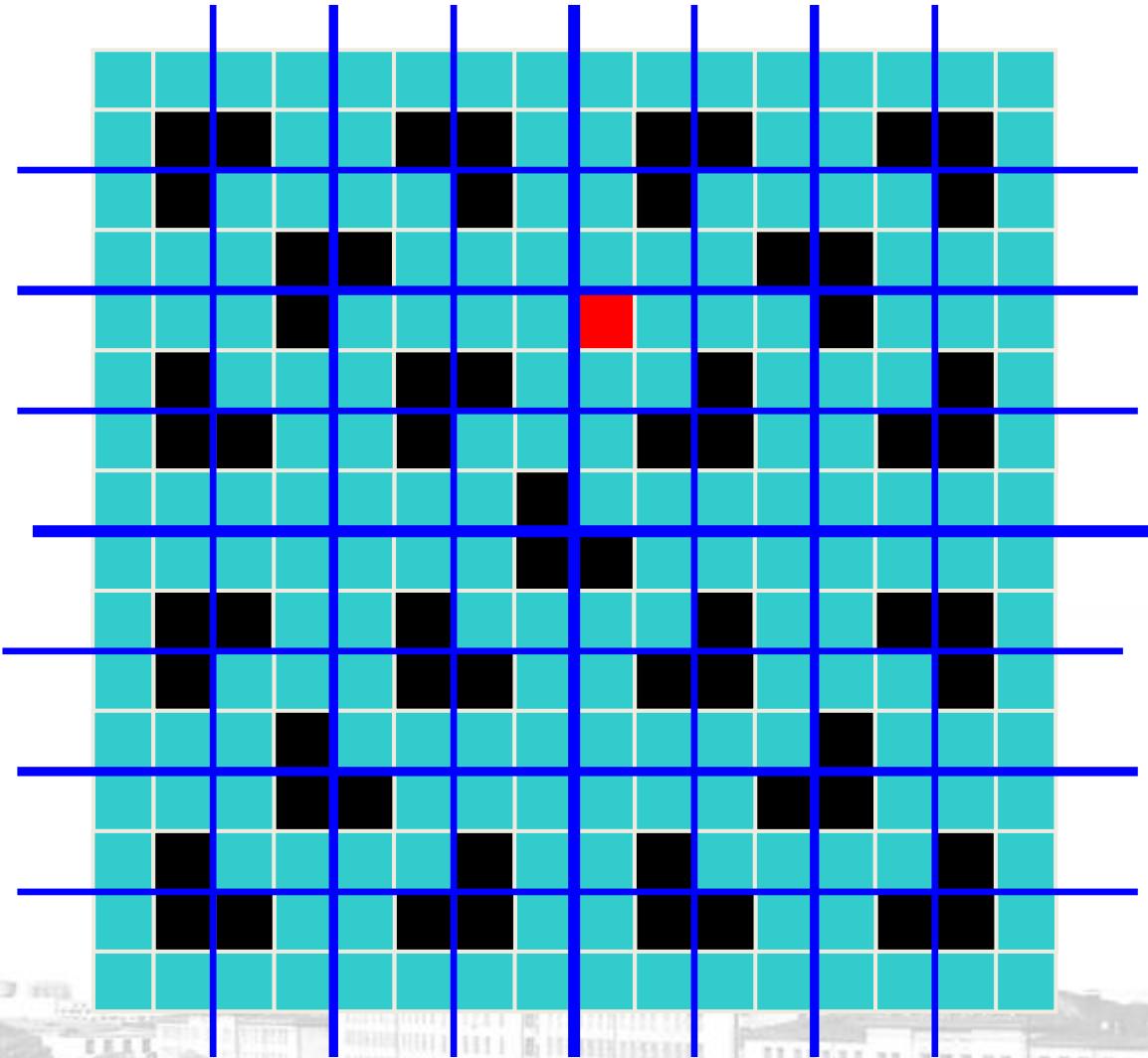
第一次分割



第二次分割



第三次分割



FillCheckBoard

Input: 棋盘的左上角坐标 (tr,tc); 特殊方块的位置(dr,dc); 棋盘大小 size;

Output: 使用 L型骨牌覆盖好的棋盘

if 特殊方格在左上角子棋盘中 then

 用L0型骨牌覆盖其余3个子棋盘相交位置

if 特殊方格在右上角子棋盘中 then

 用L1型骨牌覆盖其余3个子棋盘相交位置

if 特殊方格在左下角子棋盘中 then

 用L2型骨牌覆盖其余3个子棋盘相交位置

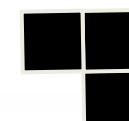
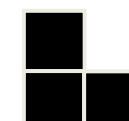
if 特殊方格在右下角子棋盘中 then

 用L3型骨牌覆盖其余3个子棋盘相交位置

If size == 2 then

 return 当前子棋盘

递归求解4个子棋盘



算法分析

建立递归方程：

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

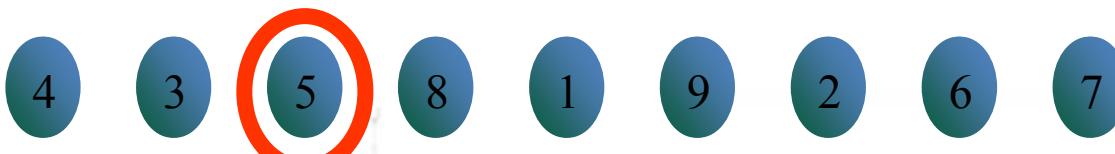
时间复杂度：

$$\begin{aligned} T(k) &= 4T(k - 1) + \Theta(1) \\ &= 4(4T(k - 2) + \Theta(1)) + \Theta(1) \\ &\quad \vdots \\ &= 4^k T(0) + \Theta(1) \sum_{i=0}^{k-1} 4^i \\ &= \Theta(4^k) \end{aligned}$$

中位数问题定义

Input: 由 n 个数构成的多重集合 X

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



中位数选取问题的复杂度

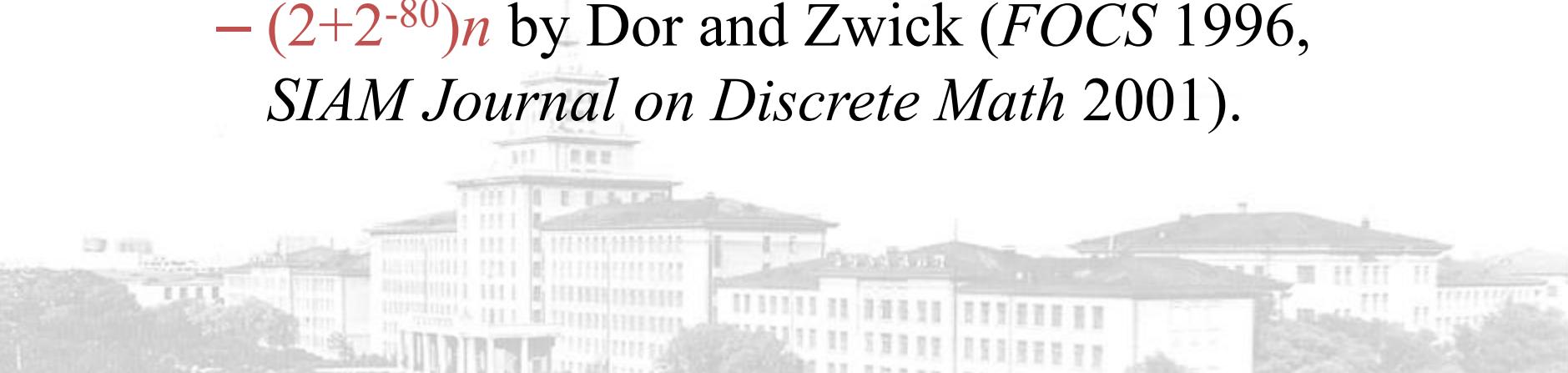


[Blum et al. *STOC'72 & JCSS'73*]

- 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

比较操作次数的上下界

- 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).



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

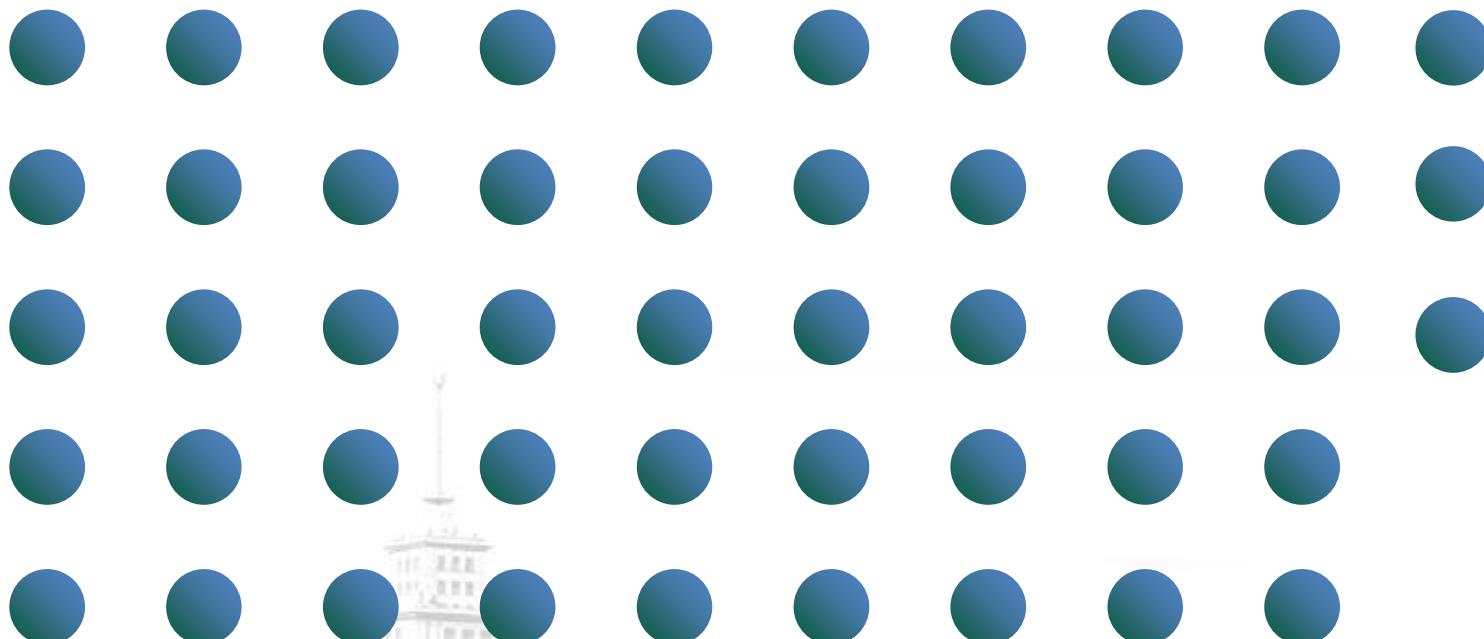
Input: n 个(不同)数构成的集合 X ,整数*i*,其中 $1 \leq i \leq n$

Output: $x \in X$ 使得 X 中恰有*i*-1个元素小于 x

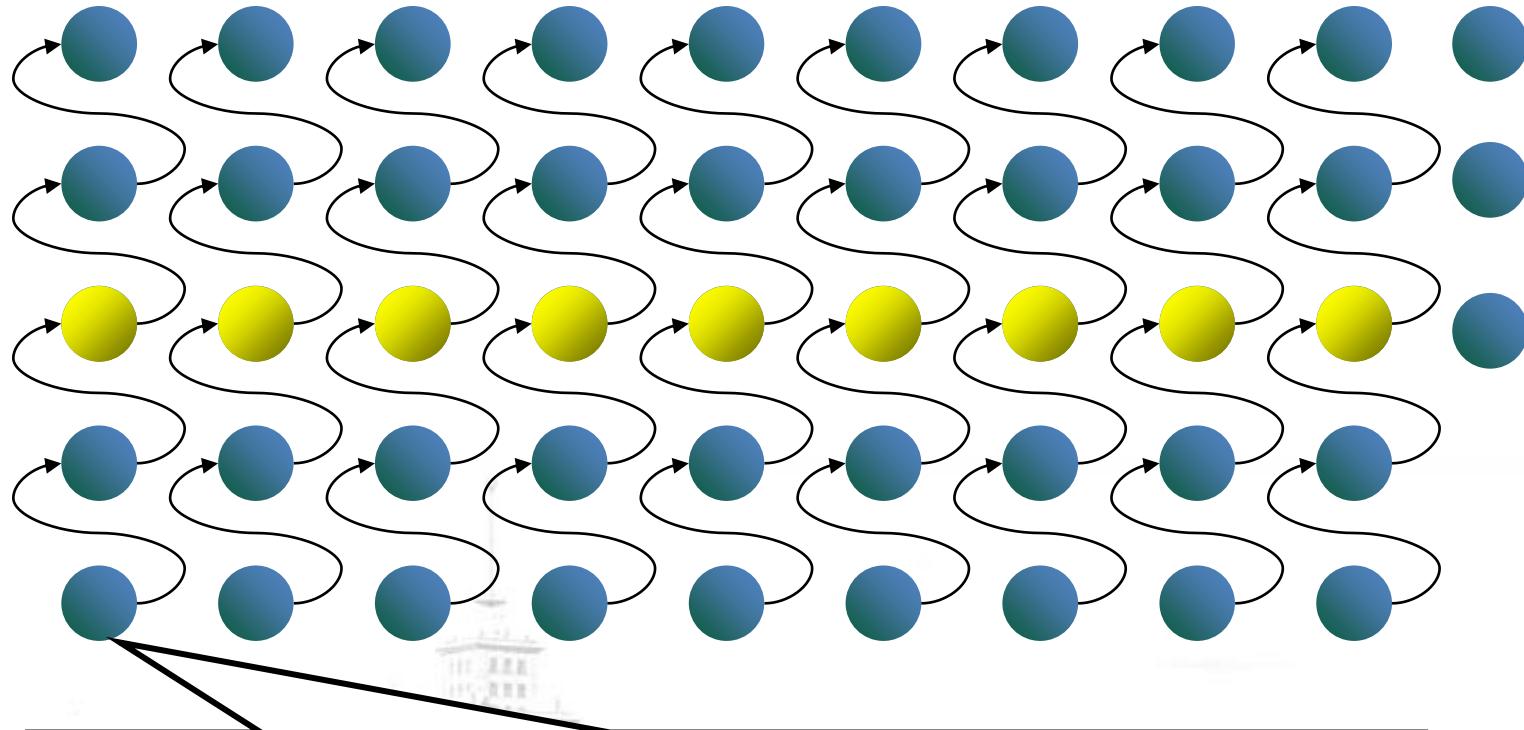
求解步骤

第一步：分组，每组5个数

最后一组可能少于5个数

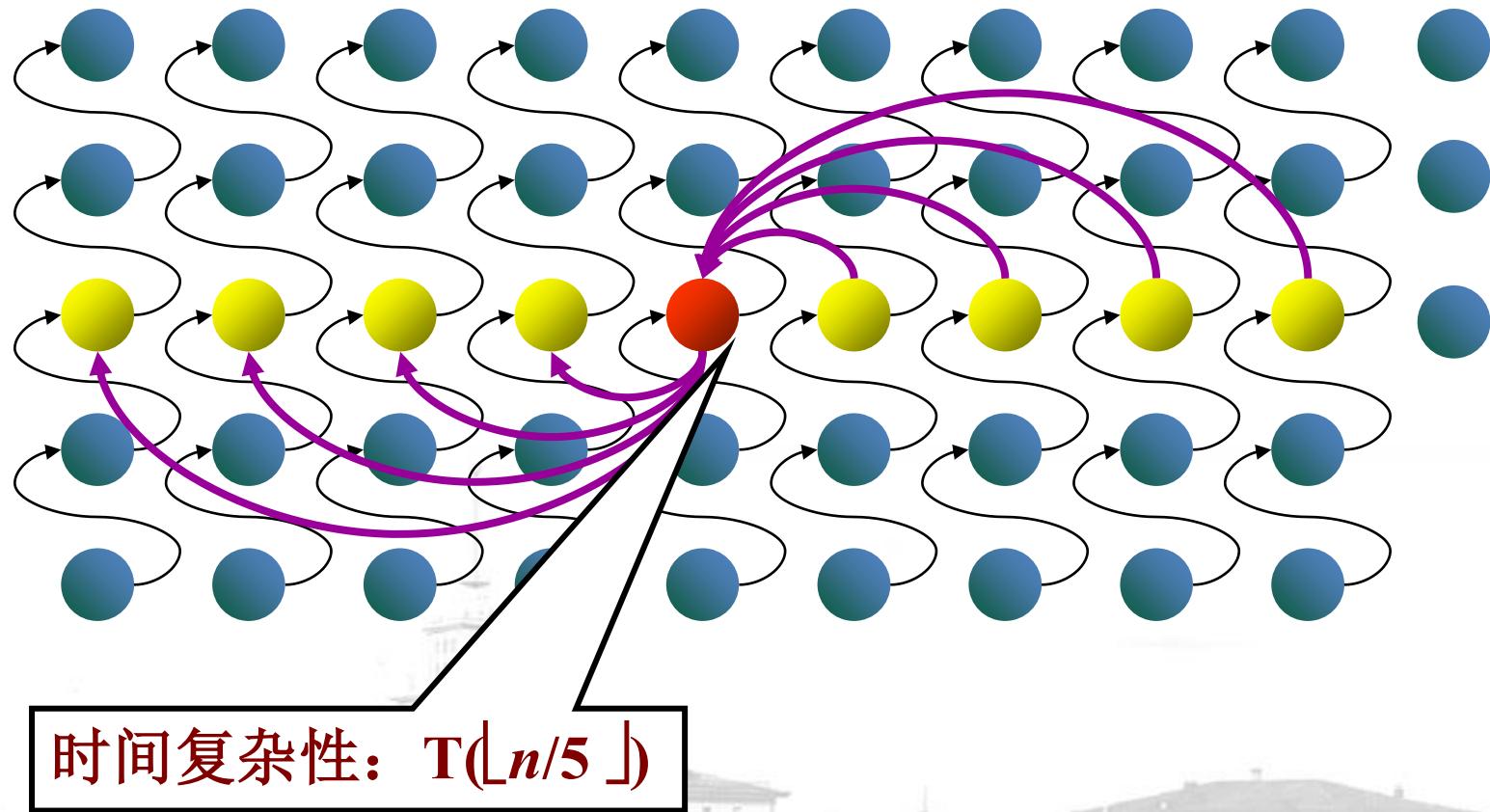


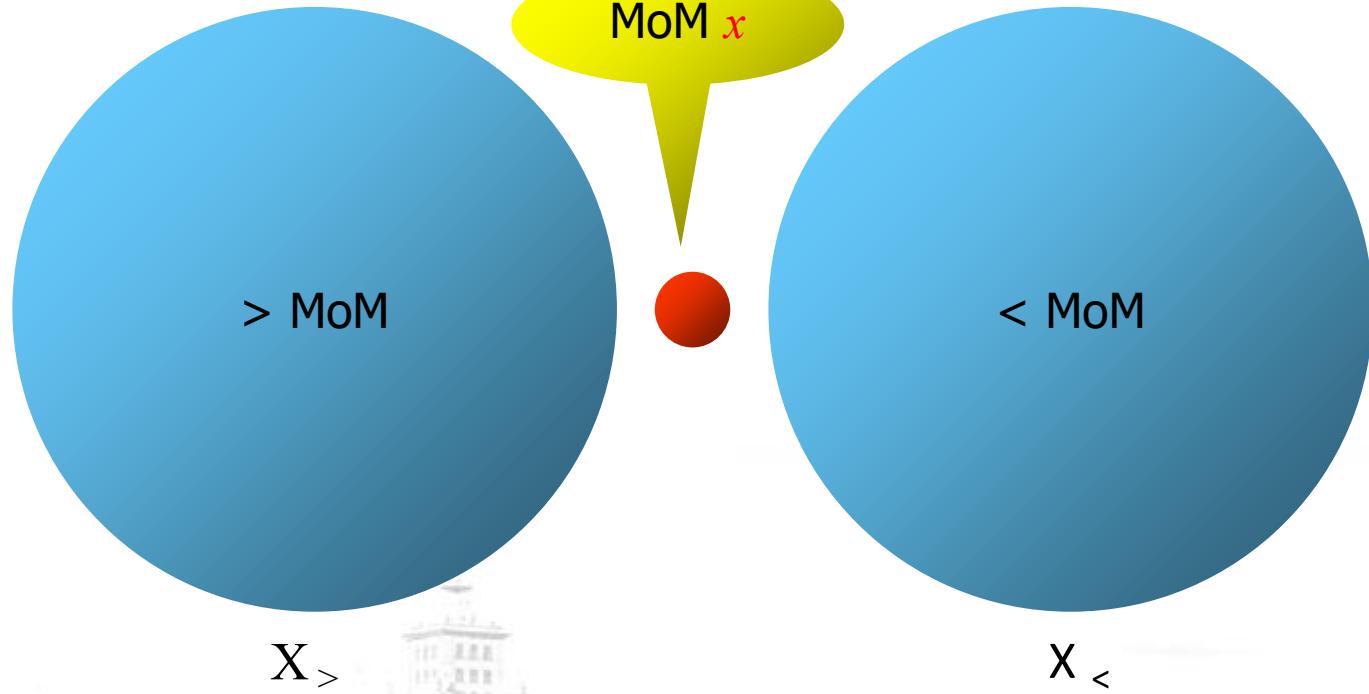
第二步：将每组数分别用InsertionSort排序 选出每组元素的中位数



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

第三步：递归调用算法求得这些中位数的中位数(MoM)





时间复杂性 $O(n)$

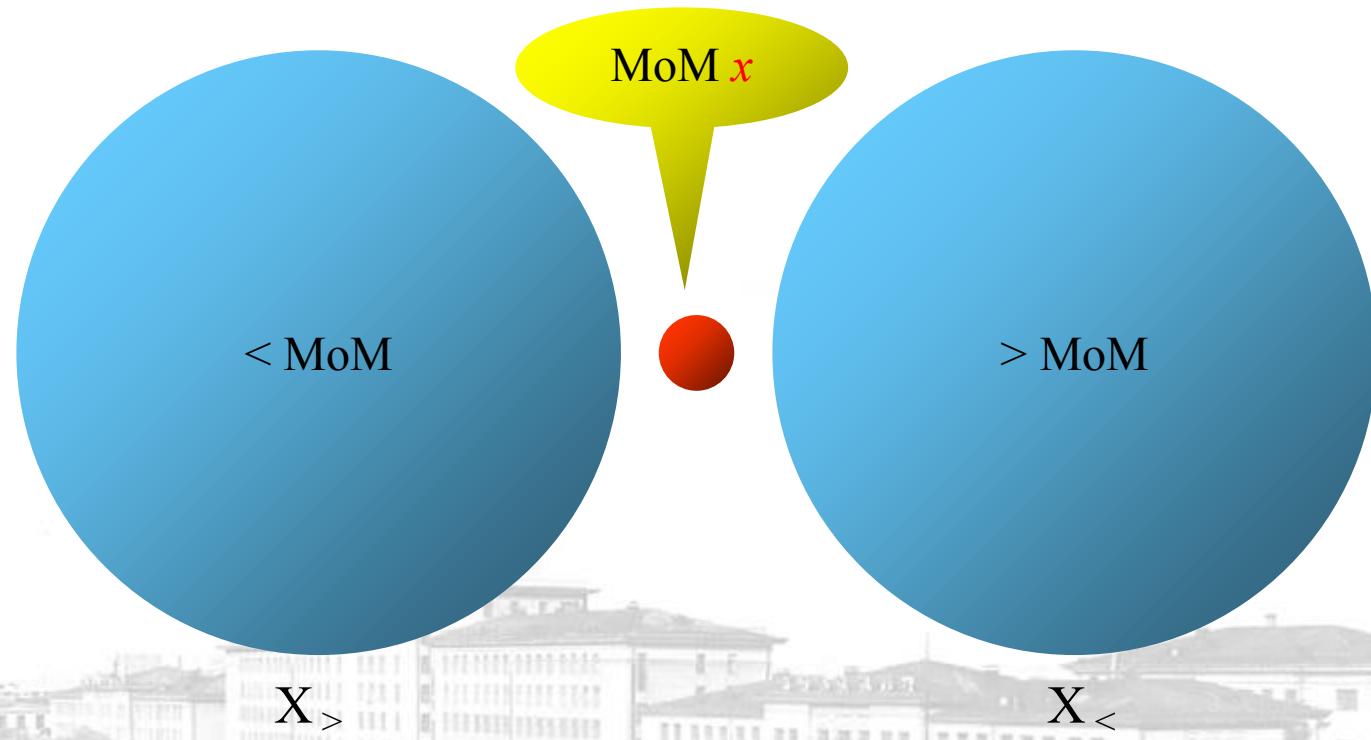
第五步:递归

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



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

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

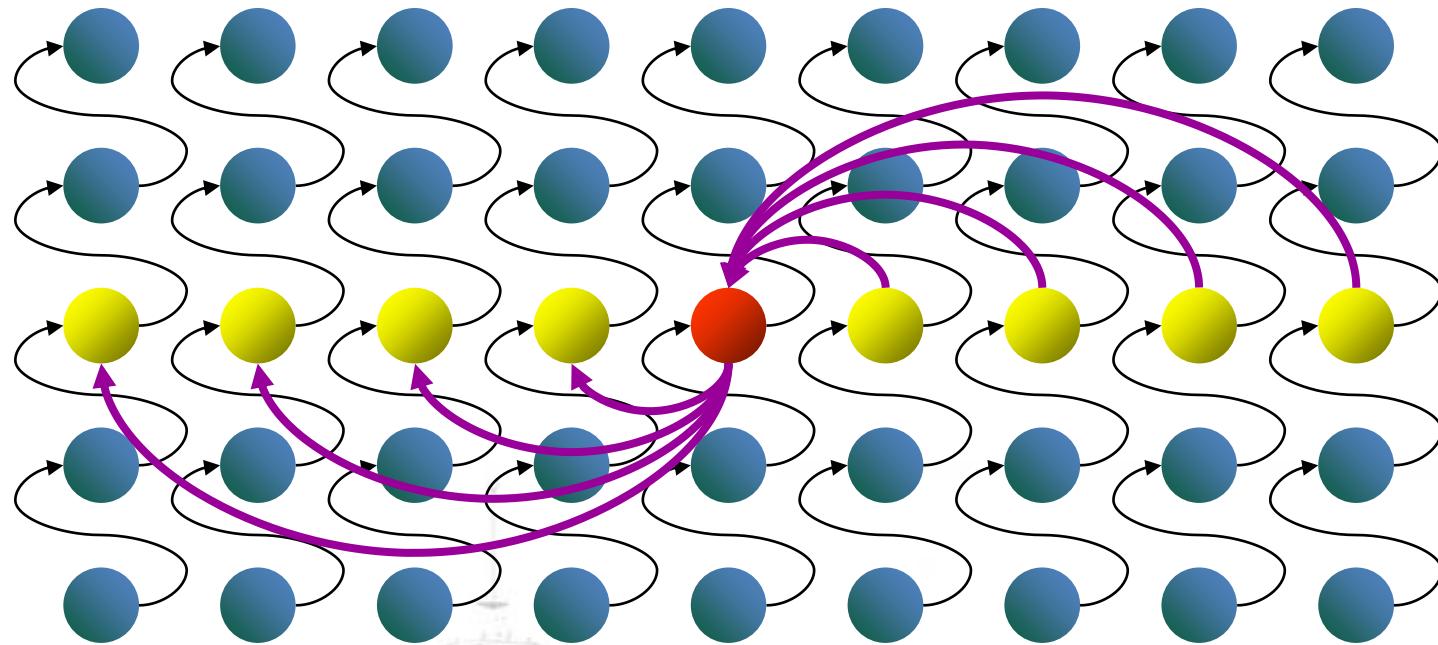
算法分析

算法Select(A,i)

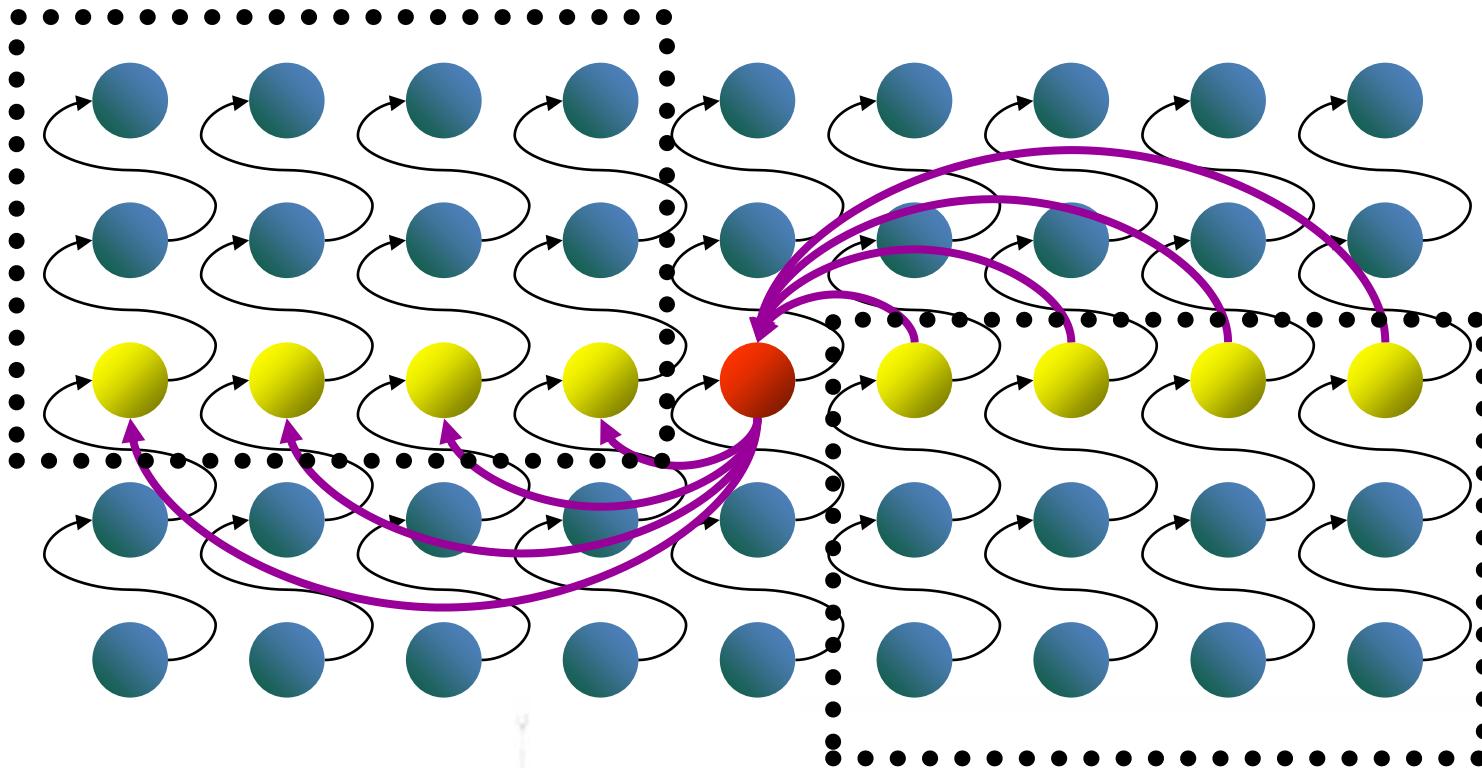
Input: 数组 $A[1:n]$, $1 \leq i \leq n$

Output: $A[1:n]$ 中的第 i -大的数

1. $\text{for } j \leftarrow 1 \text{ to } n/5$
2. $\text{InsertSort}(A[(j-1)*5+1 : (j-1)*5+5]);$
3. $\text{swap}(A[j], A[(j-1)*5+3]);$
4. $x \leftarrow \text{Select}(A[1: n/5], n/10);$ } $O(n)$
5. $k \leftarrow \text{partition}(A[1:n], x);$ } $T(\lfloor n/5 \rfloor)$
6. if $k=i$ then return $x;$
7. else if $k>i$ then retrun $\text{Select}(A[1:k-1],i);$
8. else retrun $\text{Select}(A[k+1:n],i-k);$



第五步至少删除了 $\lfloor 3n/10 \rfloor$ 个数



$$n - \lfloor 3n/10 \rfloor \leq 7n/10 + 6$$

如果时间复杂度是输入规模的递增函数
则第五步的时间开销不超过 $T(7n/10 + 6)$

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

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



3.2 introduction

排序的概念

排序是计算机内经常进行的一种操作，其目的是将一组“无序”的记录序列调整为“有序”的记录序列

例如，将下列关键字序列

52, 49, 80, 36, 14, 58, 61, 23, 97, 75

调整为

14, 23, 36, 49, 52, 58, 61 ,75, 80, 97

• 一般情况下，假设含n个记录的序列为

$$\{ R_1, R_2, \dots, R_n \}$$

其相应的关键字序列为

$$\{ K_1, K_2, \dots, K_n \}$$

这些关键字相互之间可以进行比较，即在它们之间存在着这样一个关系

$$K_{p1} \leq K_{p2} \leq \dots \leq K_{pn}$$

按此固有关系将式(1)的记录序列重新排列为

$$\{ R_{p1}, R_{p2}, \dots, R_{pn} \}$$

的操作称作排序

(张三,89), (李四,55), (王五,79), (大麻,92), (李亚鹏,10)

元组第一项表示姓名，元组第二项表示成绩

根据成绩从小到达可以将这些元组排序为

(李亚鹏,10), (李四,55), (王五,79), (张三,89), (大麻,92)

type define struct Record{

char name[20];

integer score;

} Record;

Record R[5];

R[0]=(张三,89), R[1]=(李四,55),...,R[4]=(李亚鹏,10)

内部排序与外部排序

- 若整个排序过程不需要访问外存便能完成，则称此类排序问题为 内部排序
- 若参加排序的记录数量很大，整个序列的排序过程不可能在内存中完成，则称此类排序问题为 外部排序
- 我们本章主要内部排序的各种方法



内部排序方法的分类

在排序的过程中，参与排序的记录序列中存在两个区域：有序区和无序区。内部排序的过程是一个逐步扩大记录的有序序列长度的过程



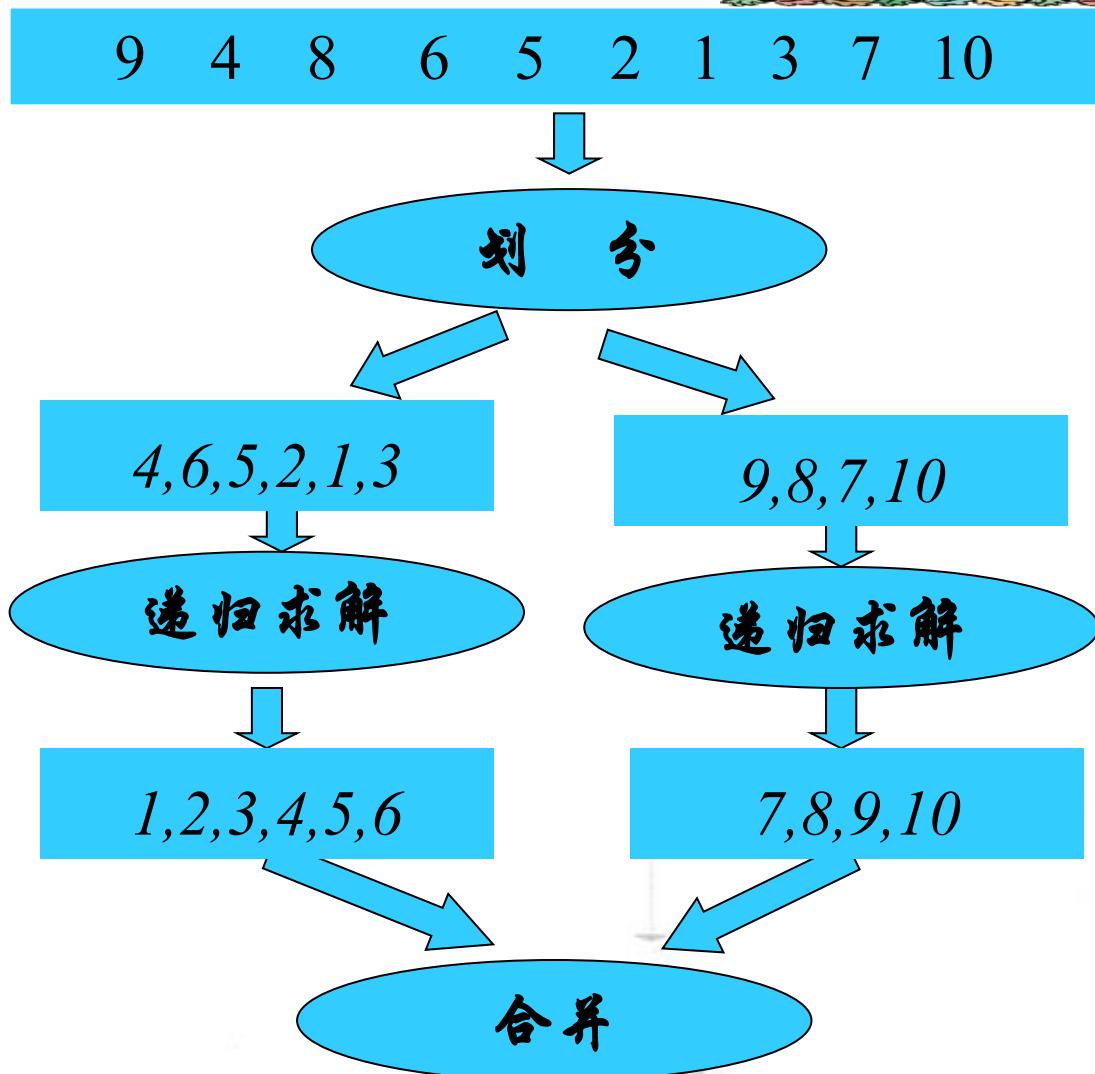
使有序区中记录的数目增加一个或几个的操作称为一趟排序



逐步扩大记录有序子序列长度的方法有下列几类：

- **插入类** 将无序子序列中的一个或几个记录“插入”到有序序列中，从而增加记录的有序子序列的长度
- **选择类** 从记录的无序子序列中“选择”关键字最小或最大的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度
- **交换类** 通过“交换”无序序列中的记录从而得到其中关键字最小或最大的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度
- **归并类** 通过“归并”两个或两个以上的记录有序子序列，逐步增加记录有序序列的长度
- **其它方法**

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

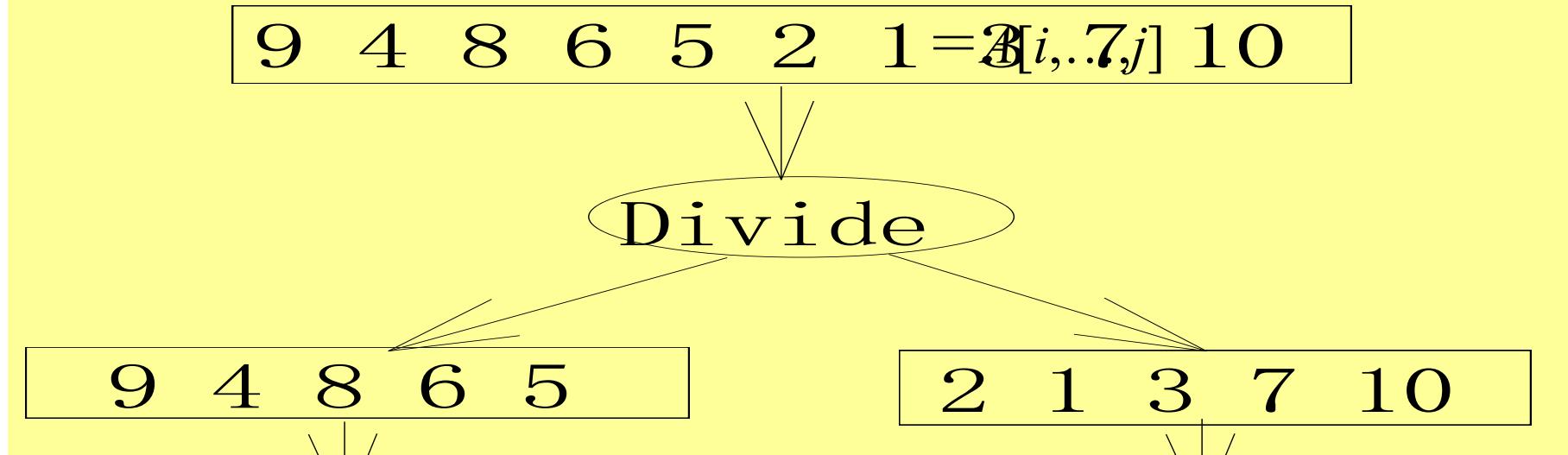
2. 选择一个划分标准 x 根据元素与 x 的大小关系来划分 quicksort

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 $A[i, \dots, j]$ 中的元素排序

3.3 Merge-sort 算法



Divide:

本质上仅需产生划分位置 k

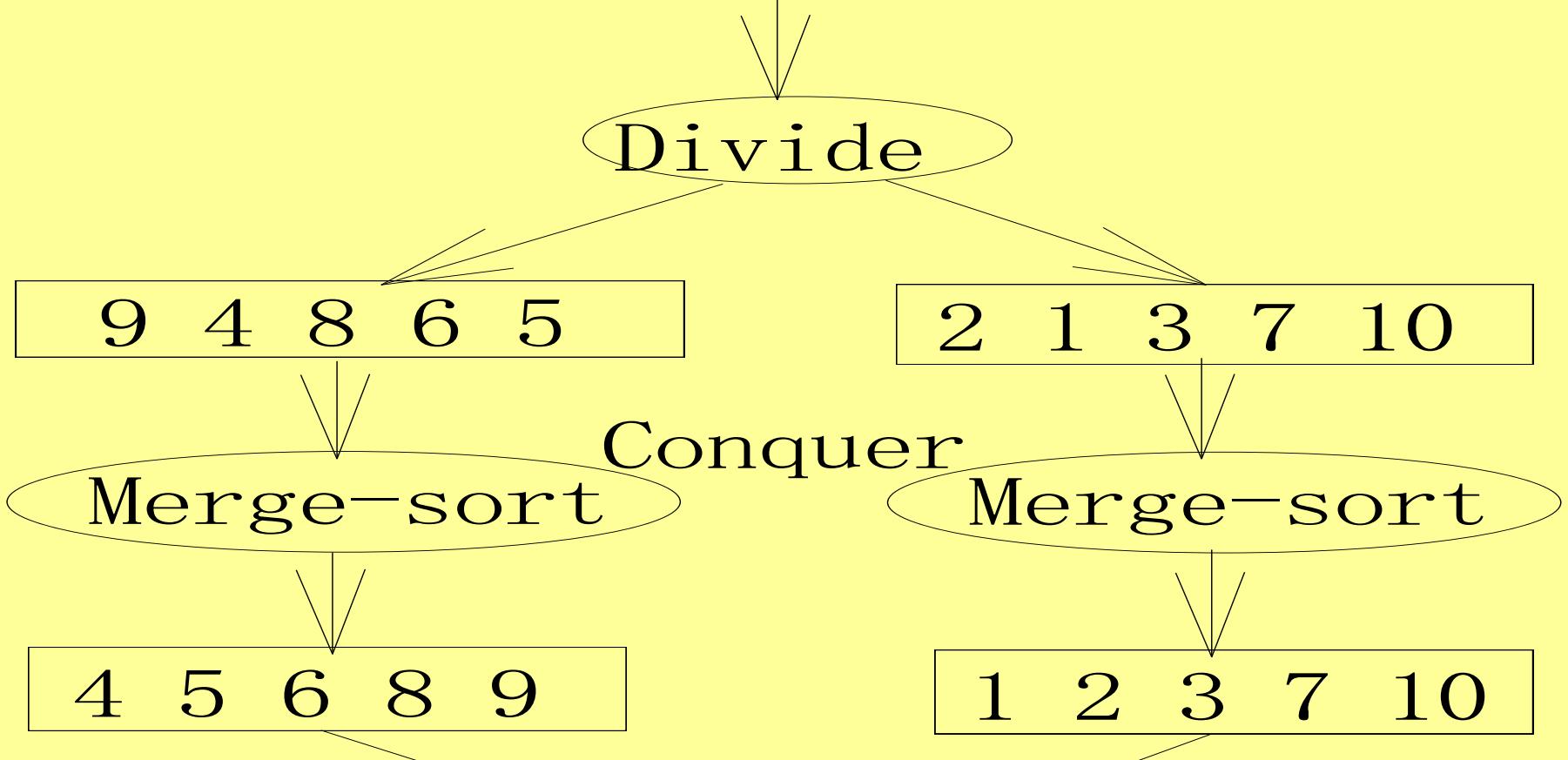
第一个子问题是 $A[i, \dots, k]$

第二个子问题是 $A[k+1, \dots, j]$

而使得两个问题的大小大致相当， k 可以如下产生

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

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



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

combine: 将两个有序序列合并成一个有序序列

1. $l \leftarrow i; h \leftarrow k+1; t \leftarrow i$ //设置指针
2. While $l \leq k \& h < j$ Do
 - IF $A[l] < A[h]$ THEN $B[t] \leftarrow A[l]; l \leftarrow l+1; t \leftarrow t+1;$
 - ELSE $B[t] \leftarrow A[h]; h \leftarrow h+1; t \leftarrow t+1;$
3. IF $l < k$ 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 $h < j$ 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

MergeSort(A, i, j)

Input: $A[i, \dots, j]$

Output: 排序后的 $A[i, \dots, j]$

1. $k \leftarrow (i+j)/2;$
2. $\text{MergeSort}(A, i, k);$
3. $\text{MergeSort}(A, k+1, j);$
4. $l \leftarrow i; h \leftarrow k+1; t = i$ //设置指针
5. **While** $l \leq k \ \& \ h < j$ **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** $l < k$ **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** $h < j$ **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. $A[v] \leftarrow B[v];$

Mergesort 算法

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

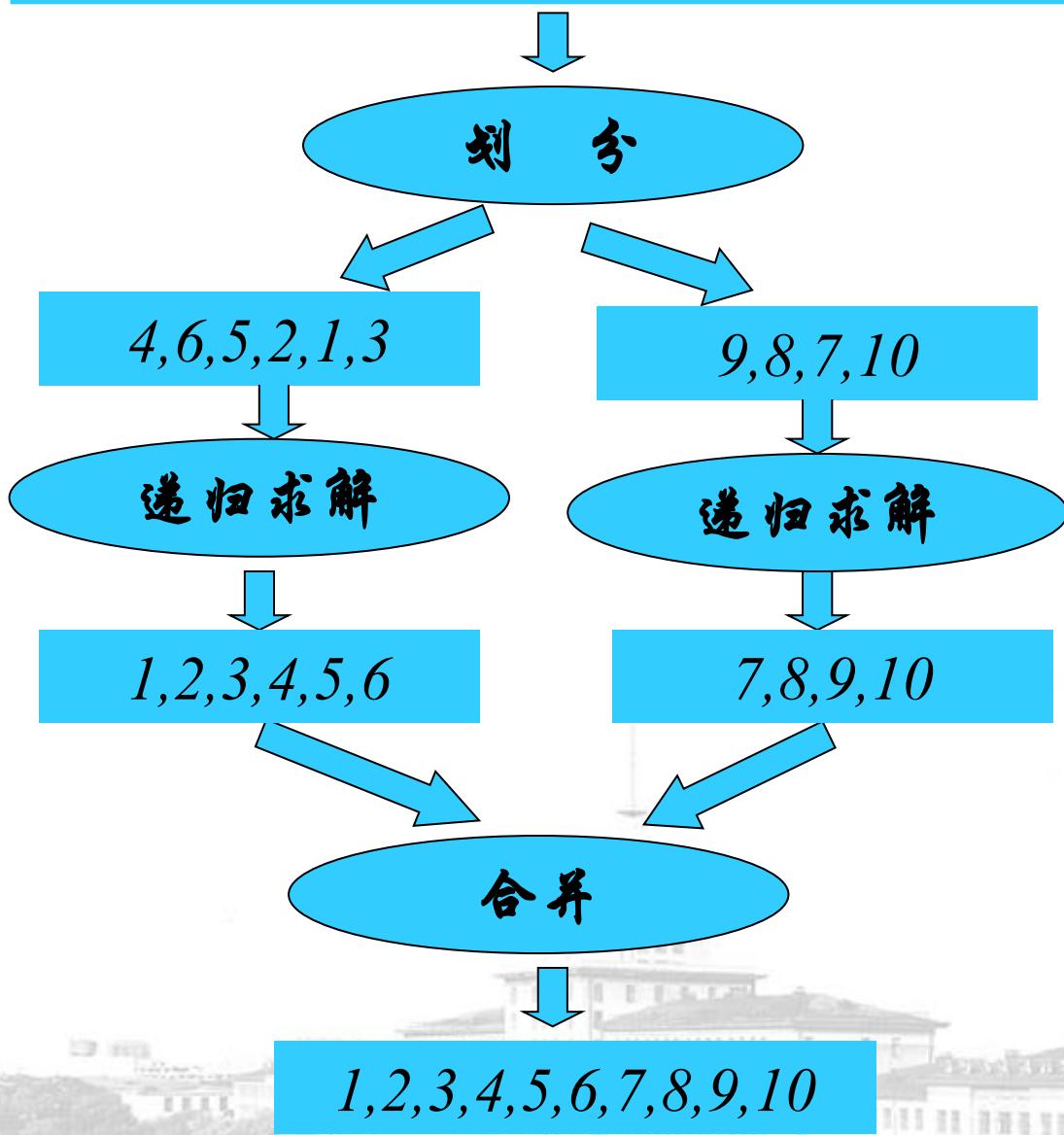
$$T(n) = O(n \log n)$$

3.4 QuickSort



Partition sort

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



基本思想

Divide:

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

Conquer:

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

Combine:

无操作

PartitionSort 算法框架

PartitionSort(A, i, j)

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

Output: 排序后的 $A[i, \dots, j]$

1. 选择划分元素 x ;
2. $k = \text{partition}(A, i, j, x);$ //用 x 完成划分
3. $\text{partitionSort}(A, i, k);$ //递归求解子问题
4. $\text{partitionSort}(A, k+1, j);$ //无需额外的合并操作

Divide

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

$x=7$

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

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

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

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

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

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

如果确实应该交low和high标记的两个元素的位置，则交换

否则划分位置就是high

Divide过程的算法描述

Partition(A, i, j, x)

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

Output: 划分位置 k 使得 $A[i, \dots, k]$ 中的元素均小于 x 且 $A[k+1, \dots, j]$ 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($A[high] \geq x$) Do
7. $high \leftarrow high - 1$;
8. return($high$)

PartitionSort 算法

PartitionSort(A, i, j)

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

Output: 排序后的 $A[i, \dots, j]$

1. $x \leftarrow A[i];$ //以确定的策略选择 x
2. $k = \text{partition}(A, i, j, x);$ //用 x 完成划分
3. $\text{partitionSort}(A, i, k);$ //递归求解子问题
4. $\text{partitionSort}(A, k+1, j);$

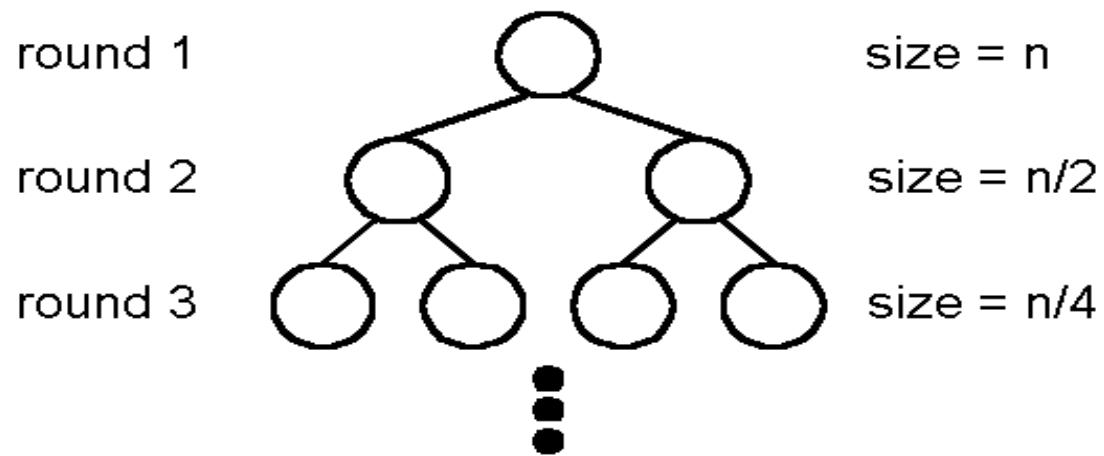
Partition(A, i, j, x)

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

算法复杂性的分析

最好情况 : $\Theta(n \log n)$

数组被分为大致相等的两个部分.



- 需要 $\lg_2 n$ 轮.
- 每轮需要 $O(n)$ 次比较

算法复杂性的分析

最坏情况 : $\Theta(n^2)$

每一轮用最大或者最小元素划分

$$(n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2} = \Theta(n^2)$$

算法复杂性的分析

平均情况: $\Theta(n \log n)$



$$\begin{aligned} T(n) &= \text{Avg}(T(s) + T(n - s)) + cn \\ &\quad 1 \leq s \leq n \\ &= \frac{1}{n} \sum_{s=1}^n (T(s) + T(n - s)) + cn \\ &= \frac{1}{n} (T(1) + T(n-1) + T(2) + T(n-2) + \dots + T(n) + T(0)) + cn, \quad T(0)=0 \\ &= \frac{1}{n} (2T(1) + 2T(2) + \dots + 2T(n-1) + T(n)) + cn \end{aligned}$$

算法复杂性的分析

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

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

$$(1) - (2)$$

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

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

$$\begin{aligned} \frac{T(n)}{n} &= \frac{T(n-1)}{n-1} + c\left(\frac{1}{n} + \frac{1}{n-1}\right) \\ &= c\left(\frac{1}{n} + \frac{1}{n-1}\right) + c\left(\frac{1}{n-1} + \frac{1}{n-2}\right) + \cdots + c\left(\frac{1}{2} + 1\right) + T(1), \quad T(1) = 0 \\ &= c\left(\frac{1}{n} + \frac{1}{n-1} + \dots + \frac{1}{2}\right) + c\left(\frac{1}{n-1} + \frac{1}{n-2} + \dots + 1\right) \end{aligned}$$

算法复杂性的分析

调和级数 [Knuth 1986]

$$\begin{aligned} H_n &= 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \\ &= \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \varepsilon, \text{ where } 0 < \varepsilon < \frac{1}{252n^6} \\ \gamma &= 0.5772156649\dots \end{aligned}$$

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

$$\begin{aligned} \frac{T(n)}{n} &= c(H_n - 1) + cH_{n-1} \\ &= c\left(2H_n - \frac{1}{n} - 1\right) \end{aligned}$$

$$\begin{aligned} \Rightarrow T(n) &= 2c n H_n - c(n+1) \\ &= O(n \log n) \end{aligned}$$

随机QuickSort算法

QuickSort(A, i, j)

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

Output: 排序后的 $A[i, \dots, j]$

1. $temp \leftarrow \text{rand}(i, j);$ //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. $k = \text{partition}(A, i, j, x);$ //用 x 完成划分
4. $\text{partitionSort}(A, i, k);$ //递归求解子问题
5. $\text{partitionSort}(A, k+1, j);$

Partition(A, i, j, x)

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

算法性能的分析

- 基本概念

- $S_{(i)}$ 表示 S 中阶为 i 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:

$X_{ij}=1$ 如果 $S_{(i)}$ 和 $S_{(j)}$ 在运行中被比较, 否则为 0

- X_{ij} 是 $S_{(i)}$ 和 $S_{(j)}$ 的比较次数

- 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 $E\left[\sum_{i=1}^n \sum_{j>i} X_{ij}\right] = \sum_{i=1}^n \sum_{j>i} E[X_{ij}]$

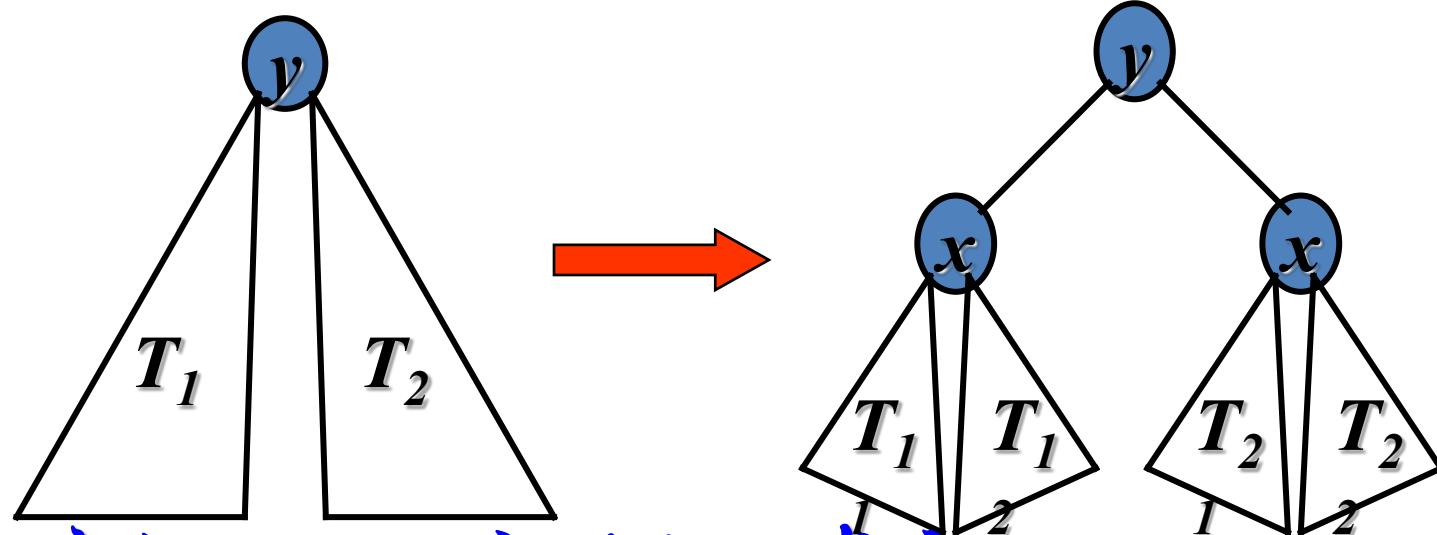
- 计算 $E[X_{ij}]$
 - 设 p_{ij} 为 $S_{(i)}$ 和 $S_{(j)}$ 在运行中比较的概率，则

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

关键问题成为求解 P_{ij}

- 求解 P_{ij}

- 我们可以用树表示算法的计算过程



- 我们可以观察到以下事实：

- 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

- 当 $S_{(i)}, S_{(i+1)}, \dots, S_{(j)}$ 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 $S_{(i+1)}, \dots, S_{(j-1)}$ 被选为划分点时, $S_{(i)}$ 和 $S_{(j)}$ 才可能比较
- $S_{(i)}, S_{(i+1)}, \dots, S_{(j)}$ 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

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

• 現在我們有

$$\begin{aligned} \sum_{i=1}^n \sum_{j>i} E[X_{ij}] &= \sum_{i=1}^n \sum_{j>i} p_{ij} = \sum_{i=1}^n \sum_{j>i} \frac{2}{j-i+1} \\ &\leq \sum_{i=1}^n \sum_{k=1}^{n-i+1} \frac{2}{k} \leq 2 \sum_{i=1}^n \sum_{k=1}^n \frac{1}{k} = 2nH_n = O(n \log n) \end{aligned}$$

定理. 隨機快速排序算法的期望時間複雜性為 $O(n \log n)$

3.5 排序问题的下界

- 问题的下界是解决该问题的算法所需要的最长时间复杂性。
 - ☆ 最坏情况下界
 - ☆ 平均情况下界
- 排序问题的下界是不唯一的
 - 例如. $\Omega(n)$, $\Omega(n \log n)$ 都是排序的下界



- 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.

最坏情况下排序的下界

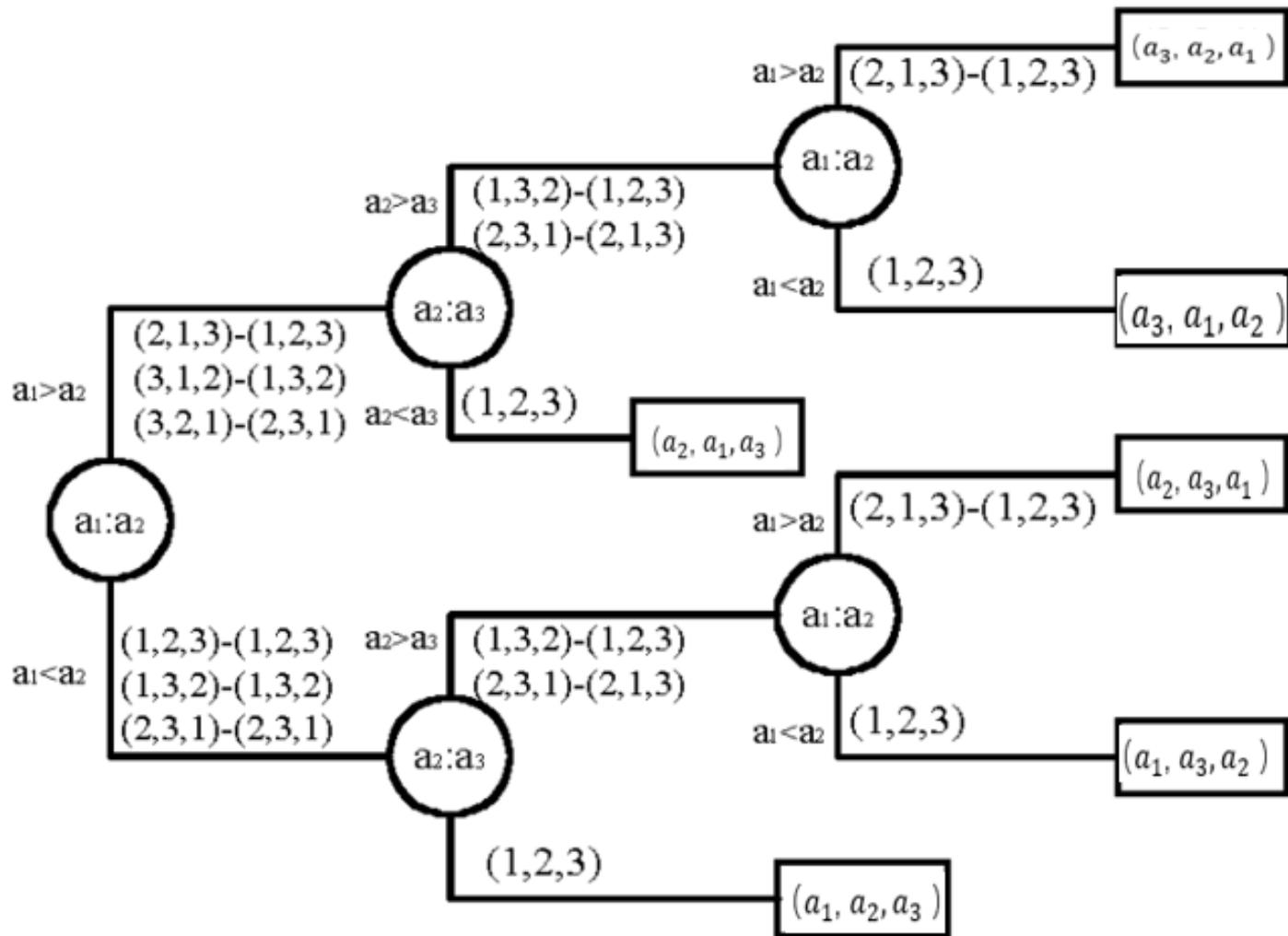
3个元素有6种排列

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

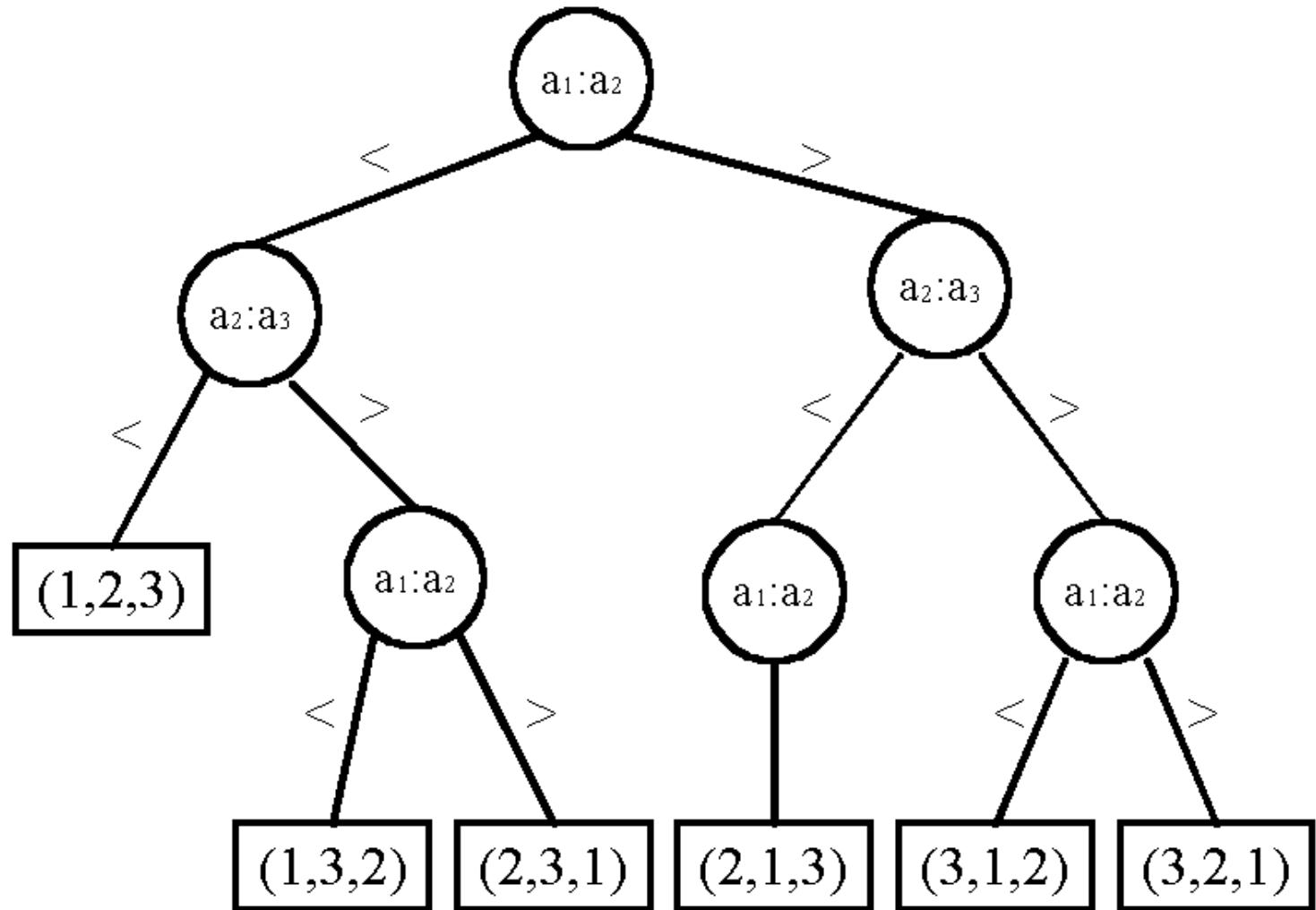
直接插入排序

- 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) $a_2:a_3$

直接插入排序的决策树



冒泡排序的决策树

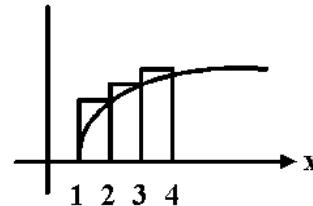


排序的下界

- 为了找到排序的下界，我们需要找到二叉树的最小高度.
- 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: $\Omega(n \log n)$

方法 1:

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



方法2:

- Stirling近似:

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

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

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	2.433x10 ¹⁸	2.423x10 ¹⁸
100	9.333x10 ¹⁵⁷	9.328x10 ¹⁵⁷

排序的平均情况下界

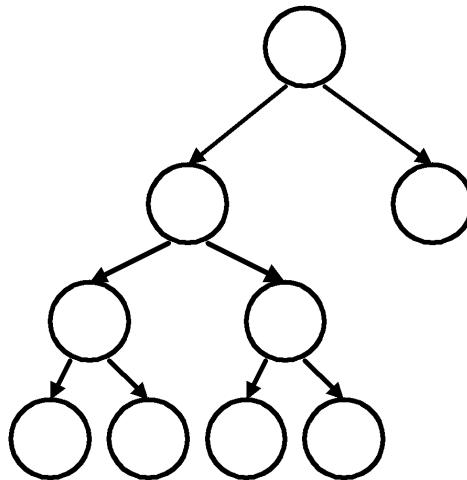
- 仍利用决策树
- 排序算法的平均复杂性用从根结点到每个叶子结点的路径长度的总长度描述。

$$n!$$

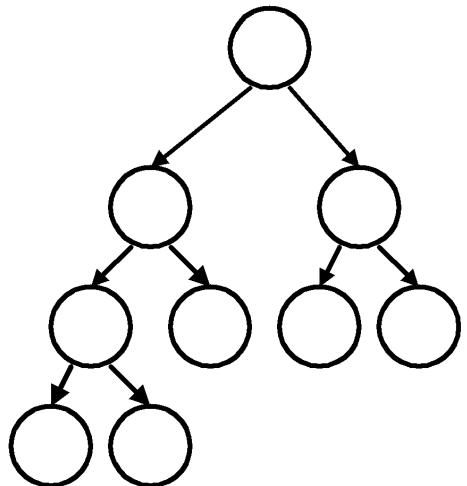
- 当树平衡时这个值最小.
(所有叶子结点的深度是 d 或 $d-1$)



平均情况下排序的下界



不平衡的情况下
总长度
 $= 4 \cdot 3 + 1 = 13$



平衡的情况下
总长度
 $= 2 \cdot 3 + 3 \cdot 2 = 12$

计算最小路径长度和

1. 有 c 个叶子结点的二叉树的深度 $d = \lceil \log c \rceil$
叶子结点仅出现在 d 或 $d-1$ 层.
2. x_1 个结点在 $d-1$ 层
 x_2 个结点在 d 层

■ $x_1 + x_2 = c$

■ $x_1 + \frac{x_2}{2} = 2^{d-1}$

$$\Rightarrow x_1 = 2^d - c$$
$$x_2 = 2(c - 2^{d-1})$$

3. 总长度

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

4. $c = n!$

$$\begin{aligned}M &= n! \lfloor \log n! \rfloor + 2(n! - 2^{\lfloor \log n! \rfloor}) \\M/n! &= \lfloor \log n! \rfloor + 2 \\&= \lfloor \log n! \rfloor + c, \quad 0 \leq c \leq 1 \\&= \Omega(n \log n)\end{aligned}$$

平均情况下排序的下界是: $\Omega(n \log n)$

思考题 1

给定长度为n的单调不减数列数 a_0, \dots, a_n 和一个数k，在 $O(\log n)$ 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



INPUT: A=[2,3,3,5,6],k=3

- Lowbound=-1 Upbound=5



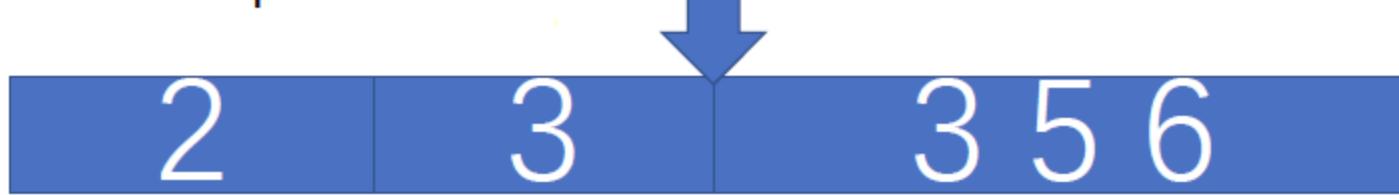
- Lowbound=-1 Upbound=3



- Lowbound =-1 Upbound=2



- Lowbound =1 Upbound= 2



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;
```



最坏情况查找最后一个元素（或者第一个元素）

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



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

思考题 2

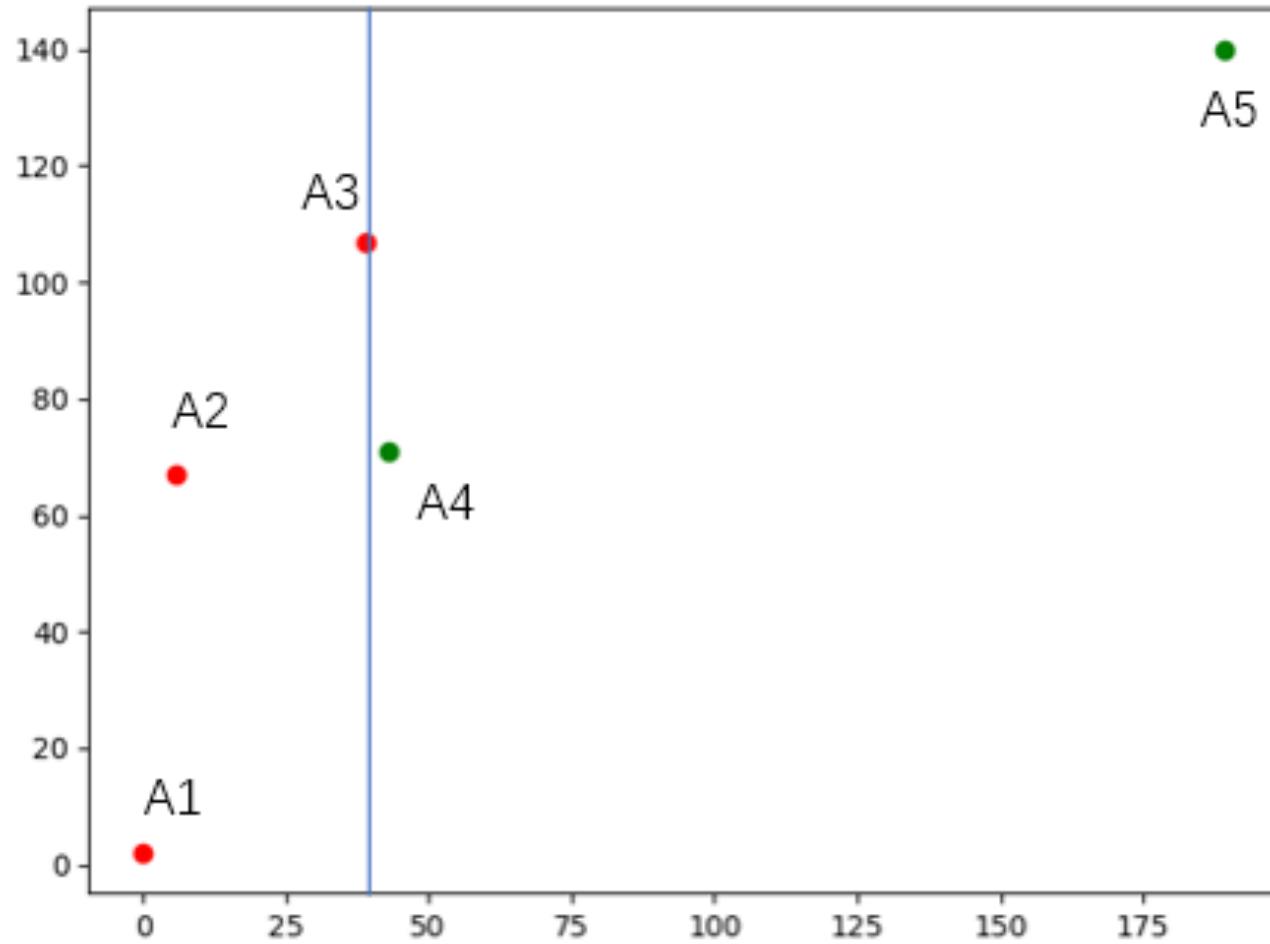
给定平面上的n个点， $O(n \log n)$ 的时间复杂度内求出距离最近两点距离，写出伪代码

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

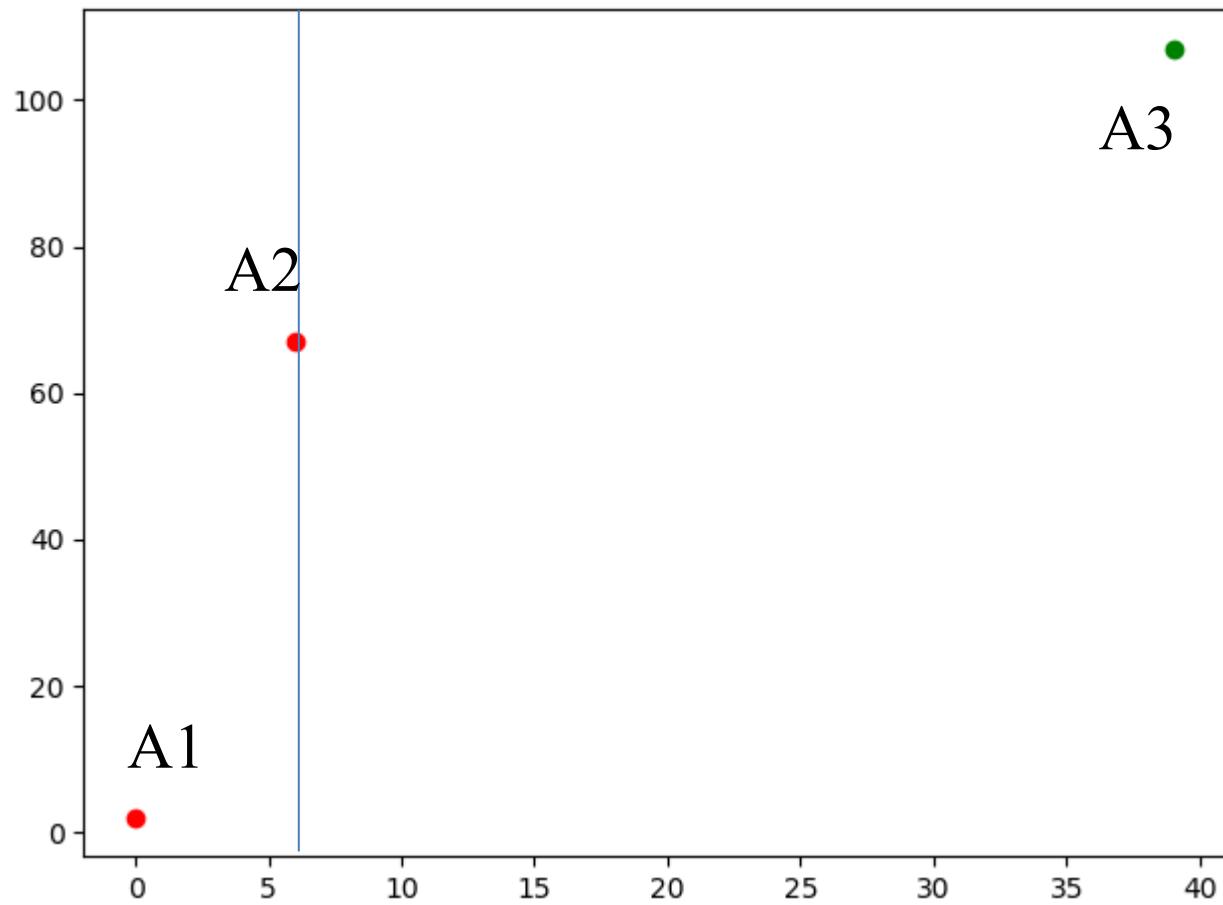
output: 36.22



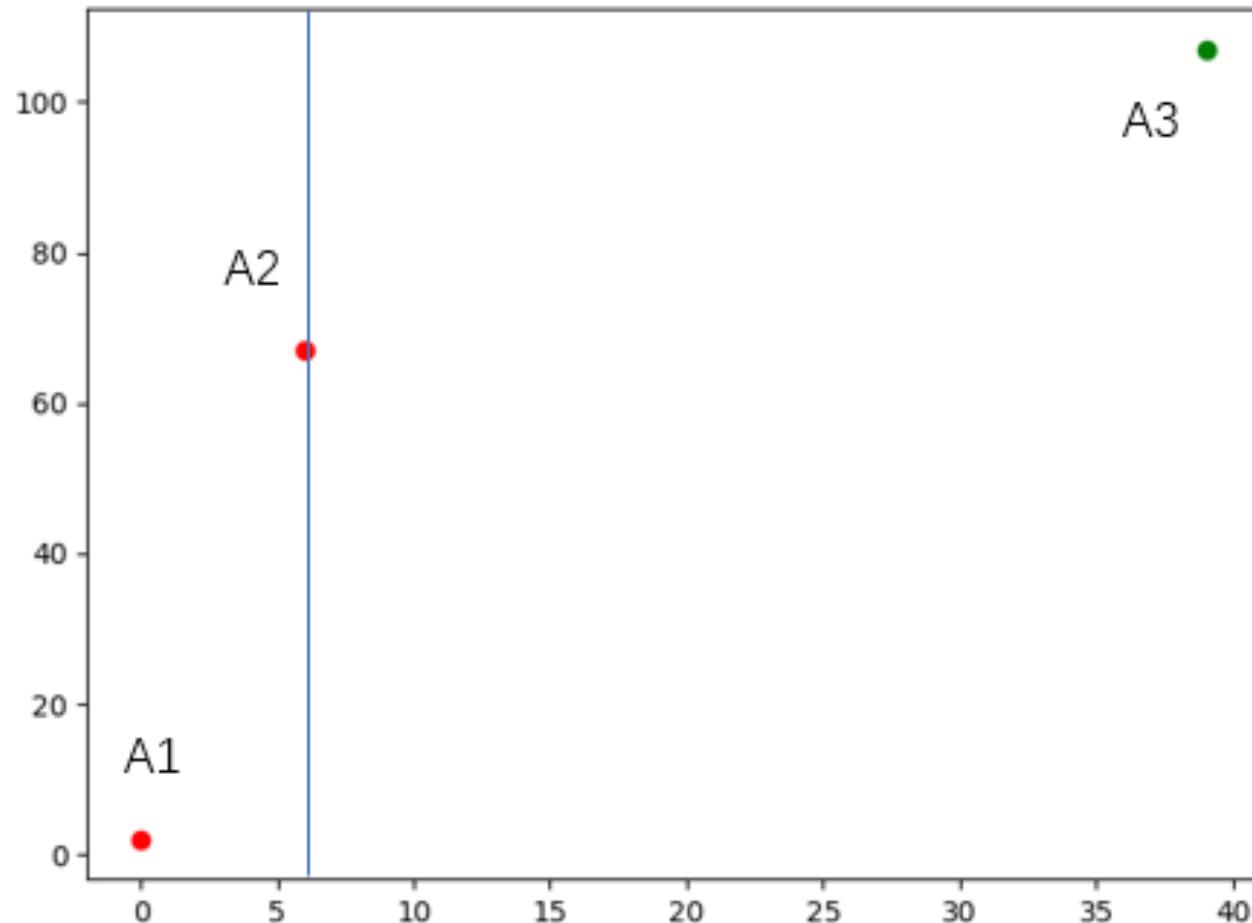
按照X坐标分为左右两个部分



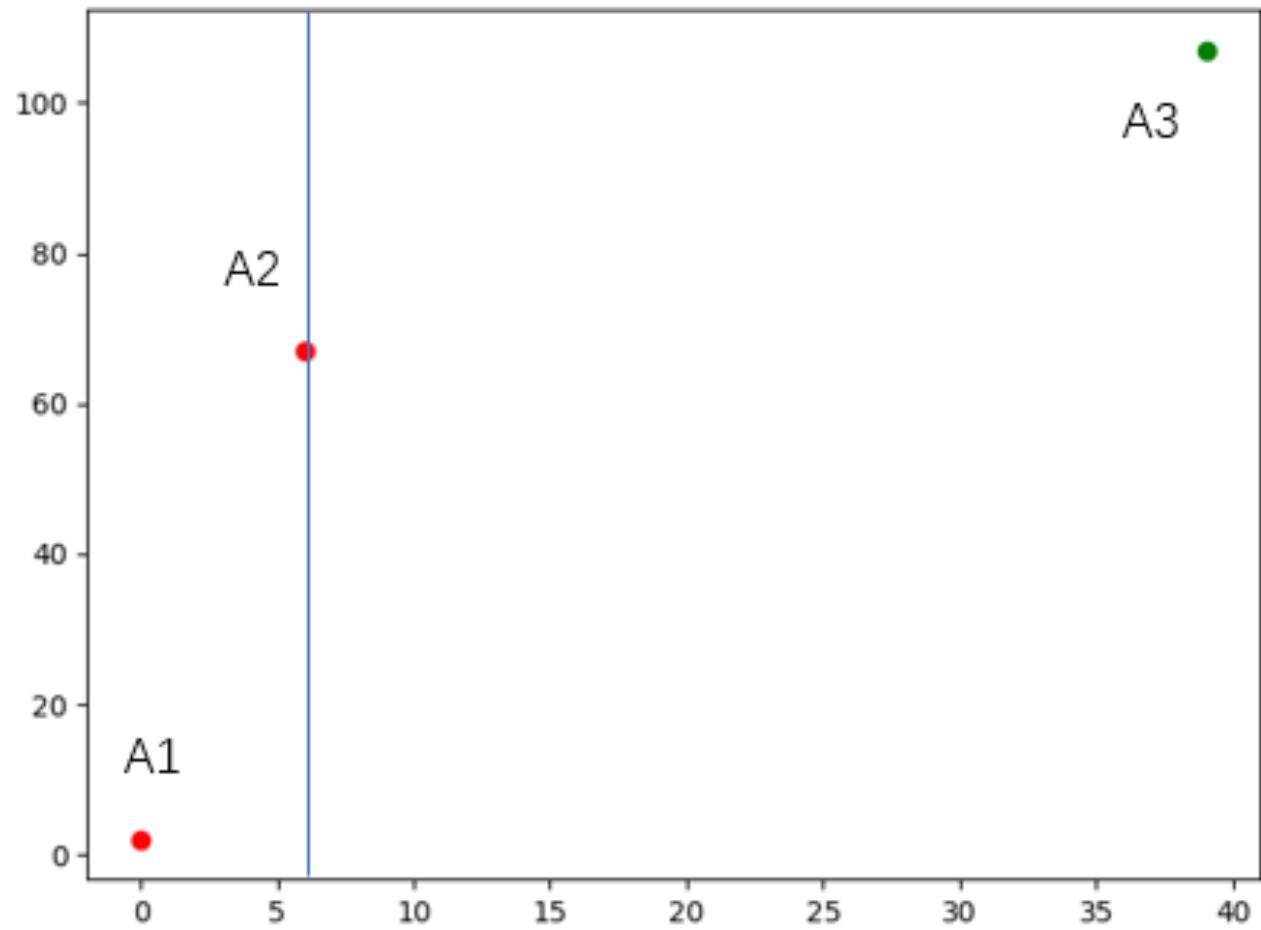
继续划分子问题



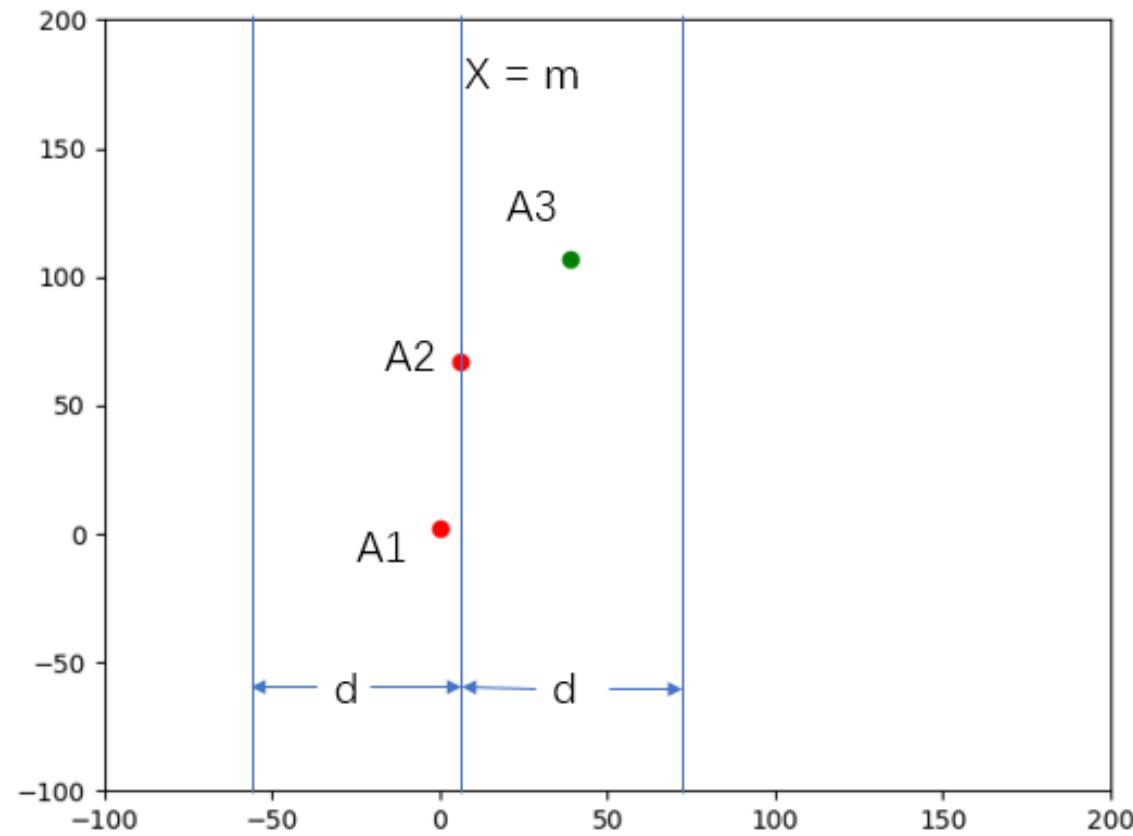
计算左区域的最小距离 d_1



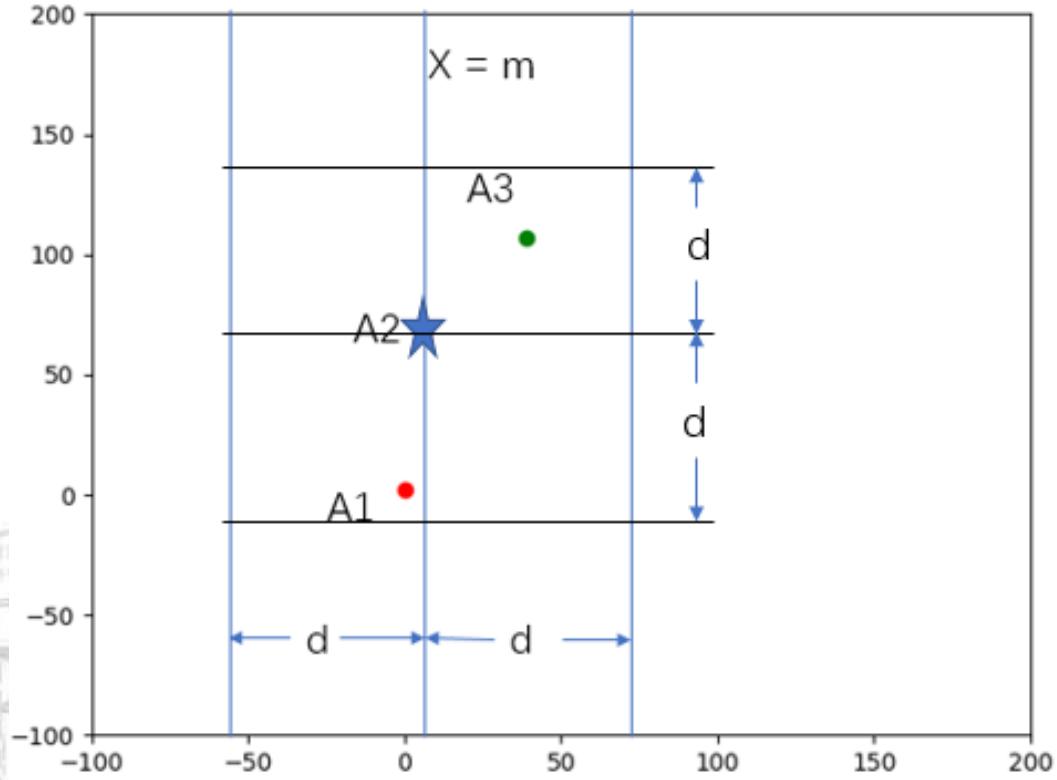
计算右区域的最小距离 d_2 ，因此左右两个区域的最小距离 $d = \min\{d_1, d_2\}$



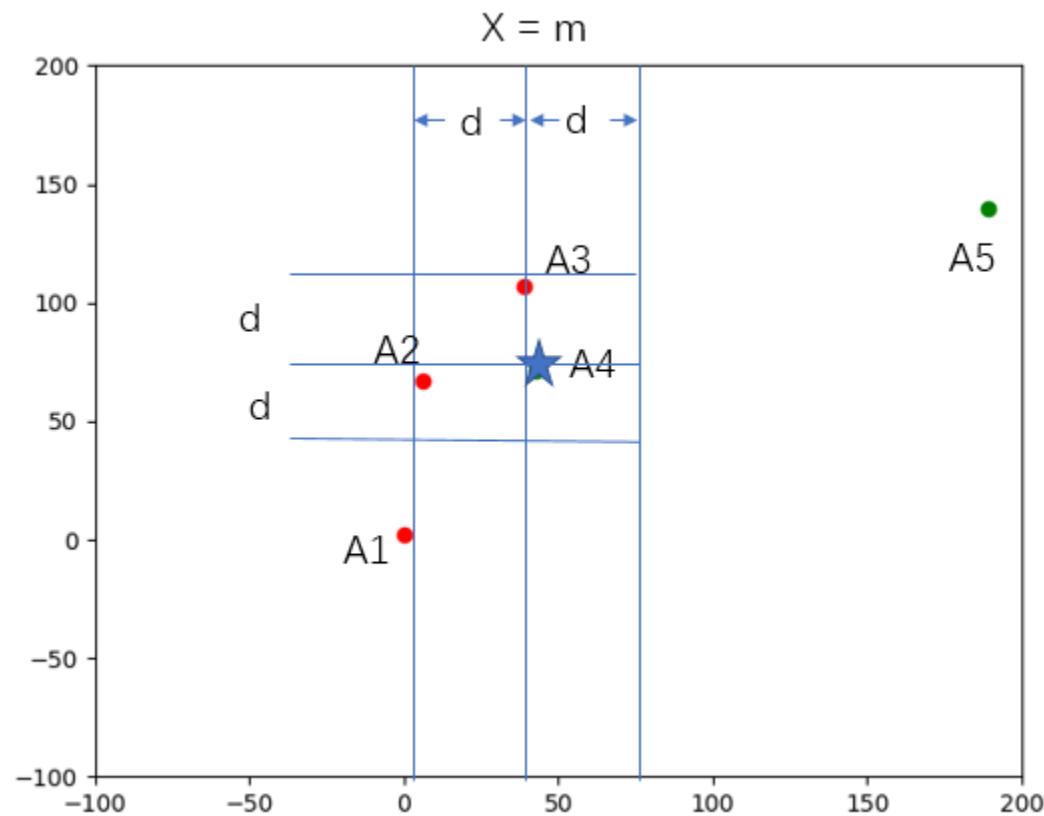
根据y坐标值从小到大排序，然后从下往上寻找在 $x \in [m-d, m+d]$ 范围内的点. $d = \min(d_1, d_2)$



对于在 $x \in [m-d, m+d]$ 范围内的每一个点（例如 $A_2(x_2, y_2)$ ），寻找在 $y \in [y_2-d, y_2+d]$ 范围内的点，并依次计算 A_2 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 $d = d'$



右侧同理，回到原问题同理求解



closest_pair(A)

If A.size<=1 return INF

sort_by_x(A)

m = A.size/2; x = A[m].first

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;

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