第三章

排序与分治算法

参考材料

《Introduction to Algorithm》

Chapter 6, 7, 8, 9

《Introduction to the Design and Analysis of Algorithm》

Chapter 4

HITWH
SE

提要

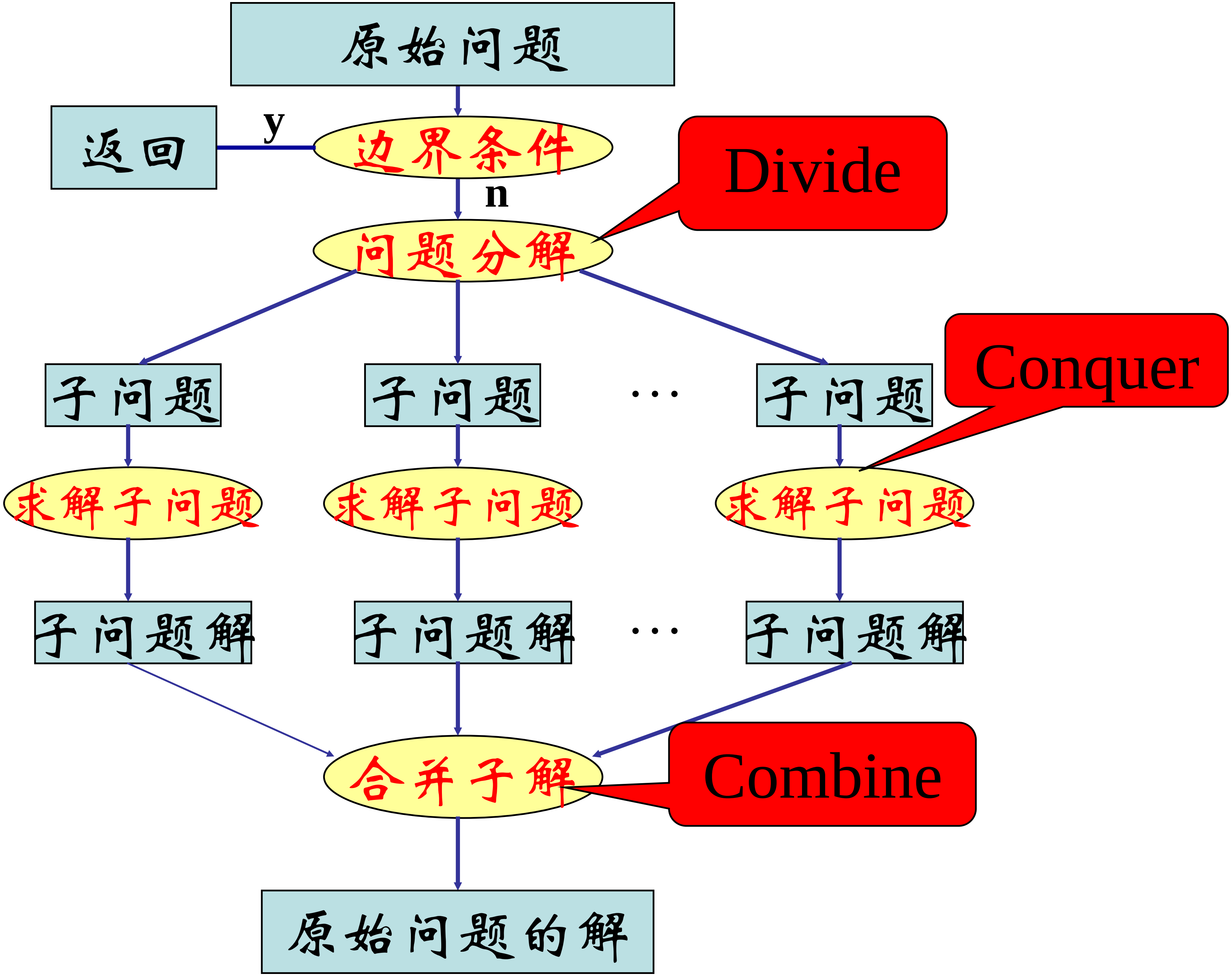
3.1 分治算法的原理
3.2 基于分治思想的排序算法
3.3 Medians and Order Statistics
3.4 最邻近点对
3.5 线性时间排序算法
3.6 凸包问题
3.7 FFT
3.8 整数乘法

3.1 Divide-and-Conquer原理

- 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
    - 注意：分解的这组子问题$p_1, p_2, \ldots p_m$未必一定是相同的子问题，即p_i和p_j可以是分别完成不同任务的子问题
  - 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



原始问题
返回
y
边界条件
n
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
合并子解
Combine
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=$C(n)$

最后得到递归方程：

- $T(n)=\theta(1)$ if $n\le c$
- $T(n)=aT(n/b)+D(n)+C(n)$ if $n>c$

举例：最大最小值问题

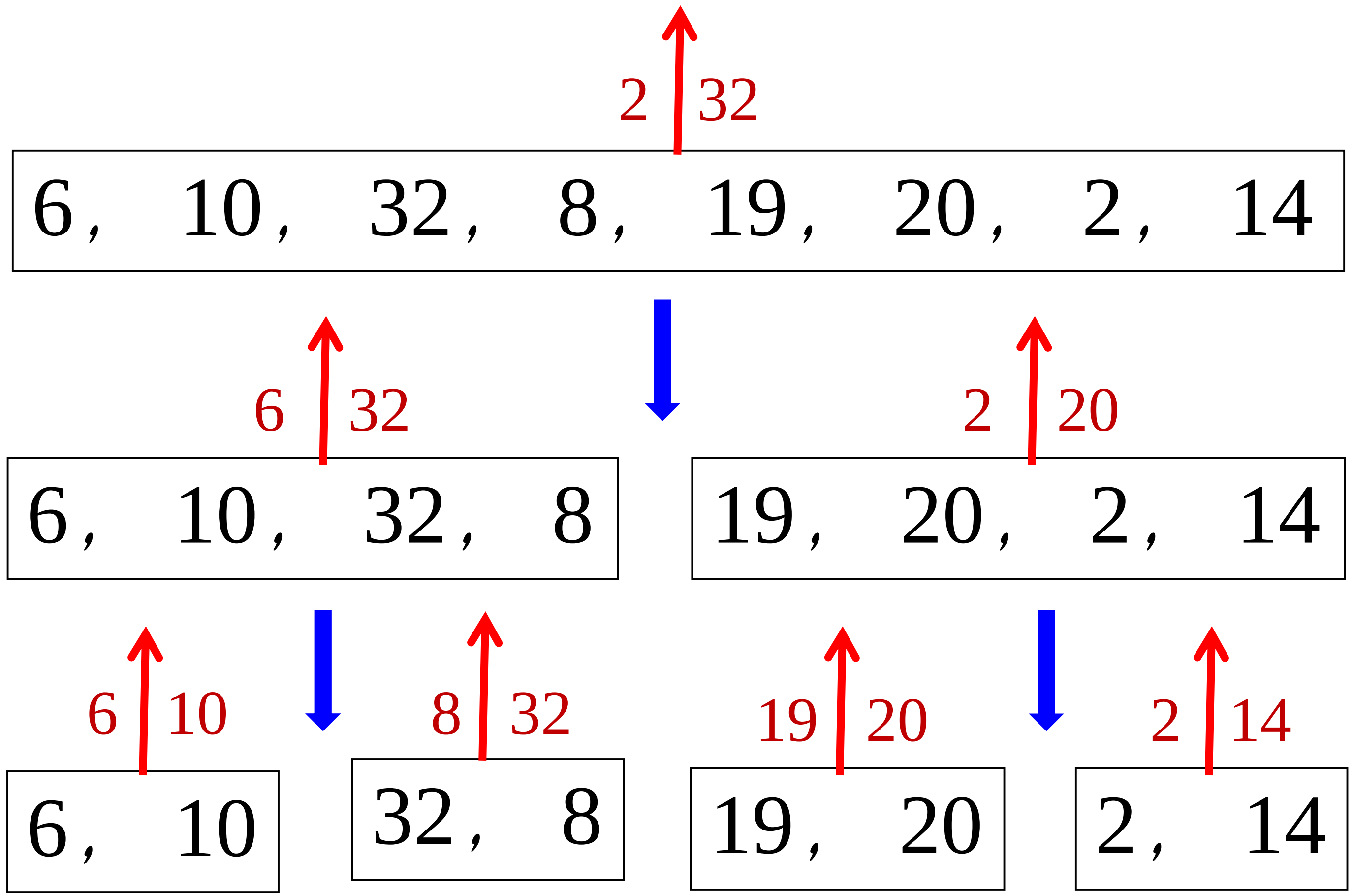
输入：数组$A[1,\ldots,n]$

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


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

算法MaxMin(A)

输入: 数组$A[i,\ldots,j]$

输出:数组$A[i,\ldots,j]$中的max和min

1. If j-i+1 =1 Then 输出$A[i]$,$A[i]$,算法结束
2. If j-i+1 =2 Then
3. If $A[i]$< A[j] Then 输出$A[i]$,$A[j]$; else 输出$A[j]$,$A[i]$; 算法结束
4. $k \leftarrow (j\text{-}i+1)/2$
5. $m_1,M_1 \leftarrow \mathrm{MaxMin}(A[i:k])$;
6. $m_2,M_2 \leftarrow \mathrm{MaxMin}(A[k+1:j])$;
7. $m \leftarrow \min(m_1,m_2)$;
8. $M \leftarrow \max(M_1,M_2)$;
9. 输出m,M

$$T(1)=0$$

$$T(2)=1$$

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

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

$$= \dots$$

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

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

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

$$=3n/2-2$$

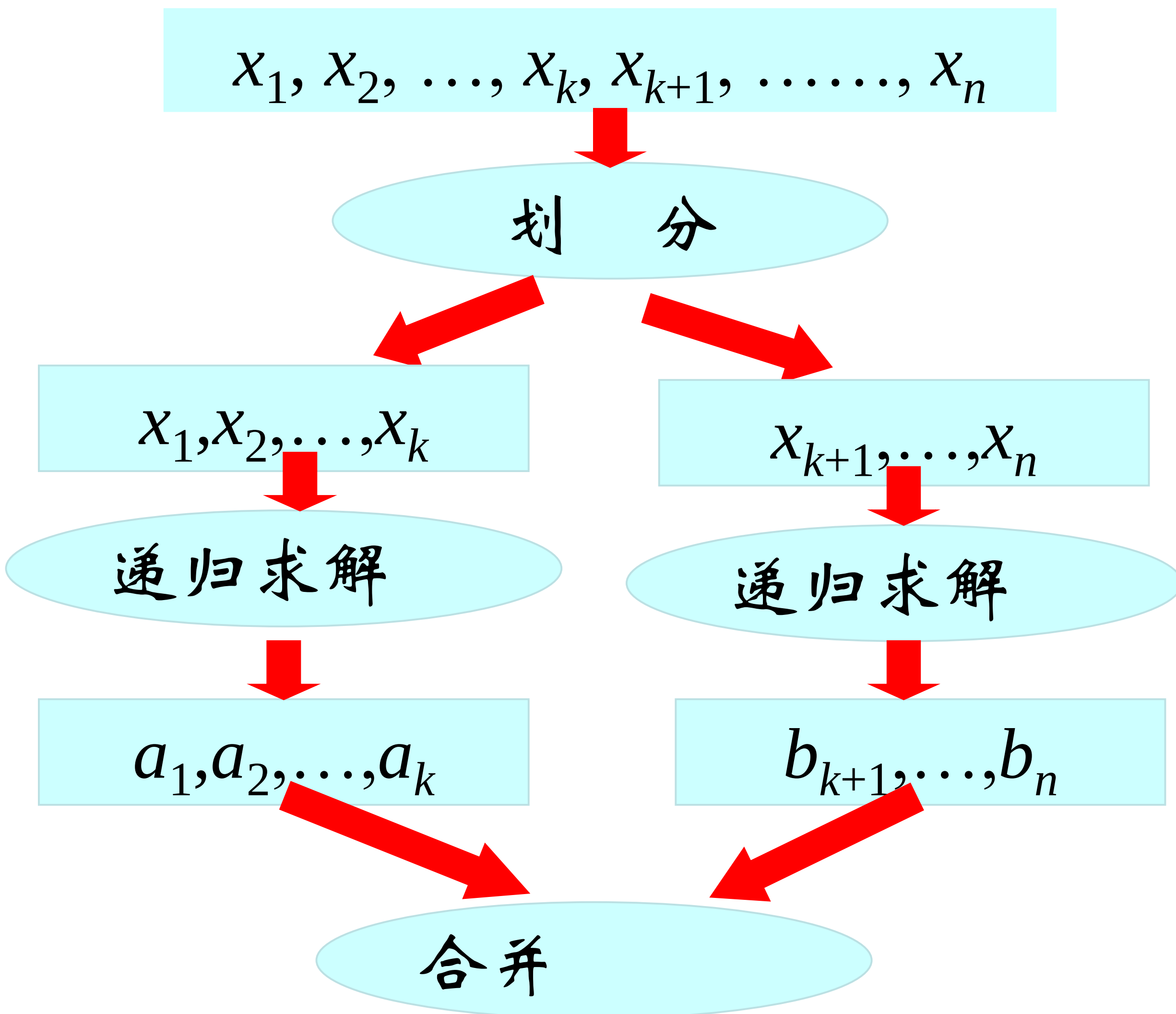
$n=2^k$

与Naïve算法相比，虽然同阶，但系数有所改进

3.2 基于分治的排序算法

- *Quicksort* Algorithm
- 排序问题的下界

基于分治思想的排序算法



划分的策略

根据某一策略将数据集合划分成两个部分

Mergesort：中间点

Quicksort：任选一个划分点x，利用x的值将数据划分成两部分

合并策略

不同的划分策略对应不同的合并策略

3.2.1 *Quicksort*

- Idea of *Quicksort*
- *Quicksort* Algorithm
- Correctness Proof
- Performance Analysis
- Randomized Quicksort Algorithms

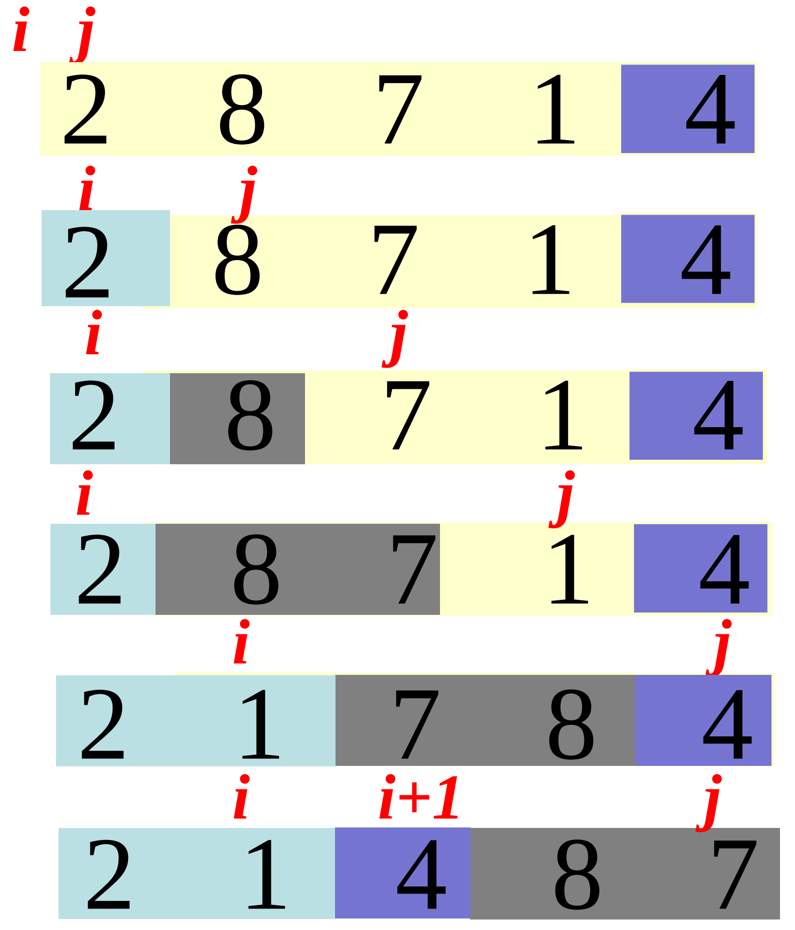
Idea of *Quicksort*

- Divide-and-Conquer
  - Divide:
    - Partition $A[p..r]$ into $A[p..q]$ and $A[q+1..r]$.

p		q-1	q	q+1		r

    - $\forall x \in A[p...q], x \le A[q], \ \forall y \in A[q+1...r], y > A[q]$.
    - q is generated by partition algorithm.
  - Conquer：
    - Sort $A[p...q\text{-}1]$ and $A[q+1...r]$ using quicksort recursively
  - Combine：
    - Since $A[p...q\text{-}1]$ and $A[q+1...r]$ have been sorted, nothing to do

- 划分 *A[p..r]*
  - 选择元素*x*作为划分点， *x*=*A*[*r*]
  - *x*逐一与其它元素作比较



算法执行过程中，
A 被分成4个区域

$$\text{Partition}(A, p, r)$$

$x \leftarrow A[r];$

$i \leftarrow p - 1;$

for $j \leftarrow p$ to $r - 1$

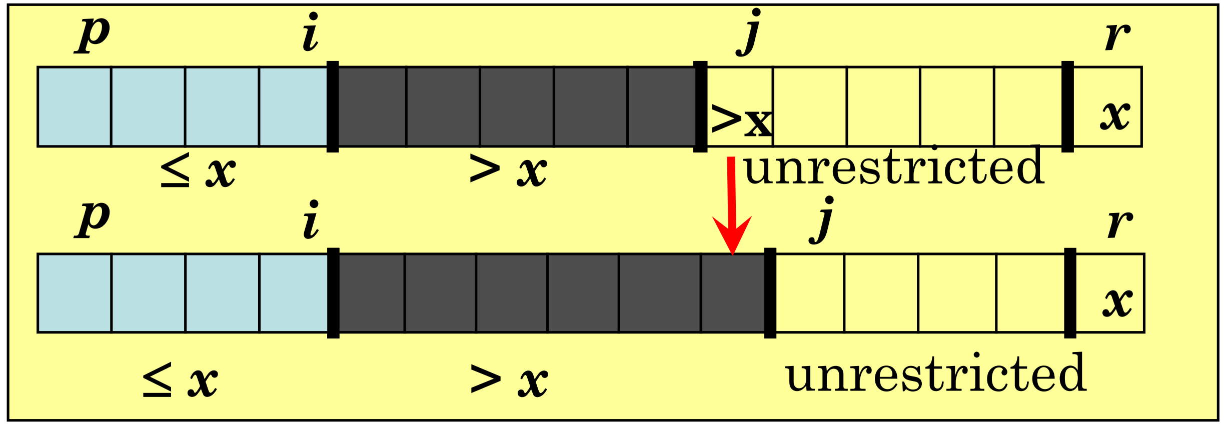
do if $A[j] \le x$

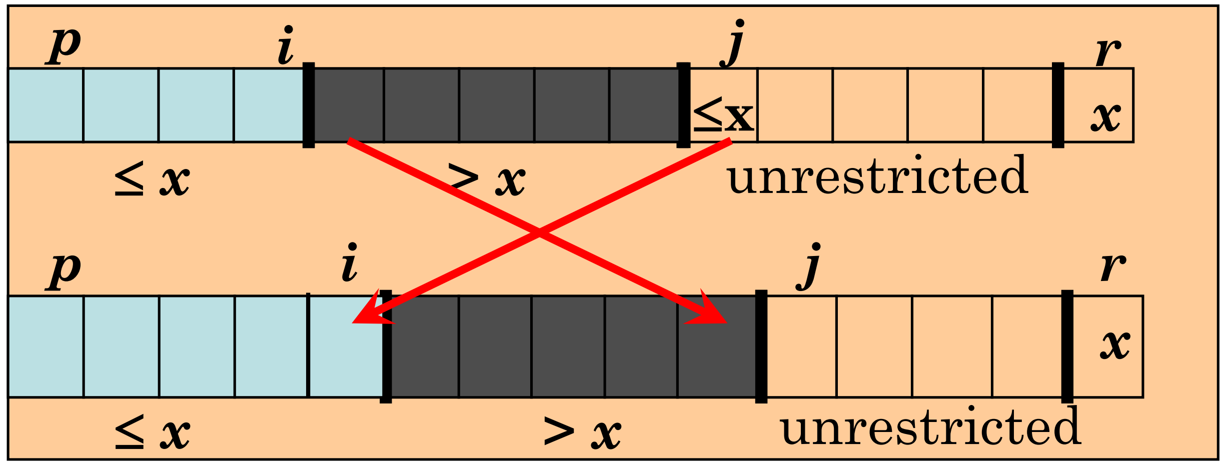
$i \leftarrow i + 1;$

exchange $A[i] \leftrightarrow A[j];$

exchange $A[i+1] \leftrightarrow A[r];$

return $i + 1;$





Running time: $\Theta(n)$

Quicksort Algorithm

Quicksort(A,p,r)

If $p<r$

Then q=Partition(A, p, r);

Quicksort (A,p,q-1);

Quicksort (A,q+1,r);

Correctness Proof

- Loop Invariant(循环不变量方法)

证明主要结构是循环结构的算法的正确性

循环不变量：数据或数据结构的关键性质

依赖于具体的算法和算法特点

证明分三个阶段

(1) 初始阶段：循环开始前循环不变量成立

(2) 循环阶段：循环体每执行一次,循环不变量成立

(3) 终止阶段：算法结束后，循环不变量保证算法正确

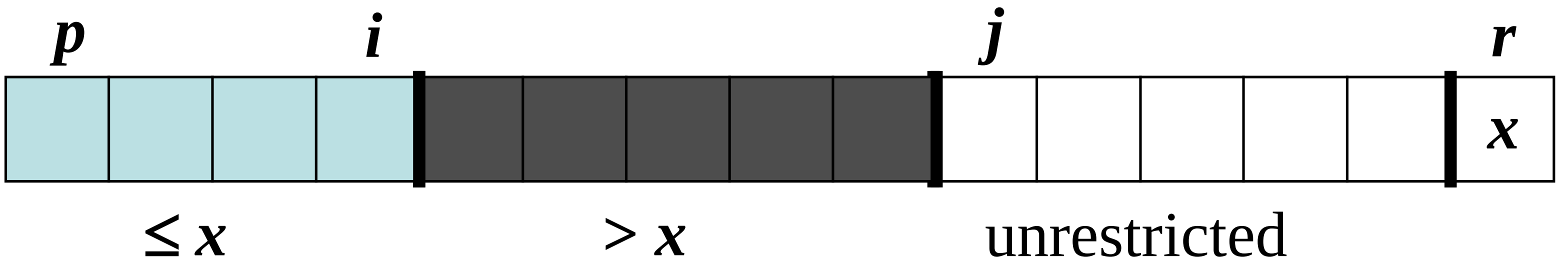
- Correctness Proof

```
    Partition(A, p, r)
    x ← A[r];
    i ← p − 1;
(3) for j ← p to r − 1
(4)     do if A[j] ≤ x
(5)         i ← i + 1;
(6)         exchange A[i] ↔ A[j];
    exchange A[i + 1] ↔ A[r];
    return i + 1;
```

定义循环不变量：

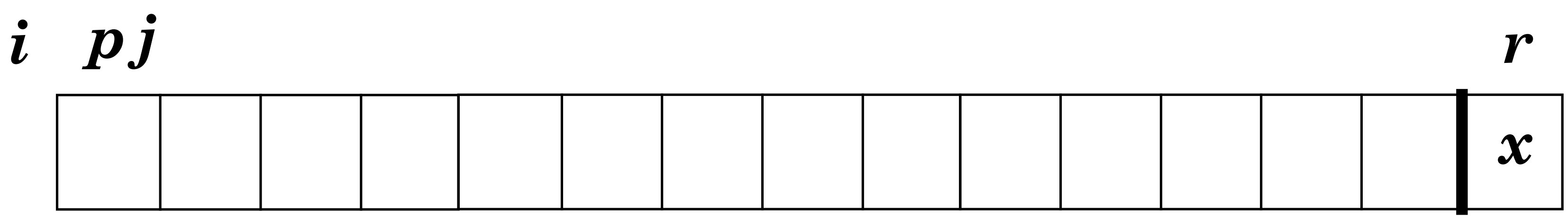
At the start of the loop of lines 3-6, for any k

1. *if $p \le k \le i$, then $A[k] \le x$.*
2. *if $i+1 \le k \le j-1$, then $A[k] > x$.*
3. *if $k=r$, then $A[k]=x$.*



– 初始阶段：$j=p$

算法迭代前：$i=p-1, j=p$, 条件 1 和 2 为真. 算法第1行使得条件 3 为真.

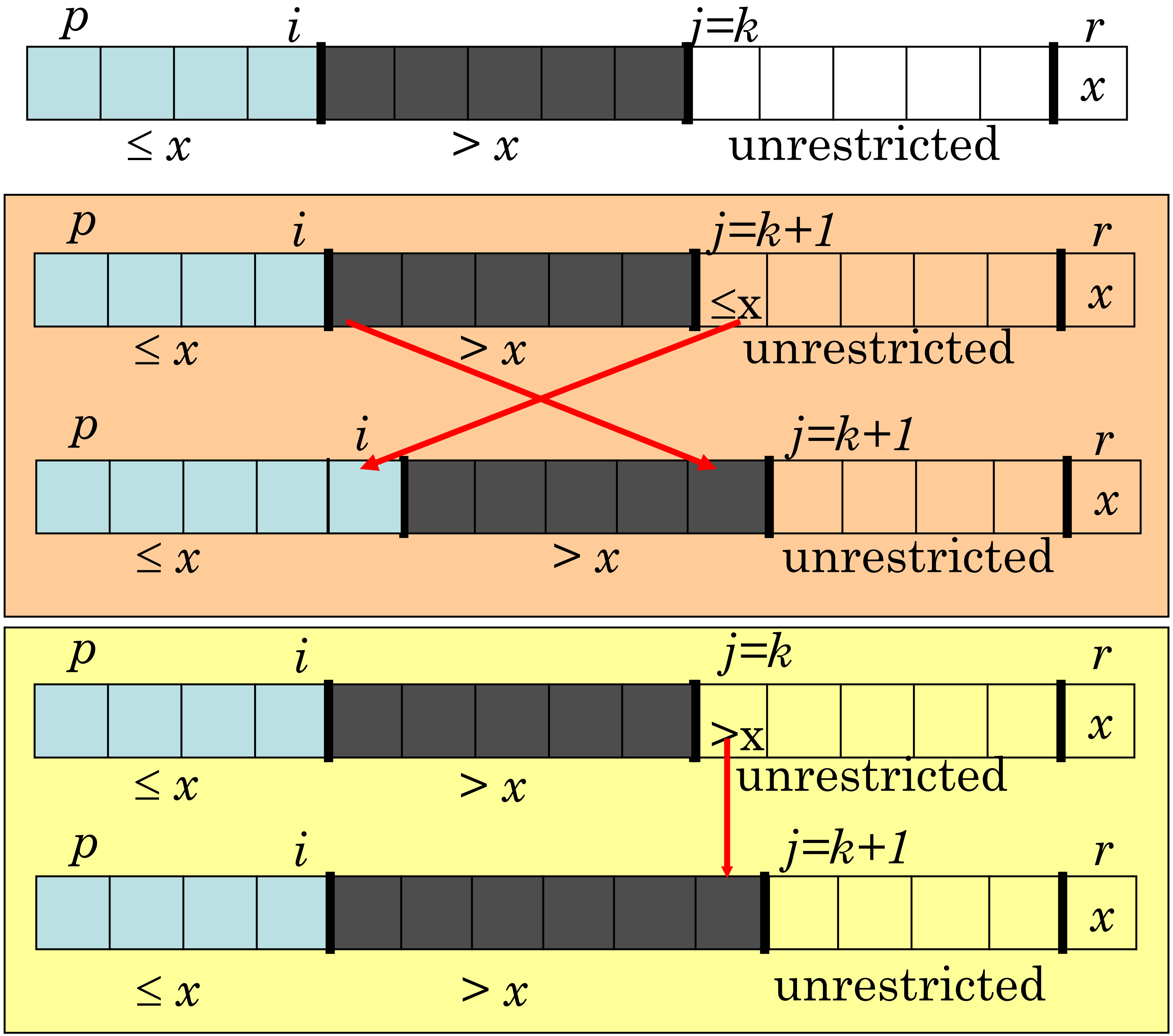


—保持阶段

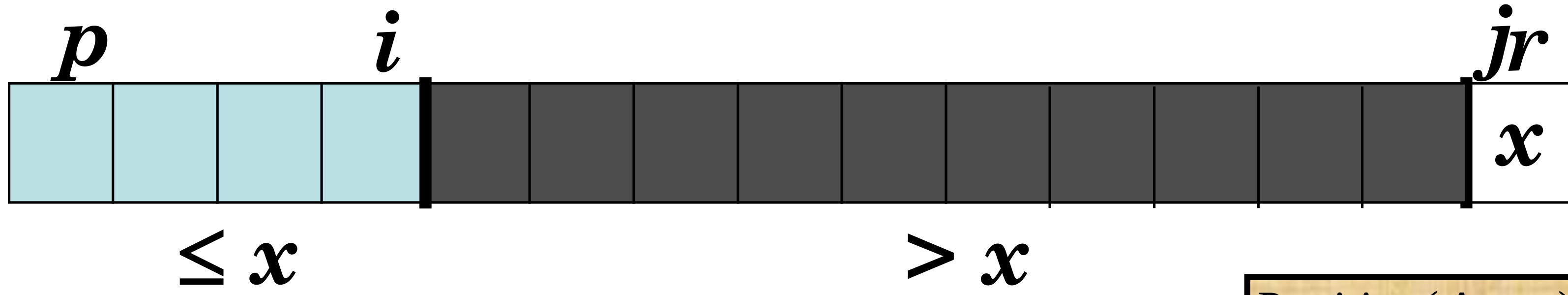
设$j=k$时循环不变量成立.

往证$j=k+1$时不变量成立.

```
Partition(A, p, r)
  x ← A[r];
  i ← p−1;
  for j ← p to r−1
    do if A[j] ≤ x
         i ← i+1;
         exchange A[i] ↔ A[j];
  exchange A[i+1] ↔ A[r];
  return i+1;
```



– 终止阶段



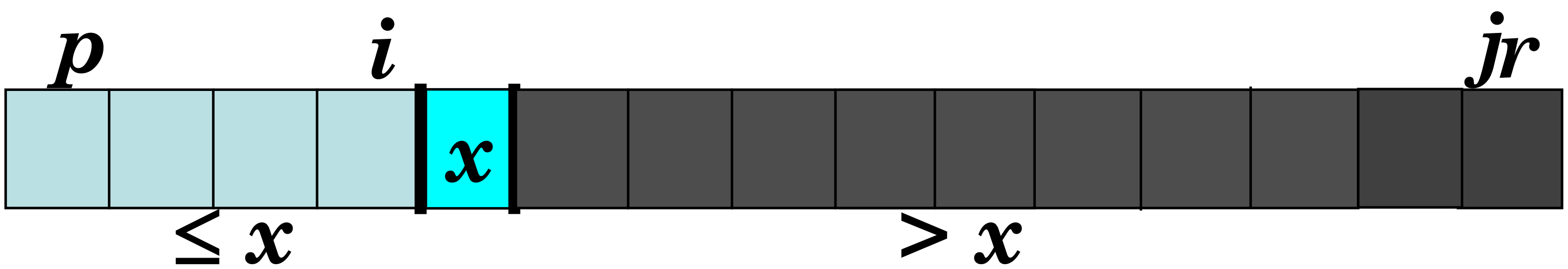
算法结束时, $j=r$, 产生三个集合:

1. 所有小于等于x的元素构成的集合.
2. 所有大于x的元素构成的集合.
3. 由元素x构成的集合.

Partition(A, p, r)
  $x \leftarrow A[r]$;
  $i \leftarrow p-1$;
  for $j \leftarrow p$ to $r-1$
    do if $A[j] \le x$
      $i \leftarrow i+1$;
      exchange $A[i] \leftrightarrow A[j]$;
  exchange $A[i+1] \leftrightarrow A[r]$;
  return $i+1$;

算法结束时

最后一个步骤将A[r]与A[i+1]互换.



Performance Analysis

- Time complexity of PARTITION: $\theta(n)$
- Best case time complexity of *Quicksort*
    - Array in partition into 2 equal sets
    - $T(n)=2T(n/2)+\theta(n)$
    - $T(n)=\theta(nlogn)$

Performance Analysis

- Worst case time complexity of Quicksort
  - Worst Case
    - $|A[p..q-1]|=0$, $|A[q+1..r]|=n-1$

    - The worst case happens in call to Partition Algorithm
  - Worst case time complexity
    - $T(n)= T(0)+T(n-1)+\theta(n)=T(n-1)+\theta(n)=\theta(n^2)$

Performance Analysis

What is the average time complexity?

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

Why?

Performance Analysis

- 假如第一次划分后产生两个子序列，第一个子序列包含s个元素，第二个子序列包含n-s个元素
- 一共有n种可能的划分，即$1 \le s \le n$，每种可能划分产生的概率为$1/n$
- 平均复杂性$\mathrm{T}(n)= \frac{1}{n}\sum_{s=1}^{n}(T(s)+T(n-s))+cn$

$$\frac{1}{n}\sum_{s=1}^{n}(T(s)+T(n-s)) = \frac{1}{n}(T(1)+T(n-1)+T(2)+T(n-2)+\ldots\ldots+T(n)+T(0))$$

由于$\mathrm{T}(0)=0$，有：$T(n)=\frac{1}{n}(2T(1)+2T(2)+\ldots\ldots+2T(n-1)+T(n))+cn$

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

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

Performance Analysis

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

用n=n-1代入上式，有：

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

两式相减： $$(n-1)T(n) - (n-2)T(n-1) = 2T(n-1) + c(2n-1)$$

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

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

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

递归地： $$\frac{T(n-1)}{n-1} = \frac{T(n-2)}{n-2} + c\left(\frac{1}{n-1} + \frac{1}{n-2}\right)$$

$$\cdots$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + c\left(\frac{1}{2} + \frac{1}{1}\right)$$

Performance Analysis

我们得到：

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

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

$$\cdots$$

$$\frac{T(2)}{2}=\frac{T(1)}{1}+c\left(\frac{1}{2}+\frac{1}{1}\right)$$

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

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

$$\frac{T(n)}{n}=c(H_n-1)+cH_{n-1}=c(H_n+H_{n-1}-1)=c(2H_n-\frac{1}{n}-1)=c(2H_n-\frac{n+1}{n})$$

$$T(n)=2cnH_n-c(n+1)=2cn\ln n-c(n+1)=O(n\log n)$$

HITWH
SE

Randomized Quicksort Algorithms

- **Randomized-Partition(A, p, r)**

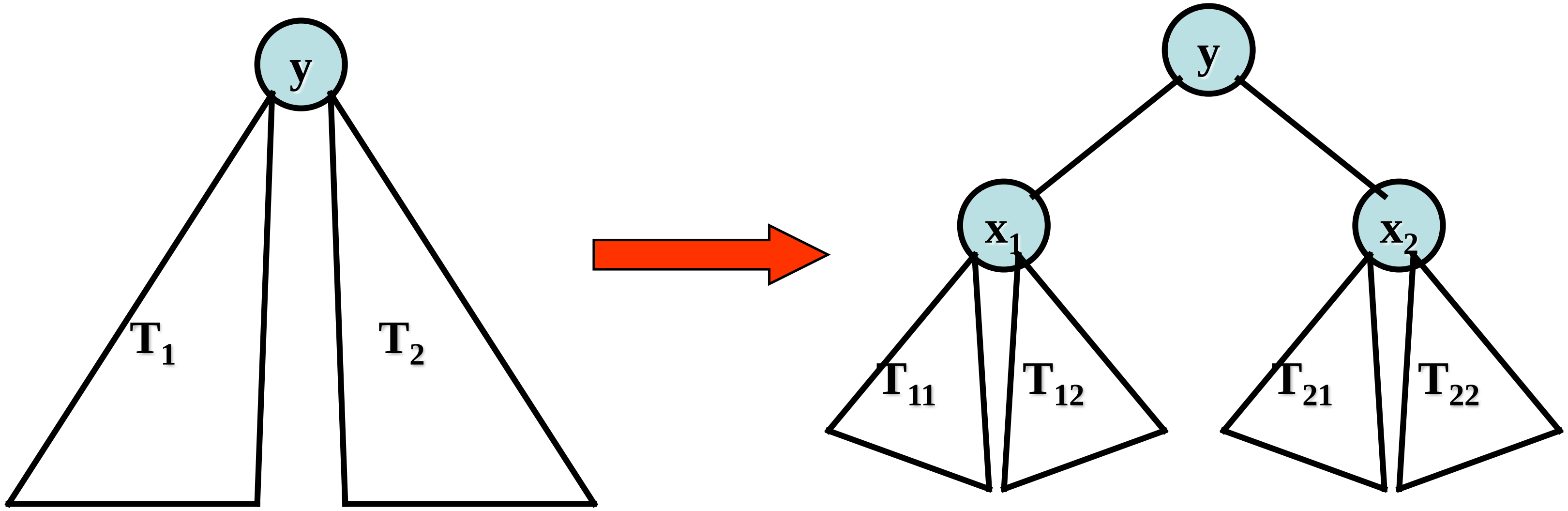
1. i := Random(p, r)
2. $A[r] \leftrightarrow A[i]$;
3. Return Partition(A, p, r)

- **Randomized-Quicksort(A, p, r)**

1. **If** $p < r$
2. **Then** q := Randomized-Partition(A, p, r);
3. Randomized-Quicksort(A, p, q-1);
4. Randomized-Quicksort(A, q+1, r).

随机快速排序复杂性分析

- 我们可以用树表示算法的计算过程



- 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

随机快速排序复杂性分析

- 基本概念
  - $x_{(i)}$表示A中Rank为i的元素(第i小元素)

    例如，$x_{(1)}$和$x_{(n)}$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$x_{(i)}$和$x_{(j)}$在运行中被比较，否则为0

    X_{ij}是$x_{(i)}$和$x_{(j)}$的比较次数
  - 算法的比较次数为$\sum_{i=1}^{n}\sum_{j>i}X_{ij}$
  - 算法的复杂性为$T(n)= E[\sum_{i=1}^{n}\sum_{j>i}X_{ij}]=\sum_{i=1}^{n}\sum_{j>i}E[X_{ij}]$

随机快速排序复杂性分析

$$T(n)=E[\sum_{i=1}^{n}\sum_{j>i}X_{ij}]=\sum_{i=1}^{n}\sum_{j>i}E[X_{ij}]$$

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

随机快速排序复杂性分析

- 求解p_{ij}
  - $Z_{ij}=\{x_{(i)}, x_{(i+1)}, \ldots, x_{(j)}\}$是$x_{(i)}$和$x_{(j)}$之间元素集合，$Z_{ij}$在同一棵子树时，$x_{(i)}$和$x_{(j)}$才可能比较.
  - $x_{(i)}$和$x_{(j)}$在执行中被比较，需满足下列条件：
    - $x_{(i)}$ 是Z_{ij}中第一个被选择的子树根节点，或者
    - $x_{(j)}$ 是Z_{ij}中第一个被选择的子树根节点
  - 一棵子树所有点等可能地被选为划分点，所以$x_{(i)}$或$x_{(j)}$被选为划分点的概率 = $2/|T|=2/(j-i+1)$.
  - $x_{(i)}$和$x_{(j)}$被进行比较的概率：

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

随机快速排序复杂性分析

- 现在我们有

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

$$=\sum_{i=1}^{n}\sum_{k=1}^{n-i}\frac{2}{k+1}\le\sum_{i=1}^{n}\sum_{k=1}^{n-i}\frac{2}{k}$$

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

定理. 随机排序算法的期望时间复杂性为O(nlogn)

3.2.2 排序问题的下界

问题的下界

- 问题的下界(lower bound of a problem)
  - 是用于解决该问题的任意算法所需要的最小时间复杂度
  - 问题难度的一种度量
    - 如果问题可由一个具有较低时间复杂性的算法解决，则该问题是简单的；否则是困难的
  - 通常指：最坏情况下界
- 问题的下界是**不唯一的**
  - 例如. $\Omega(1)$, $\Omega(n)$, $\Omega(n\log n)$ 都是排序的下界
  - 只有$\Omega(n\log n)$是有意义的
  - 下界应尽可能地高，达到上限
  - 下界的分析都是经过严格理论分析和证明，而非纯粹猜测

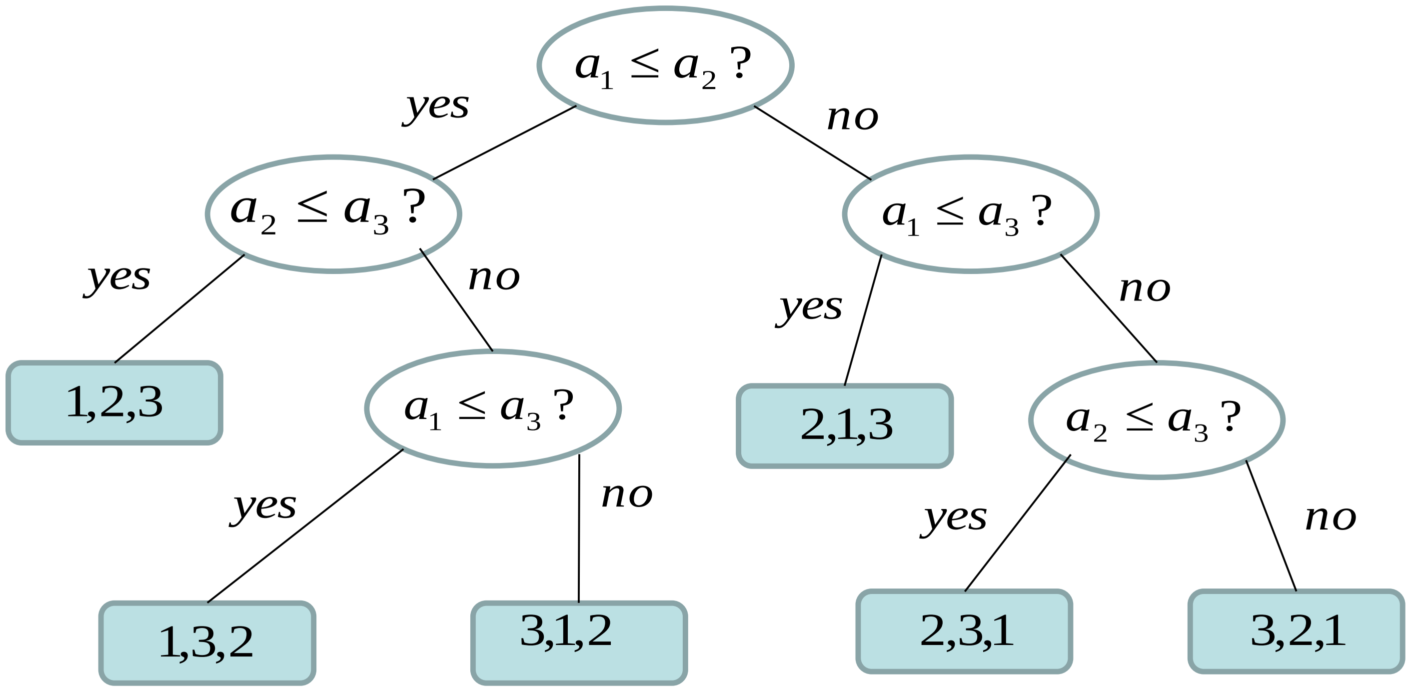
问题的下界的意义

- 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.
  - 下界和算法都是可能改进的.
- 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

排序的下界

- 通常，基本操作是比较和交换的排序算法可以用一个二叉决策树描述
  - 通过忽略比较以外的细节来抽象表示比较排序算法
  - 每个内节点表示一个比较操作$a_i \le a_j$；
  - 所有被排序元素的全排列是树的叶节点；

对于特定输入数据集的排序过程，对应于从树的根结点到叶子节点的一条路径



排序的下界

- n个元素有$n!$种不同排列
- 其排序过程对应于一个高度为h，具有$n!$个叶子节点的二叉决策树 .
- 由于高度为h的二叉树至多有2^h个叶子节点
- 则有$2^h \geq n!$

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

即： $h \geq \lg(n!) = \Omega(n\lg n)$

排序的下界是: $\Omega(n\log n)$

3.3 Medians and Order Statistics

- Decrease and Conquer原理
- Selection Problem

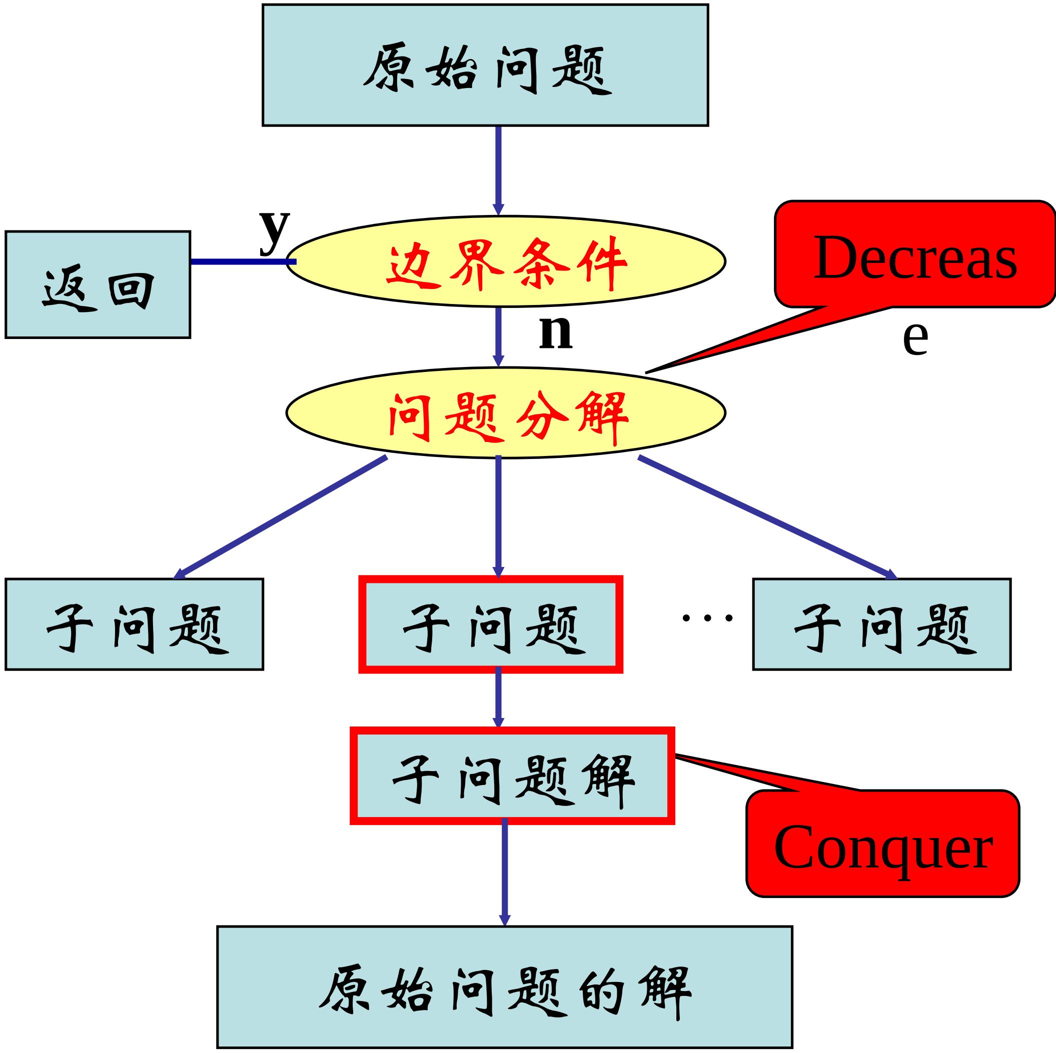
Decrease and Conquer原理

- 原始问题划分为若干子问题，将原始计算问题转化为其中某一个子问题的计算问题

  例如：折半查找

  $$T(n)=T(\lfloor n/2 \rfloor)+1$$

- 非常有效的一种方法，通常用于解决优化问题



Decrease and Conquer原理

- 与Divide-and Conquer的不同
  - 分治方法：递归求解每一个子问题，然后通过合并各个子问题的解最后得到原始问题的解
  - 减治方法：仅通过求解某一个子问题的解得到原始问题的解

Medians and Order Statistics

- **The i^{th} order statistic problem**
  - Input: set S of n (distinct) elements, and a number i.
  - Output: element x in S that is greater than exactly i-1 elements in S.
  - Special order statistics
    - The 1^{st} order statistic is the *minimum* in S
    - The n^{th} order statistic is the *maximum* in S
    - The *median* in S is at
      - $(n+1)/2$ when n is odd
      - $n/2$ and $n/2+1$ when n is even

Selection Problem

- **Problem**
  - Input: set A of n (distinct) elements, and a number k.
  - Output: element x in A that is greater than exactly k-1 elements in A, i.e the k^{th} smallest element.

The straightforward algorithm:

step 1: Sort the n elements

step 2: Locate the k^{th} element in the sorted list.

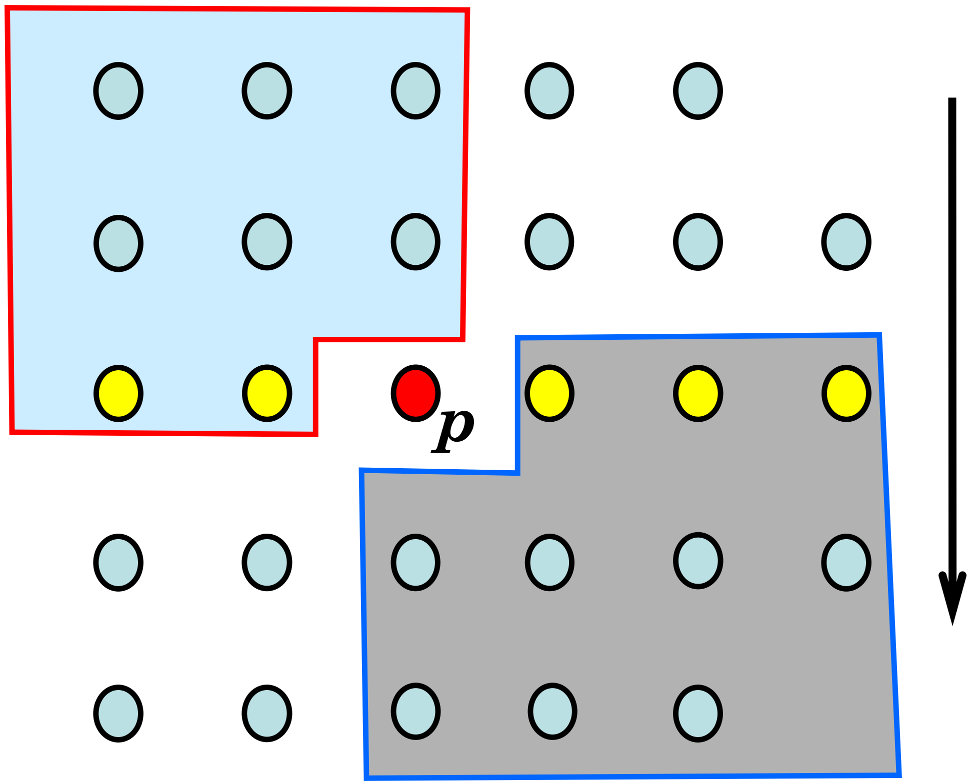
Time complexity: O($n$log$n$)

Algorithm of Selection Problem

- **Main Idea**
  - $S=\{a_1, a_2, \ldots, a_n\}$
  - Let $p \in S$, 用p 将S 划分为3 个子集合S_1 , S_2 , S_3:
    - $S_1=\{ a_i \mid a_i < p , 1 \le i \le n\}$
    - $S_2=\{ a_i \mid a_i = p , 1 \le i \le n\}$
    - $S_3=\{ a_i \mid a_i > p , 1 \le i \le n\}$
  - 3 种情况:
    - 若$|S_1| > k$, 则在集合S_1中搜索第k小的元素.
    - 否则, 若$|S_1| + |S_2| > k$, 则 p 是S中第k小的元素.
    - 否则, 在S_3中搜索第$(k - |S_1| - |S_2|)$小的元素.

Algorithm of Selection Problem

- 如何选择*p*?
  - The *n* elements are divided into $\left\lceil \frac{n}{5} \right\rceil$ subsets (Each subset has 5 elements.)
  - Sort each subset
  - Find the element ***p*** which is the median of the medians of the $\lceil n/5 \rceil$ subsets



HITWH
SE

Algorithm of Selection Problem

算法步骤：

$O(n)$

Step 1: 划分 S 为 $\lceil n/5 \rceil$ 个组. 每组包含5个元素. 若最后一组不足5个元素，则用 ∞ 补足.

Step 2: 排序每组5个元素，并确定每一分组的中位数. $O(n)$

Step 3: 计算 $\lceil n/5 \rceil$ 个中位数的中位数 p. **T(n/5)**

Step 4: p 将 S 划分为三个子集合 S_1, S_2 及 S_3, S_1 中元素均小于 p，S_2 中元素均等于 p，S_3 中元素均大于 p. $O(n)$

Step 5: 若 $|S_1| \geq k$, 则递归地在 S_1 中搜索第 k 小元素;

否则，若 $|S_1| + |S_2| \geq k$，则 p 即为 S 中第 k 小元素;

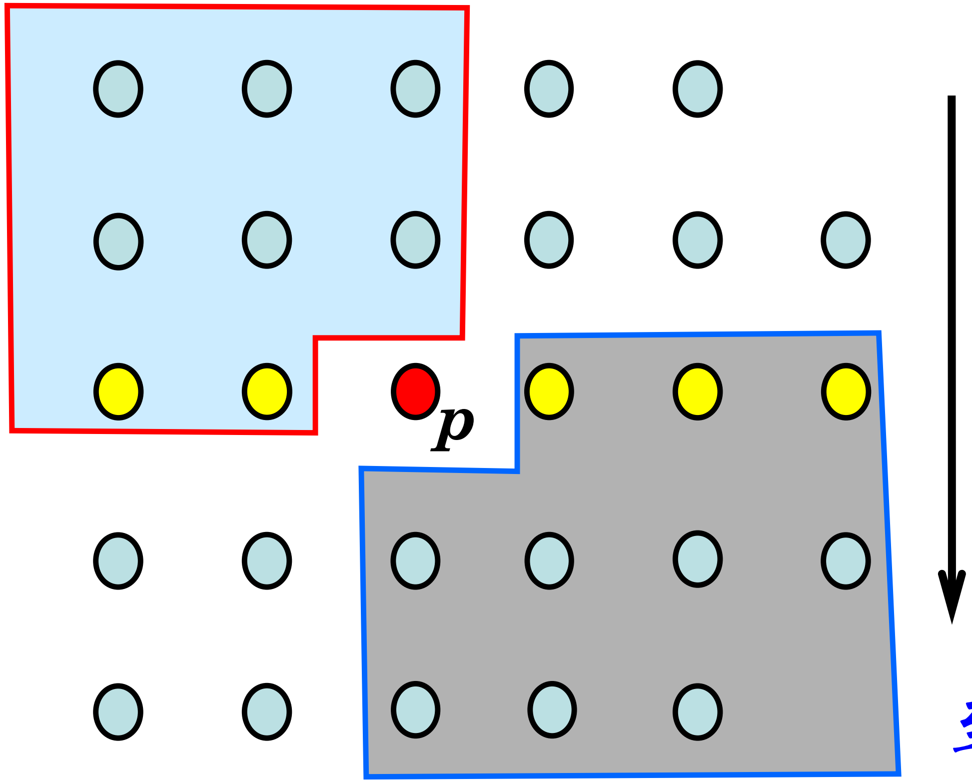
否则，令 $k' = k - |S_1| - |S_2|$，递归地在 S_3 中搜索第 k' 小元素.

?

Performance Analysis

Each 5-element subset is sorted in non-decreasing sequence.

至少|*S*|/4 个元素小于等于*p*



至少|*S*|/4 个元素大于等于*p*

Algorithm of Selection Problem

算法步骤:

$O(n)$

Step 1: 划分 S 为 $\lceil n/5 \rceil$ 个组. 每组包含5个元素. 若最后一组不足5个元素，则用 ∞ 补足.

Step 2: 排序每组5个元素，并确定每一分组的中位数. $O(n)$

Step 3: 计算 $\lceil n/5 \rceil$ 个中位数的中位数 p. **T(n/5)**

Step 4: p 将 S 划分为三个子集合 S_1, S_2 及 S_3, S_1 中元素均小于 p， S_2 中元素均等于 p， S_3 中元素均大于 p.

$O(n)$

Step 5: 若 $|S_1| \geq k$, 则递归地在 S_1 中搜索第 k 小元素;

否则， 若 $|S_1| + |S_2| \geq k$， 则 p 即为 S 中第 k 小元素;

否则， 令 $k' = k - |S_1| - |S_2|$， 递归地在 S_3 中搜索第 k' 小元素.

T(3n/4)

Performance Analysis

- 算法复杂性分析

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

Let $T(n) = a_0 + a_1 n + a_2 n^2 + \ldots \ , a_1 \neq 0$

$T(3n/4) = a_0 + (3/4)a_1 n + (9/16)a_2 n^2 + \ldots$

$T(n/5) = a_0 + (1/5)a_1 n + (1/25)a_2 n^2 + \ldots$

$T(3n/4 + n/5) = T(19n/20) = a_0 + (19/20)a_1 n + (361/400)a_2 n^2 + \ldots$

$T(3n/4) + T(n/5) = a_0 + a_0 +(19/20)a_1 n + (241/400)a_2 n^2 + \ldots$

$\leq a_0 + T(19n/20)$

$\Rightarrow T(n) \leq cn + T(19n/20)$

Performance Analysis

$$\Rightarrow T(n) \le cn + T(19n/20)$$

$$\le cn + (19/20)cn + \mathrm{T}((19/20)^2 n)$$

$$\vdots$$

$$\le cn + (19/20)cn + (19/20)^2 cn + \ldots + (19/20)^p cn + T((19/20)^{p+1} n) \ ,$$

$$(19/20)^{p+1} n \le 1 \le (19/20)^p n$$

$$= cn\,(1 + 19/20 + (19/20)^2 + \ldots + (19/20)^p)$$

$$= \frac{1-(\frac{19}{20})^{p+1}}{1-\frac{19}{20}} cn + b$$

$$\le 20\,cn + b$$

$$= O(n)$$

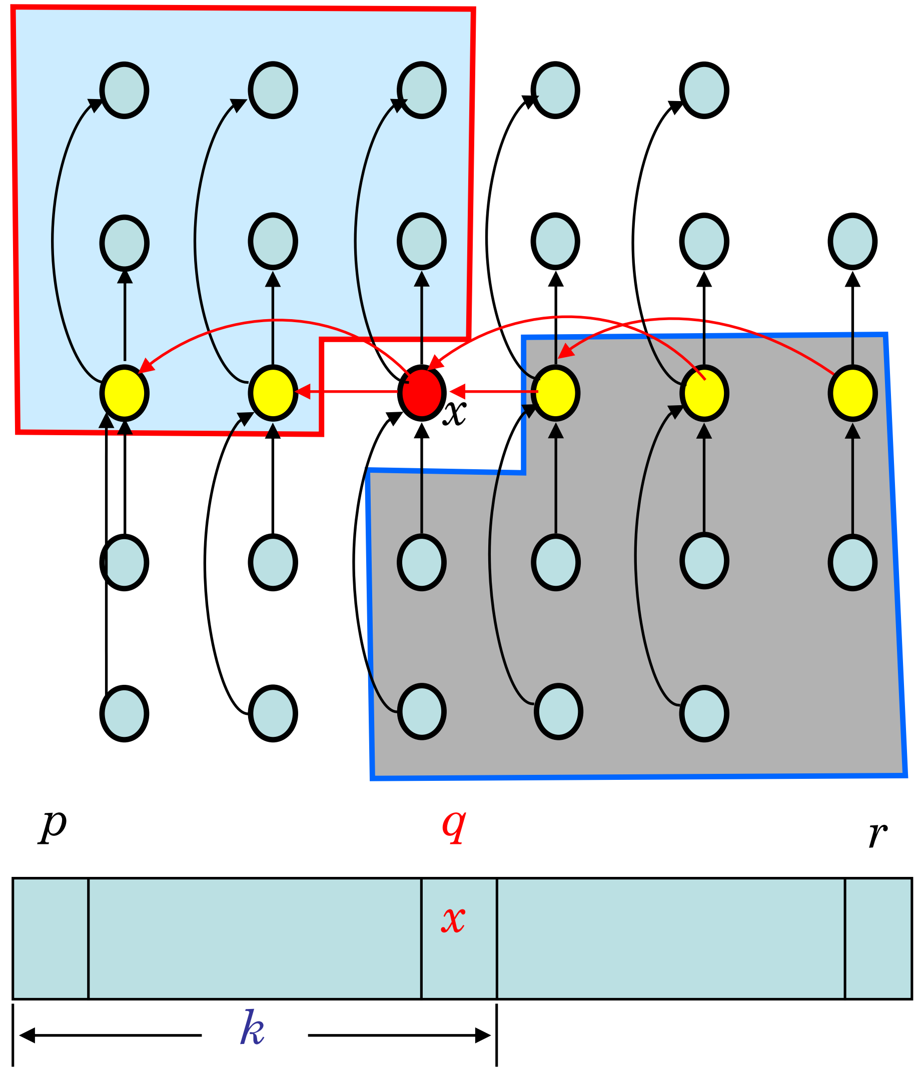
- **Time complexity analysis**
  - The number of elements that greater than the partition element *x* is at least

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil\right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

  - Thus, in the worst case, the number of elements that great than the *x* is at most

    n-((3n/10)-6) =7n/10+6.
  - Similarly, the number of elements that less than the *x* is also at most

    7n/10+6



1. Divide n elements in A into $\lceil n/5 \rceil$ groups of 5 elements each, at most one group has (n mod 5) elements. $O(n)$

2. Find median of each group by sorting first. $O(n)$

3. Use Select recursively to find the median x of the $\lceil n/5 \rceil$ medians. In case of having two medians, take the lower. $T(\lceil n/5 \rceil)$

4. Exchange x with the last element in A and apply Partition subroutine. Let k be the number of elements on the low side of the partition including x. $O(n)$

5. If $i = k$, return x. Otherwise, use Select recursively to find the i^{th} smallest element on the low side if $i \le k$, or the $(i - k)^{\text{th}}$ smallest element on the high side if $i > k$. $T(7n/10 + 6)$

$$T(n) \le \begin{cases} \theta(1) & \text{if } n \le 140 \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{if } n > 140 \end{cases}$$

- **Now we have**

$$T(n) \le \begin{cases} \theta(1) & \text{if } \; n \le 140 \\ T(\lceil n/5 \rceil) + T(7n/10+6) + O(n) & \text{if } \; n > 140 \end{cases}$$

- **Using inductive method, we can prove *T(n)≤cn* for some *c* and *n*>140.**
- **Thus, the worst case time complexity is *T(n)=O(n)*.**

$$T(n)=\theta(1) \quad \text{if } n\le c$$
$$T(n)=aT(n/b)+D(n)+C(n) \quad \text{if } n>c$$

3.4 Finding the closest pair of points

优化combine阶段，降低$T(n)=aT(n/b)+f(n)$中的$f(n)$

问题定义

输入：Euclidean空间上的n个点的集合Q

输出： $A, B \in Q$,

$Dis(A, B)=Min\{Dis(P_i, P_j) \mid P_i, P_j \in Q\}$

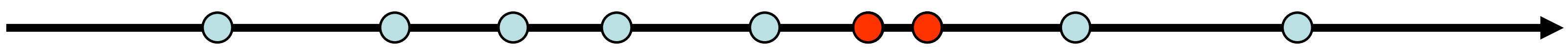
$Dis(P_i, P_j)$是Euclidean距离：

如果$P_i=(x_i, y_i)$, $P_j=(x_j, y_j)$，则

$$Dis(P_i,P_j)=\sqrt{(x_i-x_j)^2+(y_i-y_j)^2}$$

一维空间算法

- 利用排序的算法
  - 算法
    - 把Q中的点排序



    - 通过有序集合的线性扫描找出最近点对
  - 时间复杂性
    - $T(n)=O(n\log n)$

一维空间算法(续)

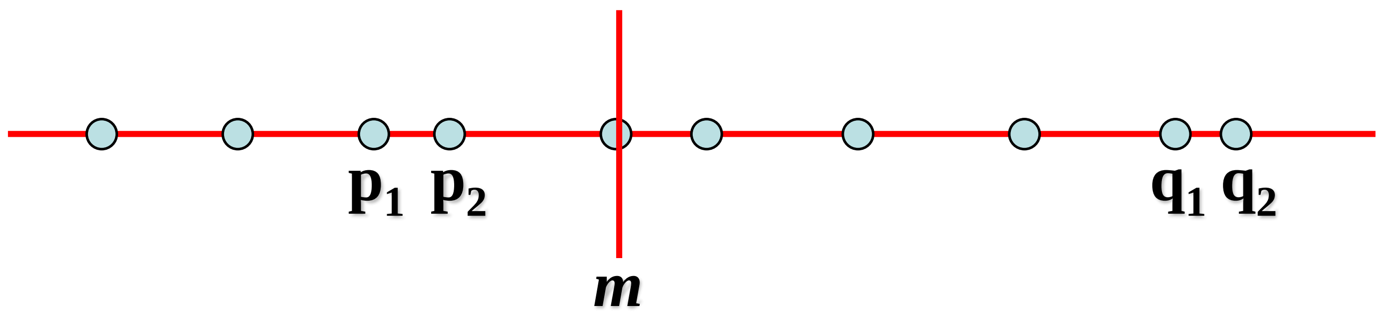
- Divide-and-conquer算法

边界条件:

1. 如果Q中仅包含2个点，则返回这个点对;

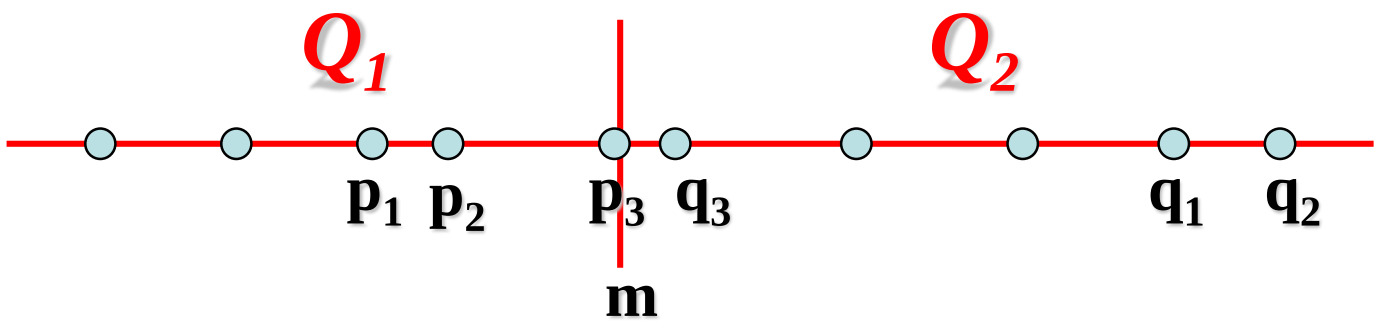
Divide:

2. 求Q中点的中位数m;



HITWH
SE

3. 用Q中点坐标中位数m把Q划分为两个大小相等的子集合$Q_1=\{x\in Q \mid x\le m\}$，$Q_2=\{x\in Q \mid x>m\}$

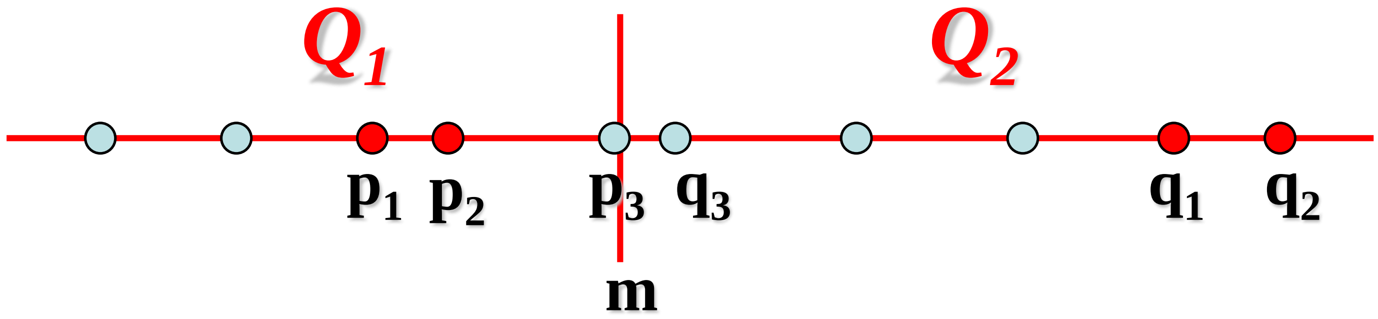


Conquer:

4. 递归地在Q_1和Q_2中找出最接近点对

$$(p_1, p_2)\text{和}(q_1, q_2)$$

Merge:



5. 在(p_1, p_2)、(q_1, q_2)和某个(p_3, q_3)之间选择最接近点对(x, y)，其中 p_3是Q_1中最大点，q_3是Q_2中最小点。

(x, y)是Q中最接近点对

- 时间复杂性
  - Divide阶段需要$O(n)$时间
  - Conquer阶段需要$2T(n/2)$时间
  - Merge阶段需要$O(n)$时间
  - 递归方程

$$T(n) = O(1) \qquad n = 2$$

$$T(n) = 2T(n/2) + O(n) \qquad n \geq 3$$

  - 用Master定理求解$T(n)$

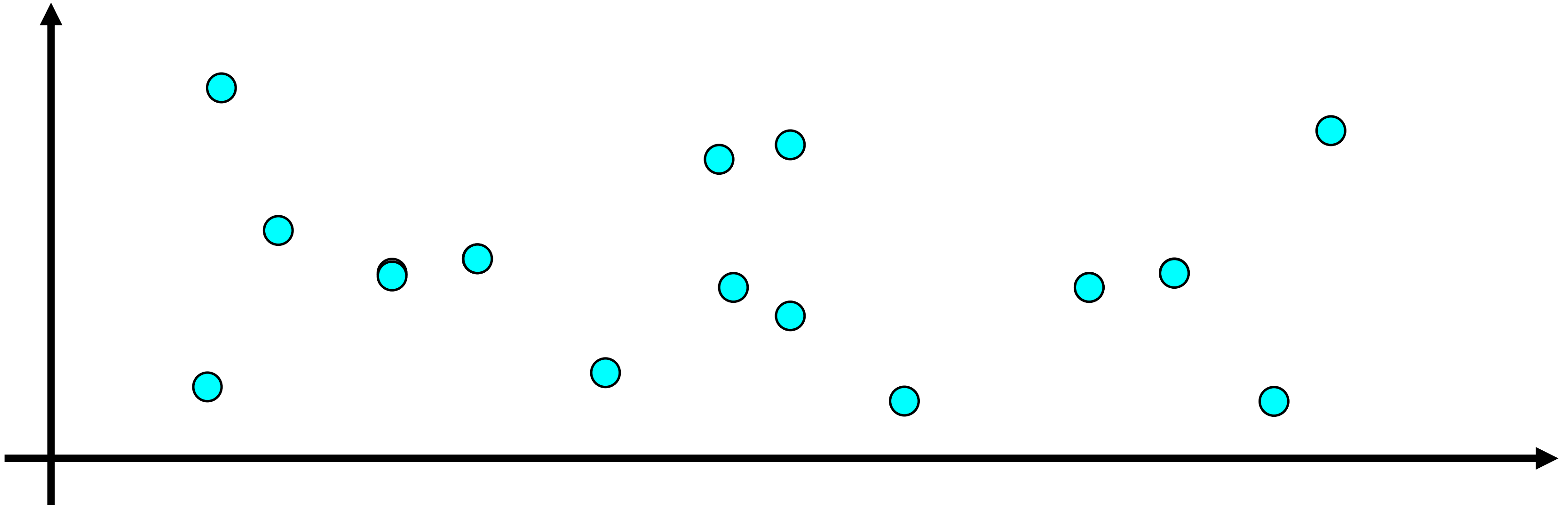
$$T(n) = O(n\log n)$$

- Divide-and-conquer算法

Assume: Q中点已经分别按x坐标和y坐标排序后存储在X和Y中.

边界 条件:

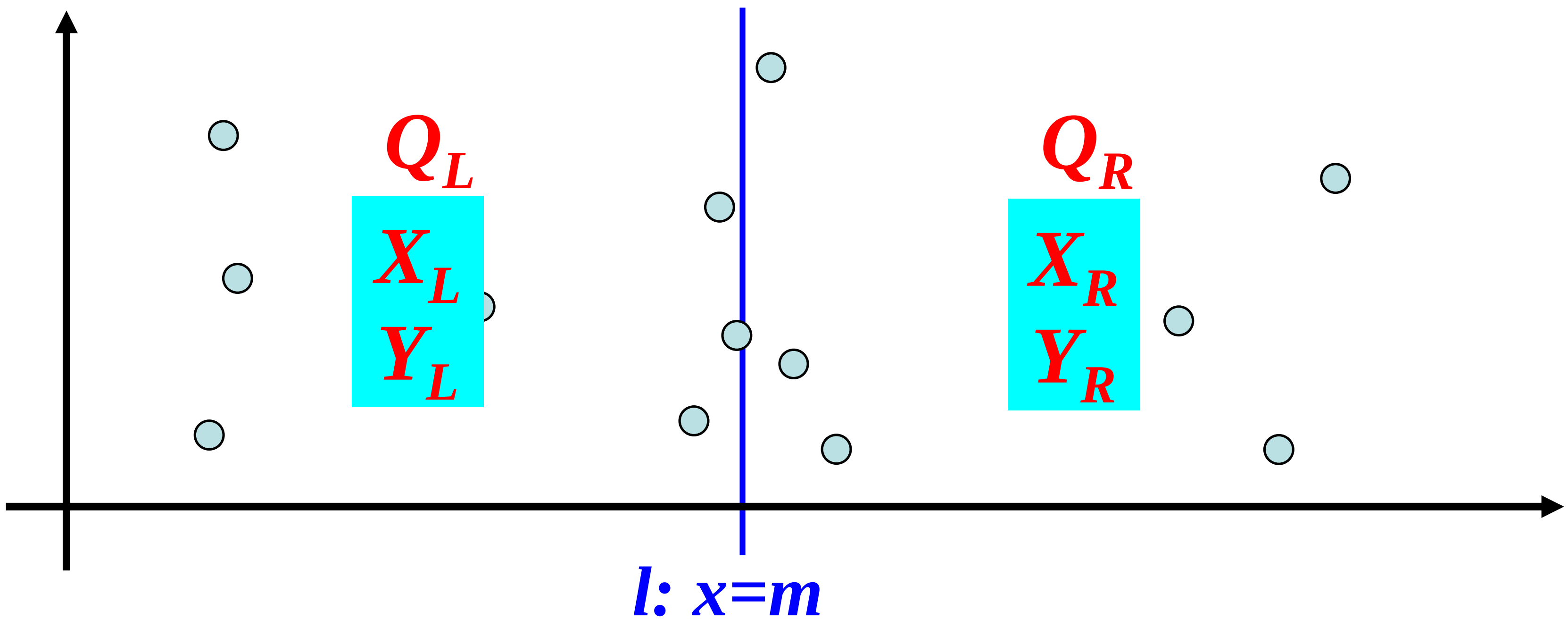
1. 如果Q中仅包含3个点,则返回最近点对,结束;

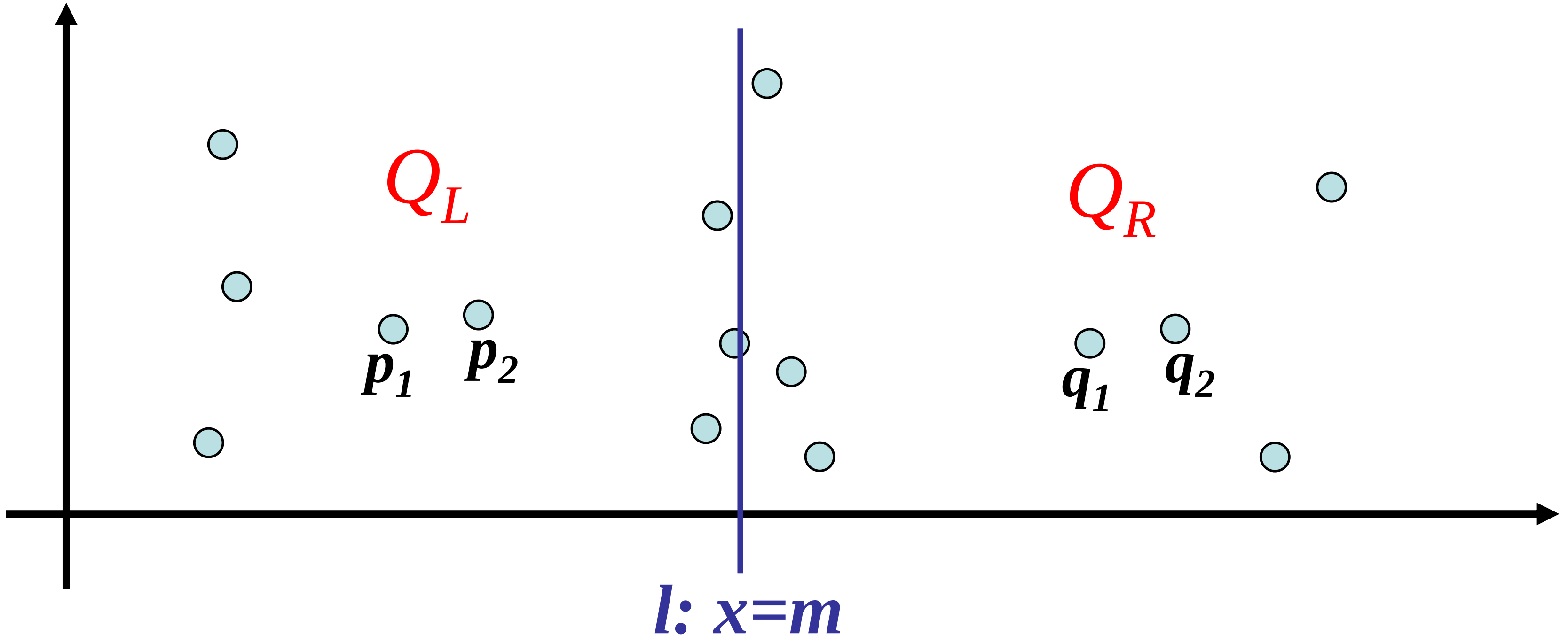


HITWH
SE

Divide:

2. 计算Q中各点x-坐标的中位数m；
3. 用垂线l: $x=m$把Q划分成两个大小相等的子集合Q_L和Q_R，Q_L中点在l左边，Q_R中点在l右边；
4. 把X划分为X_L和X_R；把Y划分为Y_L和Y_R；



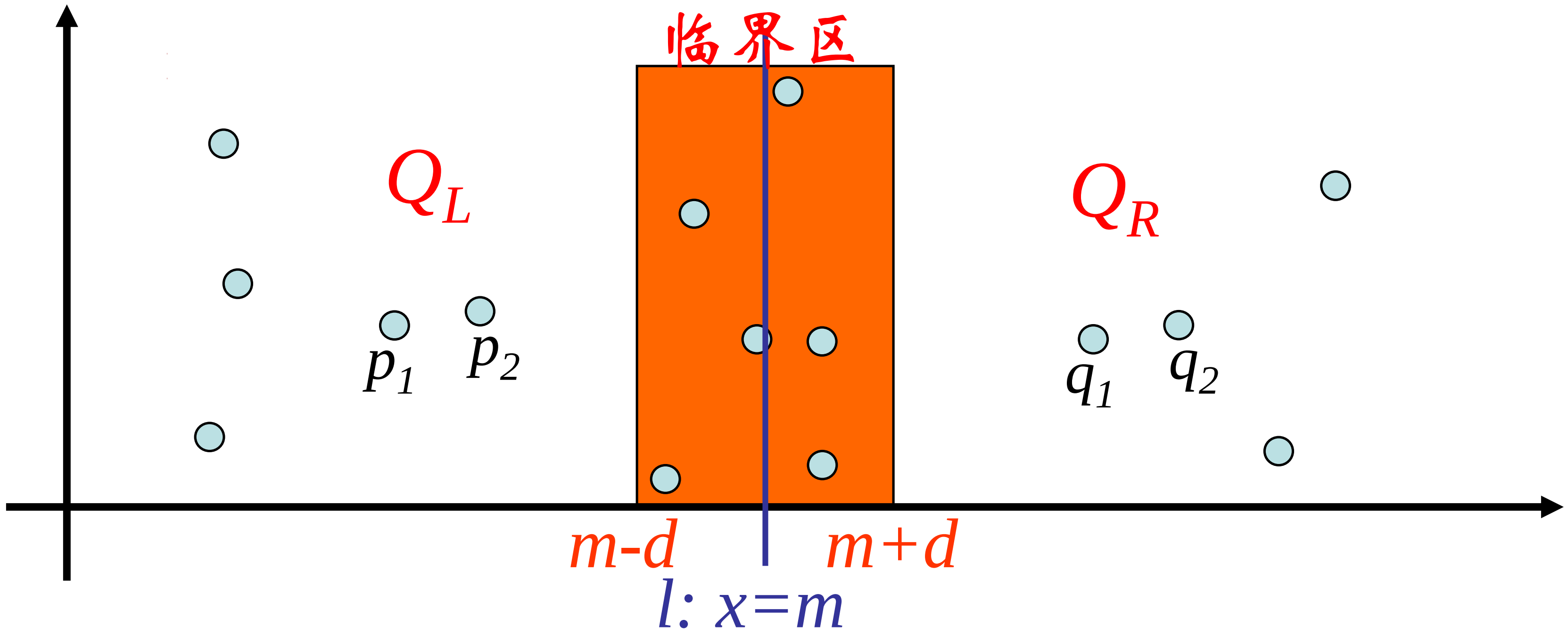


Conquer:

5. 递归地在Q_L、Q_R中找出最近点对:

$$(p_1, p_2) \in Q_L, (q_1, q_2) \in Q_R$$

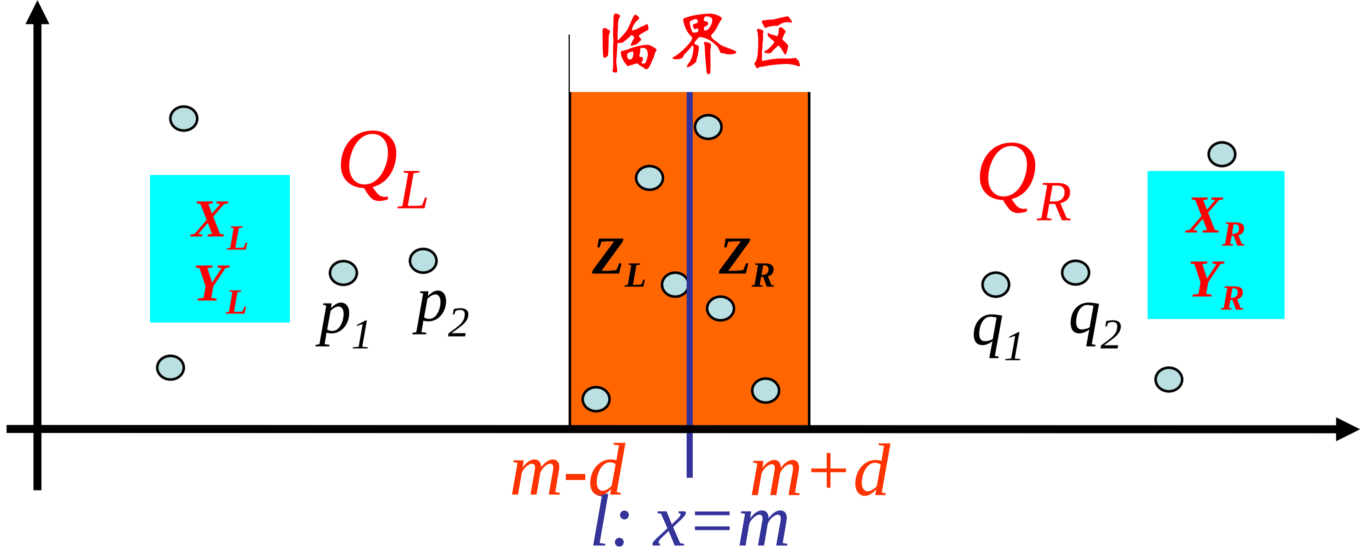
6. $d=\min\{Dis(p_1, p_2), Dis(q_1, q_2)\}$;



Merge:

1. 在临界区查找距离小于d的最近点对(p_l, q_r), $p_l \in Q_L$, $q_r \in Q_R$;
2. 若找到，则(p_l, q_r)是Q中最近点对，否则(p_1, p_2)和(q_1, q_2)中距离最小者为Q中最近点对.

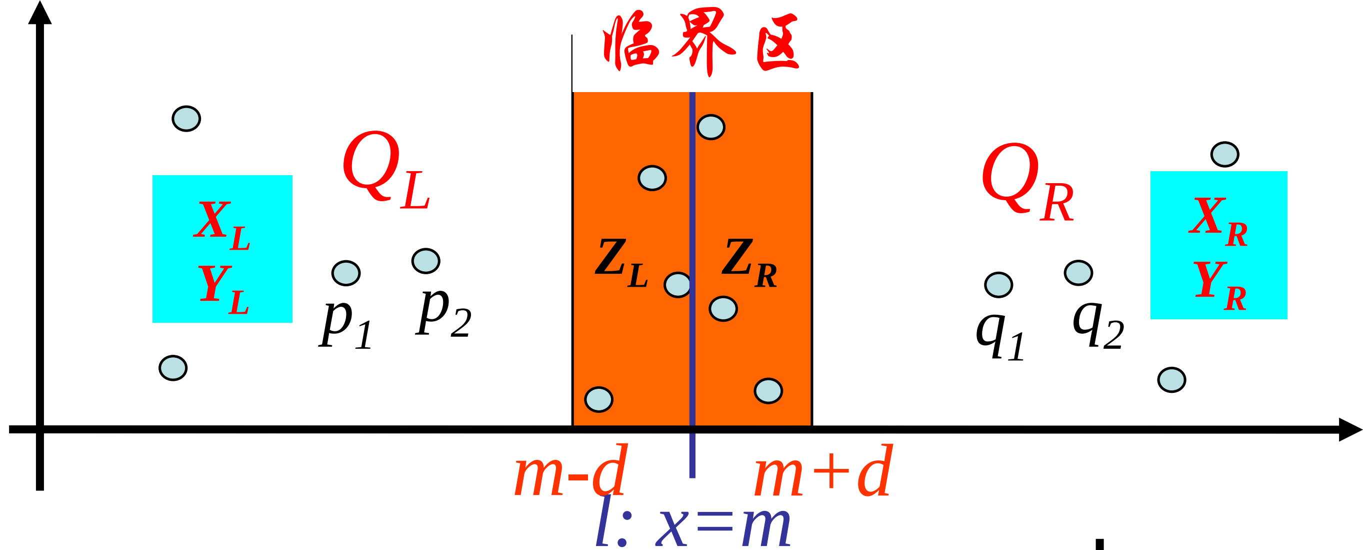
关键是(p_l, q_r)的搜索方法及其搜索时间



- (p_l, q_r)搜索算法
  1. $Z_L = \{Q_L$中左临界区点$\}$;
     $Z_R = \{Q_R$中右临界区点$\}$;

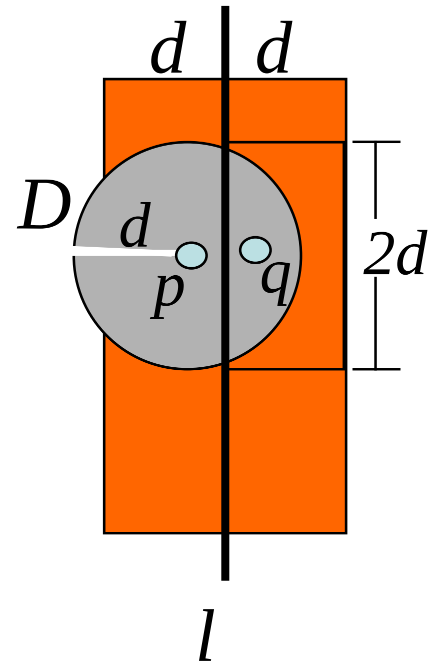
HITWH
SE

- 时间复杂性 $O(6n)=O(n)$



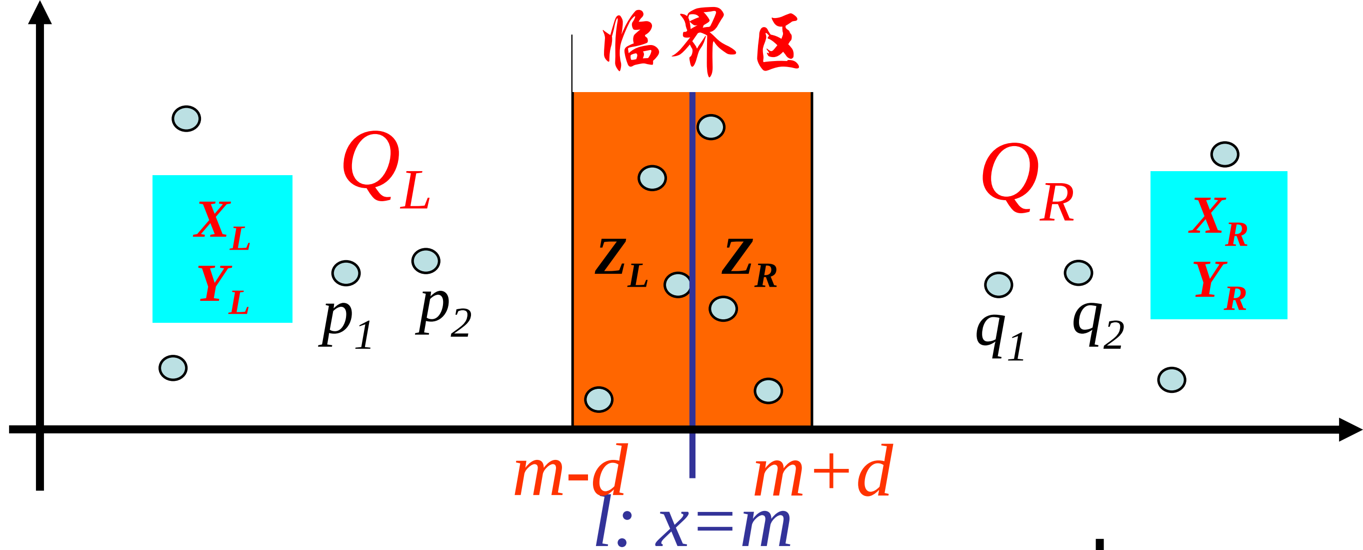
- (p_l, q_r)搜索算法

1. $Z_L=\{Q_L$ 中左临界区点$\}$;
   $Z_R=\{Q_R$ 中右临界区点$\}$;
2. For $\forall p(x_p, y_p)\in Z_L$ Do
3.   For $\forall q(x_q, y_q)\in Z_R$ Do $(y_p-d\le y_q\le y_p+d)$
   这样点至多6个\
4.   If $Dis(p, q)<d$
5.   Then $d=Dis(p, q)$, 记录(p, q);
6. 如果d发生过变化，与最后的d对应的点对即为(p_l, q_r)，否则不存在(p_l, q_r).



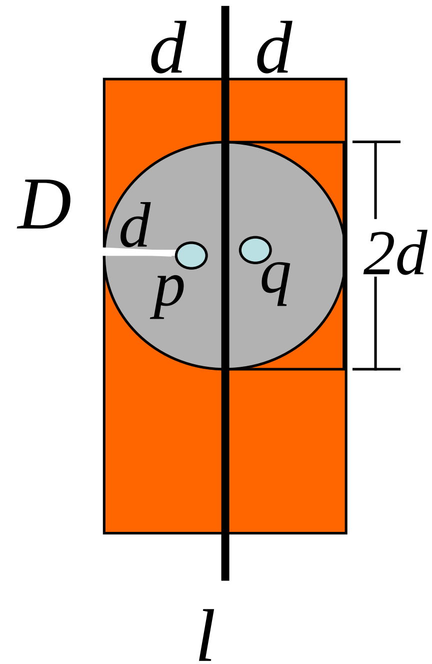
HITWH
SE

- **时间复杂性**
  $O(6n)=O(n)$



- (p_l, q_r)**搜索算法**

1. Z_L = {Q_L中左临界区点};
   Z_R = {Q_R中右临界区点};
2. For $\forall p(x_p, y_p) \in Z_L$ Do
3. For $\forall q(x_q, y_q) \in Z_R$ Do $(y_p - d \le y_q \le y_p + d)$
   这样点至多6个\
4. If $Dis(p, q) < d$
5. Then $d = Dis(p, q)$, 记录(p, q);
6. 如果d发生过变化, 与最后的d对应的点对即为(p_l, q_r), 否则不存在(p_l, q_r).



- 时间复杂性
  - Divide阶段需要$O(n)$时间
  - Conquer阶段需要$2T(n/2)$时间
  - Merge阶段需要$O(n)$时间
  - 递归方程

$$T(n) = O(1) \qquad n \le 3$$

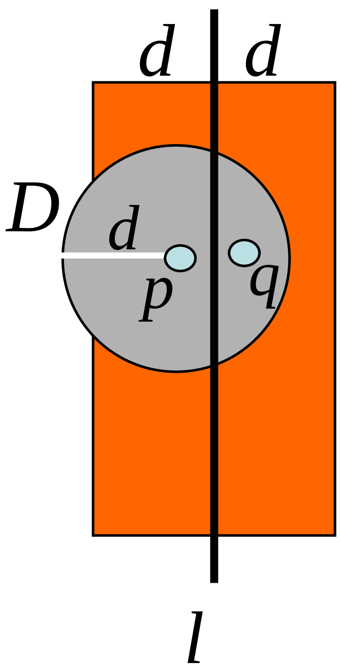
$$T(n) = 2T(n/2) + O(n) \qquad n > 3$$

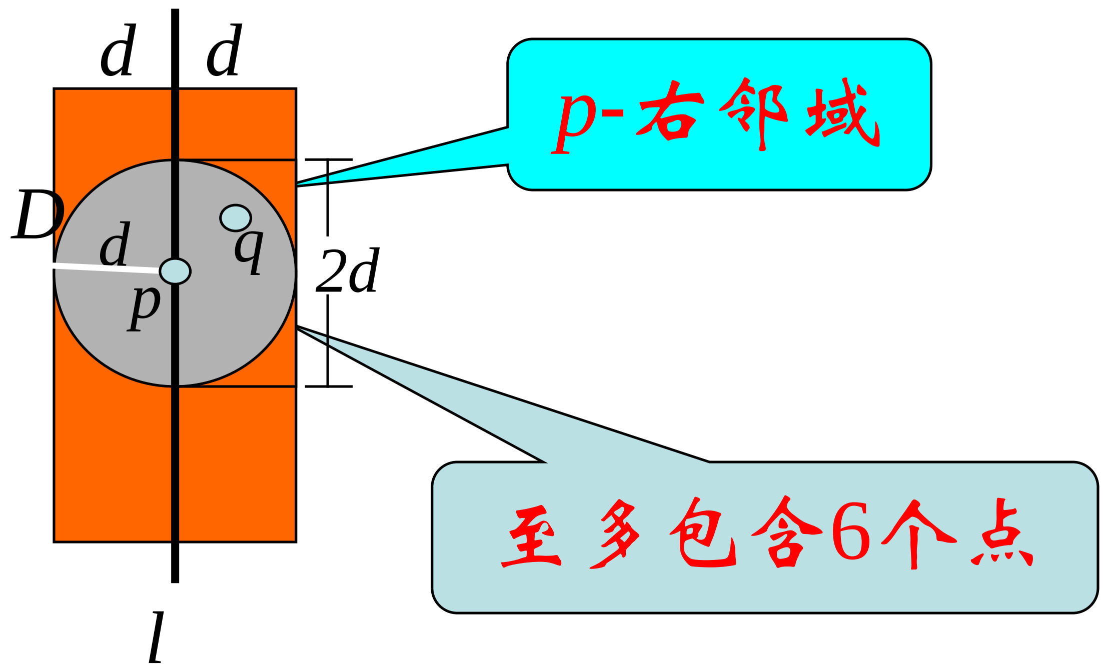
  - 用Master定理求解$T(n)$

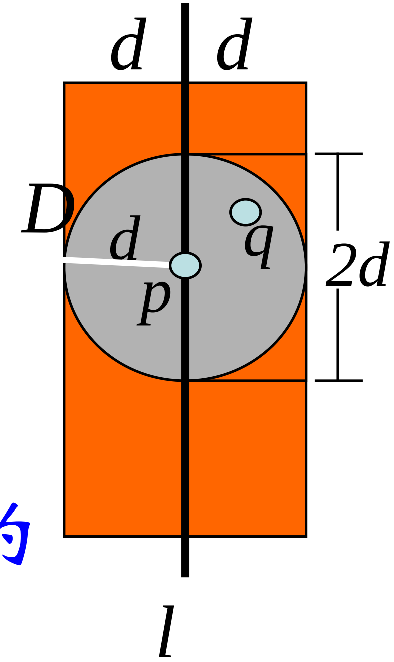
$$T(n) = O(n\log n)$$

(p_l, q_r)的搜索时间：

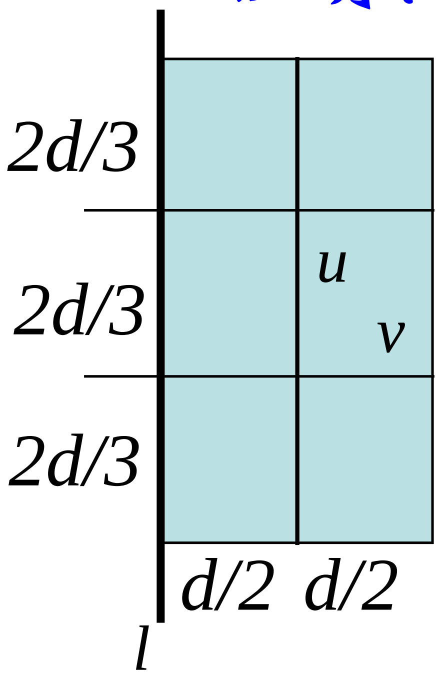
- 若(p, q)是最近点对而且$p \in Q_L$, $q \in Q_R$, $dis(p, q)<d$，(p, q)只能在下图的区域D.
- 若p在分割线l上，包含(p, q)的区域D最大，嵌于$d \times 2d$的矩形(p-右邻域)中，如下图所示.







定理1. 对于左临界区中的每个点p, p-右邻域中至多包含6个点。



证明：把p-右邻域划分为6个$(d/2)\times(2d/3)$的矩形。

若p-右邻域中点数大于6, 由鸽巢原理, 至少有一个矩形中有两个点. 设为u、v, 则

$$(x_u-x_v)^2+(y_u-y_v)^2\le(d/2)^2+(2d/3)^2=25d^2/36$$

即$Dis(u, v)\le 5d/6<d$, 与d的定义矛盾。

- **Assume:**

Q中点已经分别按x坐标和y坐标排序后存储在X和Y中.

1. X=按x排序Q中点;
2. Y=按y排序Q中点;
3. FindCPP(X, Y).

时间复杂性=$O(n\log n)$+T(FindCPP)=$O(n\log n)$

扩展到三维空间或更高维空间如何?

3.5 Sorting in Linear Time

- Counting Sort Algorithm
- Radix Sort Algorithm
- Bucket Sort Algorithm

Counting Sort

- Input: $A[1..n]$, $0 \le A[i] \le k$ for $1 \le i \le n$
- Output: $B[1..n]$=sorted $A[1..n]$
- Idea
  - Use $C[0..k]$ to compute the position of each $A[i]$
  - Put each $A[i]$ for $i=n$ *to* 1 into $B[C[A[i]]]$

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B								

	1	2	3	4	5	6	7	8
B							3	

	1	2	3	4	5	6	7	8
B		0					3	

	1	2	3	4	5	6	7	8
B		0				3	3	

	1	2	3	4	5	6	7	8
B		0		2		3	3	

	1	2	3	4	5	6	7	8
B	0	0		2		3	3	

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

	0	1	2	3	4	5
C	2	0	2	3	0	1

	0	1	2	3	4	5
C	2	2	4	7	7	8

	0	1	2	3	4	5
C	2	2	4	6	7	8

	0	1	2	3	4	5
C	1	2	4	6	7	8

	0	1	2	3	4	5
C	1	2	4	5	7	8

	0	1	2	3	4	5
C	1	2	3	5	7	8

	0	1	2	3	4	5
C	0	2	3	5	7	8

- Algorithm and Time complexity

$$\text{for } i \leftarrow 0 \text{ to } k$$
$$\quad \text{do } C[i] \leftarrow 0;$$

$O(k)$

$$\text{for } j \leftarrow 1 \text{ to } length[A]$$
$$\quad \text{do } C[A[j]] \leftarrow C[A[j]]+1;$$

$O(n)$

$$\text{for } i \leftarrow 1 \text{ to } k$$
$$\quad \text{do } C[i] \leftarrow C[i]+C[i-1];$$

$O(k)$

$$\text{for } j \leftarrow length[A] \text{ downto } 1$$
$$\quad \text{do begin}$$
$$\qquad B[C[A[j]]] \leftarrow A[j];$$
$$\qquad C[A[j]] \leftarrow C[A[j]]-1;$$

$O(n)$

Time Complexity=O(n+k)

- Counting Sort算法性质
  - Counting sort doesn’t sort in place
  - Counting sort is stable
    - That is, the same values appear in the output array in the same order as they do in the input array.
  - Problems
    - *A[i]* must be integer.
    - *k* should be small

Radix Sort Algorithm

- Idea of Radix sort algorithm
  - Use stable sort algorithm
  - Sort the *n* *d*-digit elements from the lowest digit to the highest digit

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

- Radix sort algorithm

Ininput: Array A, each element is a number of d digit.

Radix-Sort(A, d)

for $i \leftarrow 1$ to d do

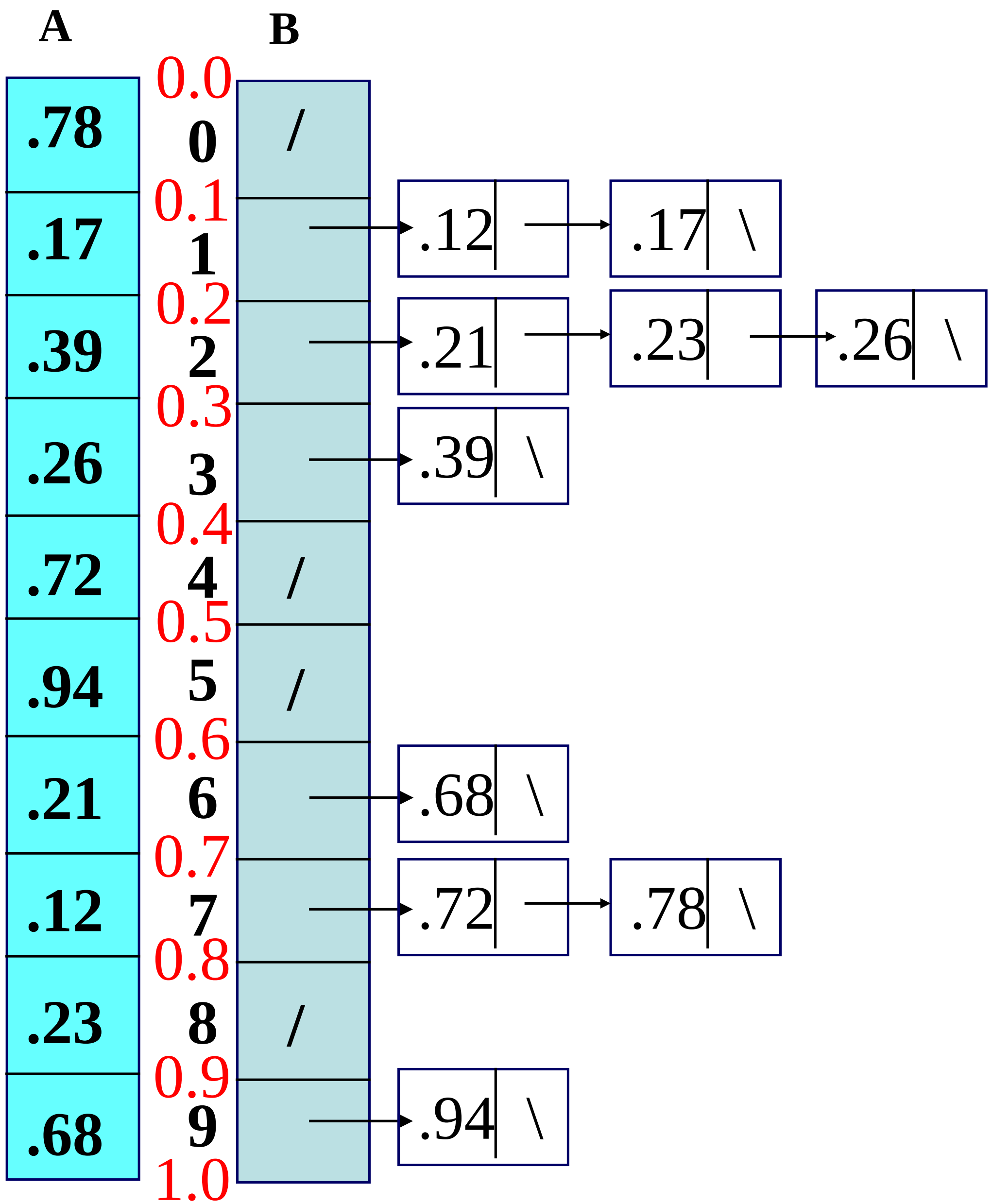
    use a stable sort to sort array A on digit i;

- Time complexity of Radix sort algorithm
  - Using Counting sort algorithm, 0≤A[i]≤k
  - The time complexity is *$O(d(n+k))$*
- Problems

- Extension of Radix sort
  - Input: n b-binary-digit number, any $r \leq b$
  - Radix sort can sort these numbers in $\Theta((b/r)(n+2^r))$
  - Why
    - View each number as $d=\lceil b/r \rceil$ digits of r bits each.
    - Each digit is an integer in the range 0 to 2^r-1
    - Use counting sort with $k= 2^r-1$
  - How about b=500, r=100?

Bucket Sort

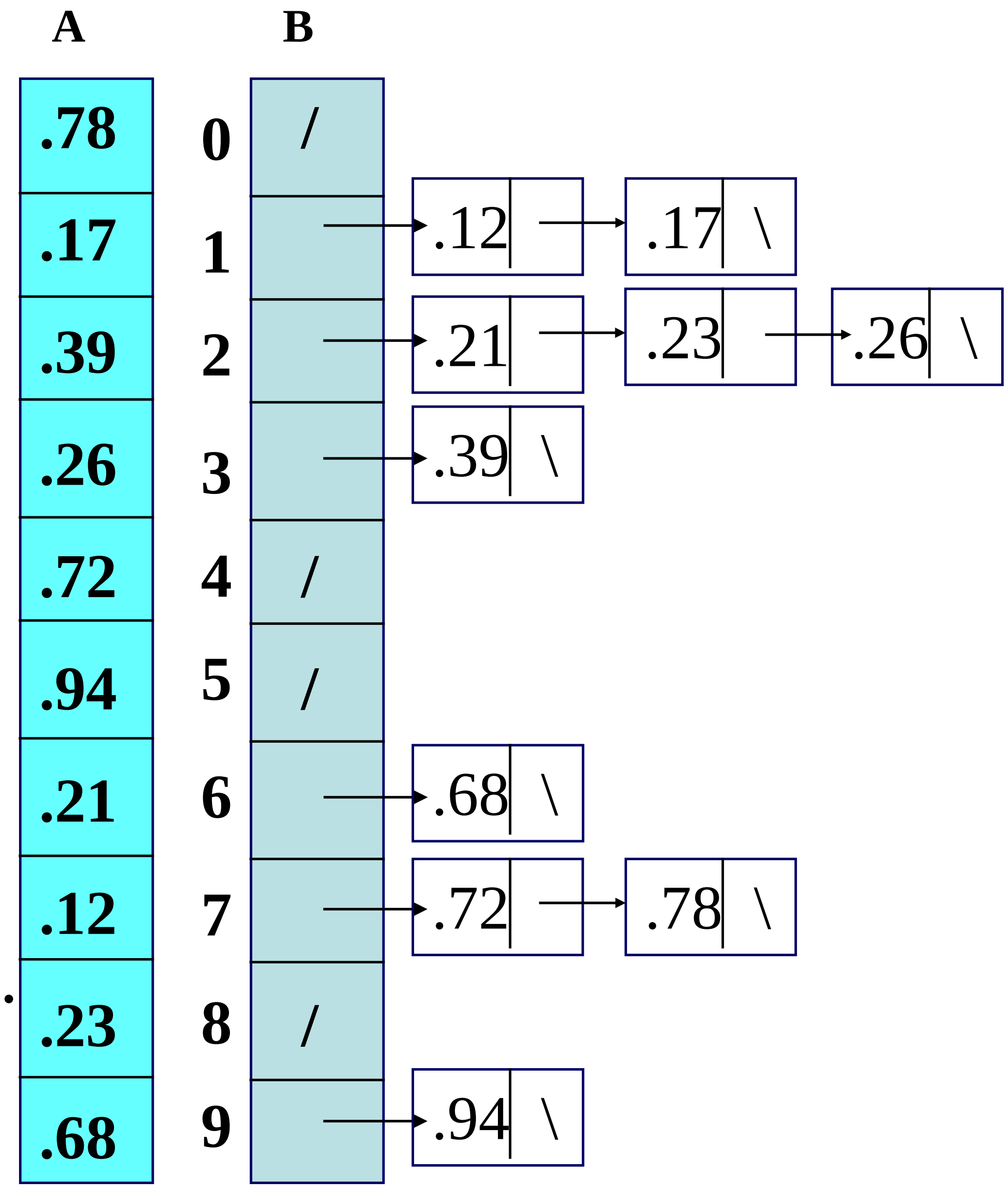
- **Assumption of Bucket Sort**
  - Input is elements uniformly distributed in [0, 1) independently
- **Idea of Bucket Soot**
  - Divide [0, 1) into *n* equal-sized bucket
  - Distribute the input into the *n* bucket
  - Sort the numbers in each bucket
  - List all the sorted numbers in each bucket in order



- **Bucket Sort Algorithm**

Bucket-Sort(A)

1. n=*length*[A];
2. For i=*1* To n Do
3. Insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$;
4. For i=0 To n-1 Do
5. Sort list $B[i]$ with insert sort;
6. concatenate lists $B[0]$, …, $B[n\text{-}1]$.



- **Time complexity**
  - Let the random variable $n_i=|B[i]|$
  - The time complexity:

$$T(n)=\Theta(n)+\sum_{i=0}^{n-1}O(n_i^{\ 2})$$

  - Since $E[n_i^2]=2-1/n$

$$\begin{aligned}E[T(n)]&=\Theta(n)+\sum_{i=0}^{n-1}E[O(n_i^{\ 2})]\\&=\Theta(n)+O(n(2-1/n))=\Theta(n)\end{aligned}$$

3.6 Finding the convex hull

问题定义

输入：平面上的n个点的集合Q

输出：CH(Q): Q的convex hull

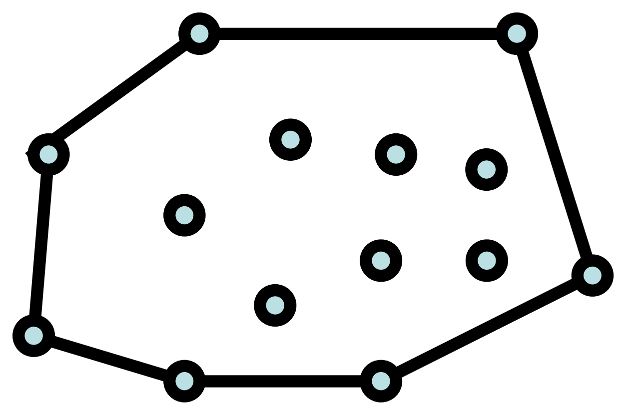
Q的convex hull是一个最小凸多边形P，Q的点或者在P上或者在P内

凸多边形P是具有如下性质多边形：连接P内任意两点的边都在P内

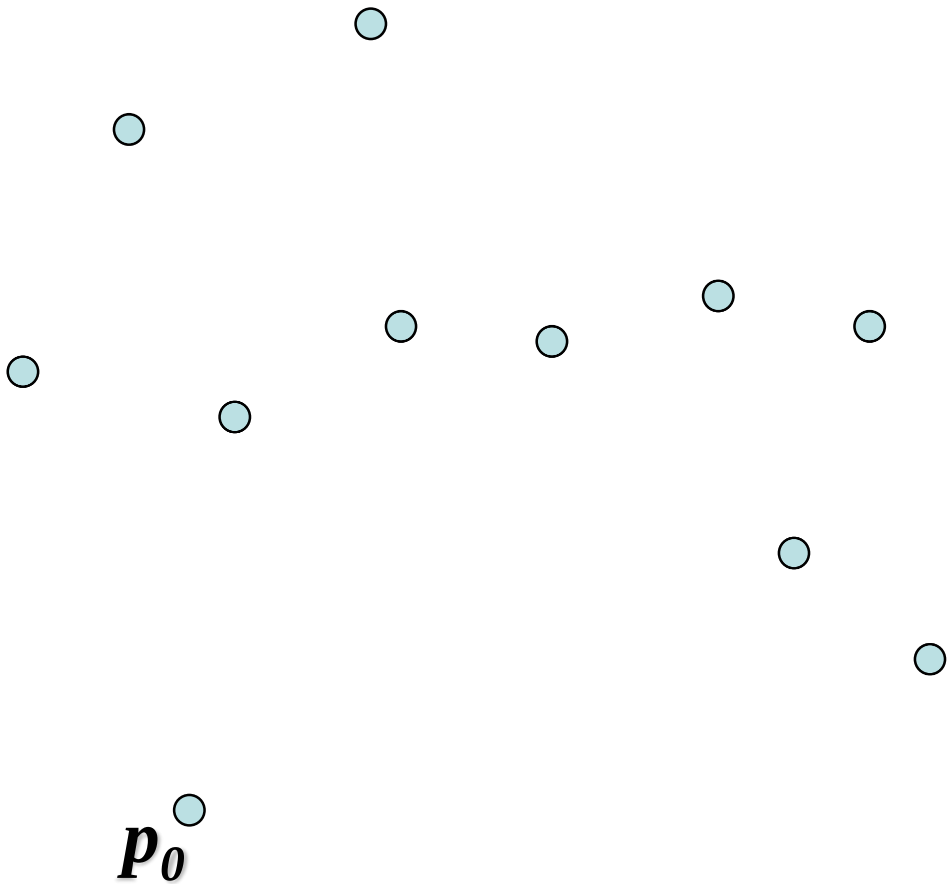
HITWH
SE

Graham-Scan算法

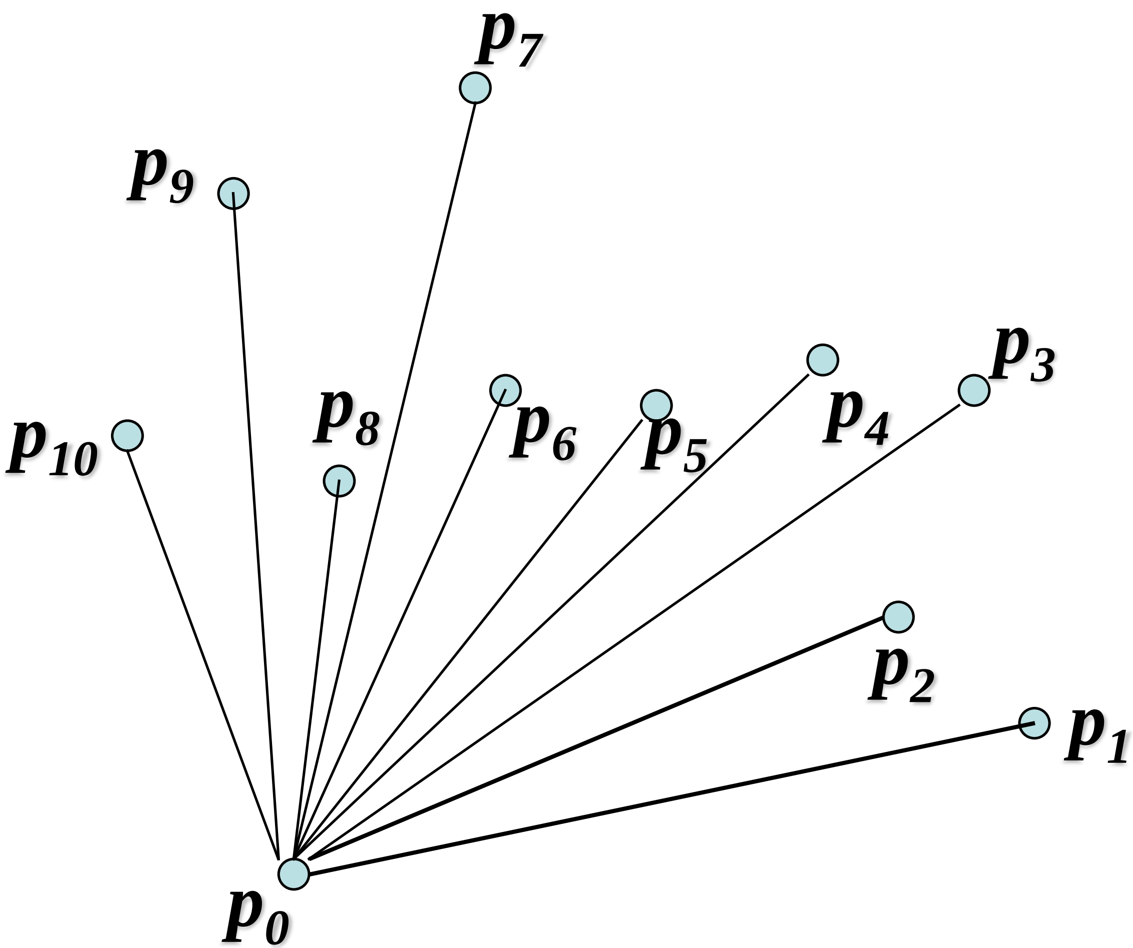
- 基本思想
  - 当沿着Convex hull逆时针漫游时，总是向左转
  - 在极坐标系下按照极角大小排列，然后逆时针方向漫游点集，去除非Convex hull顶点(非左转点)。



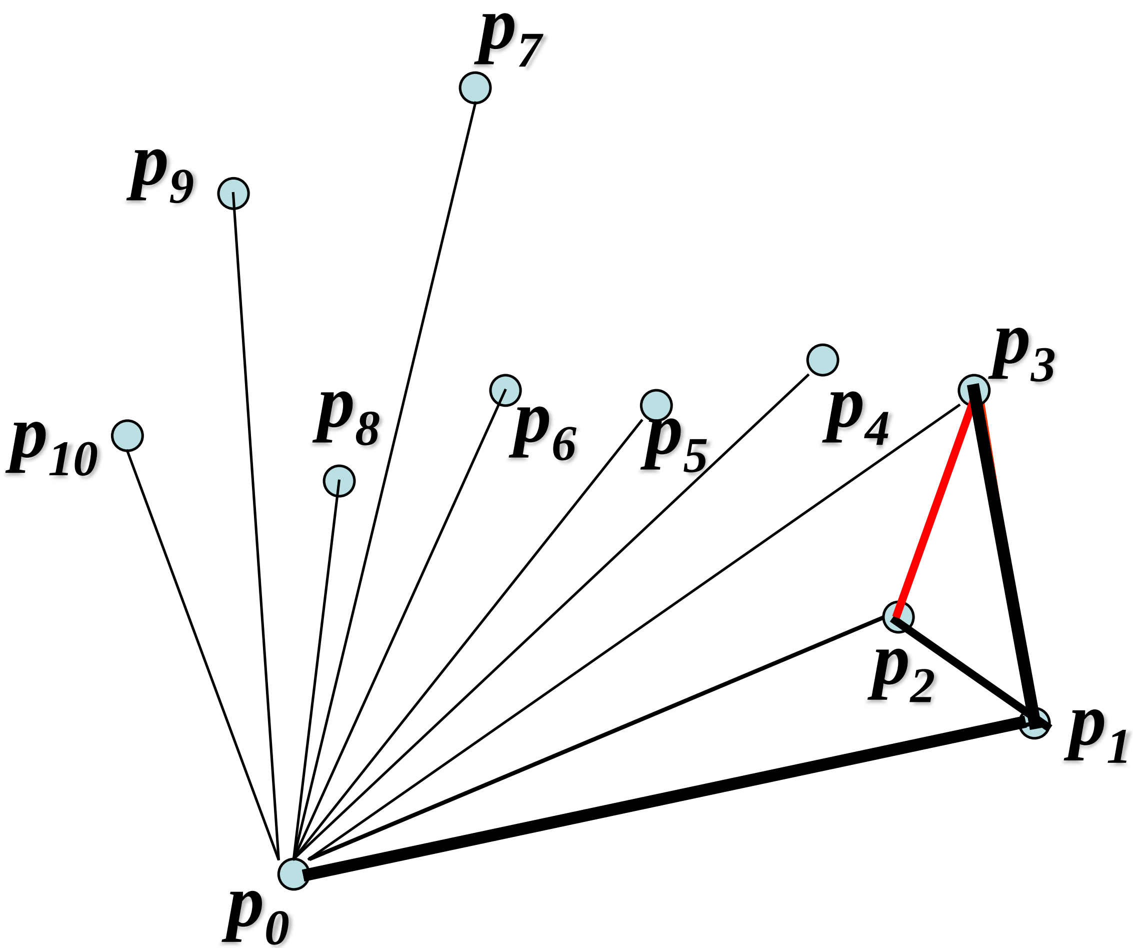
第1步



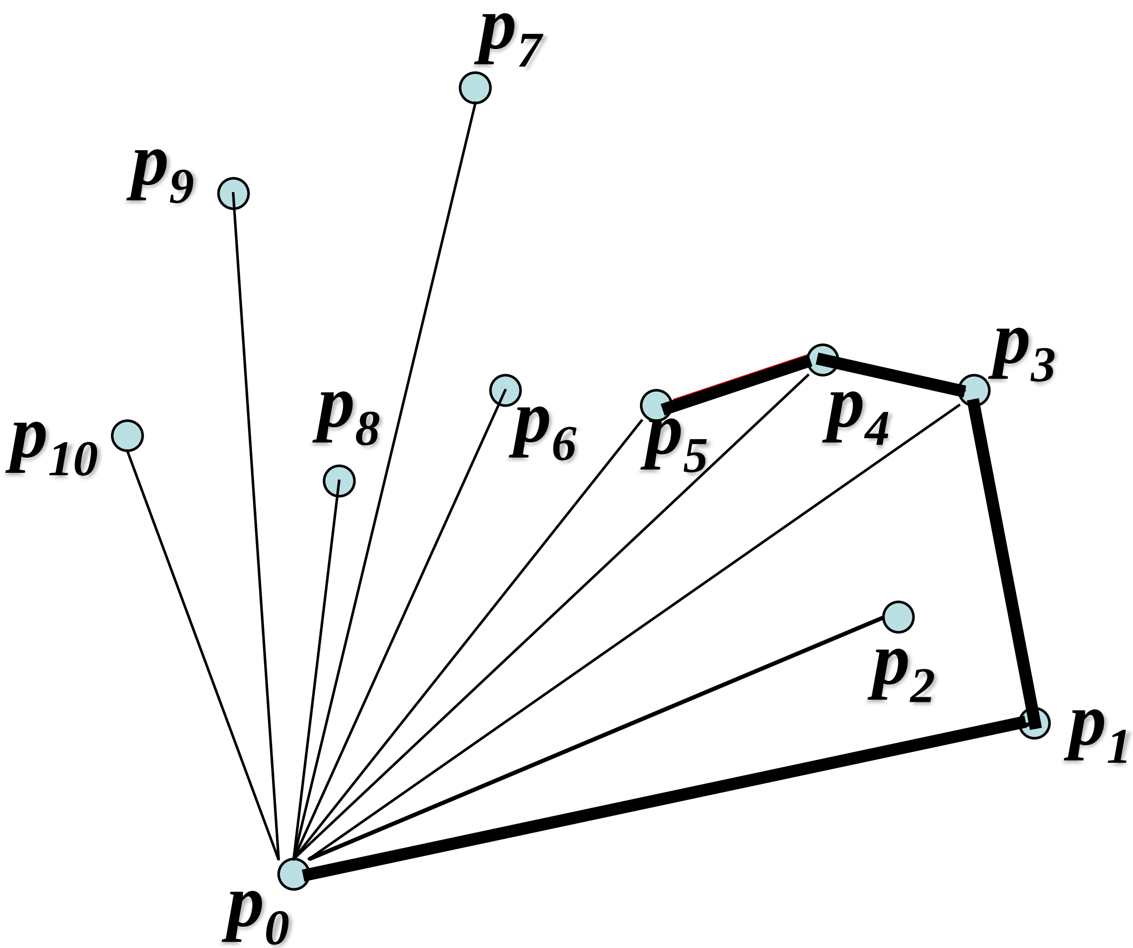
第2步

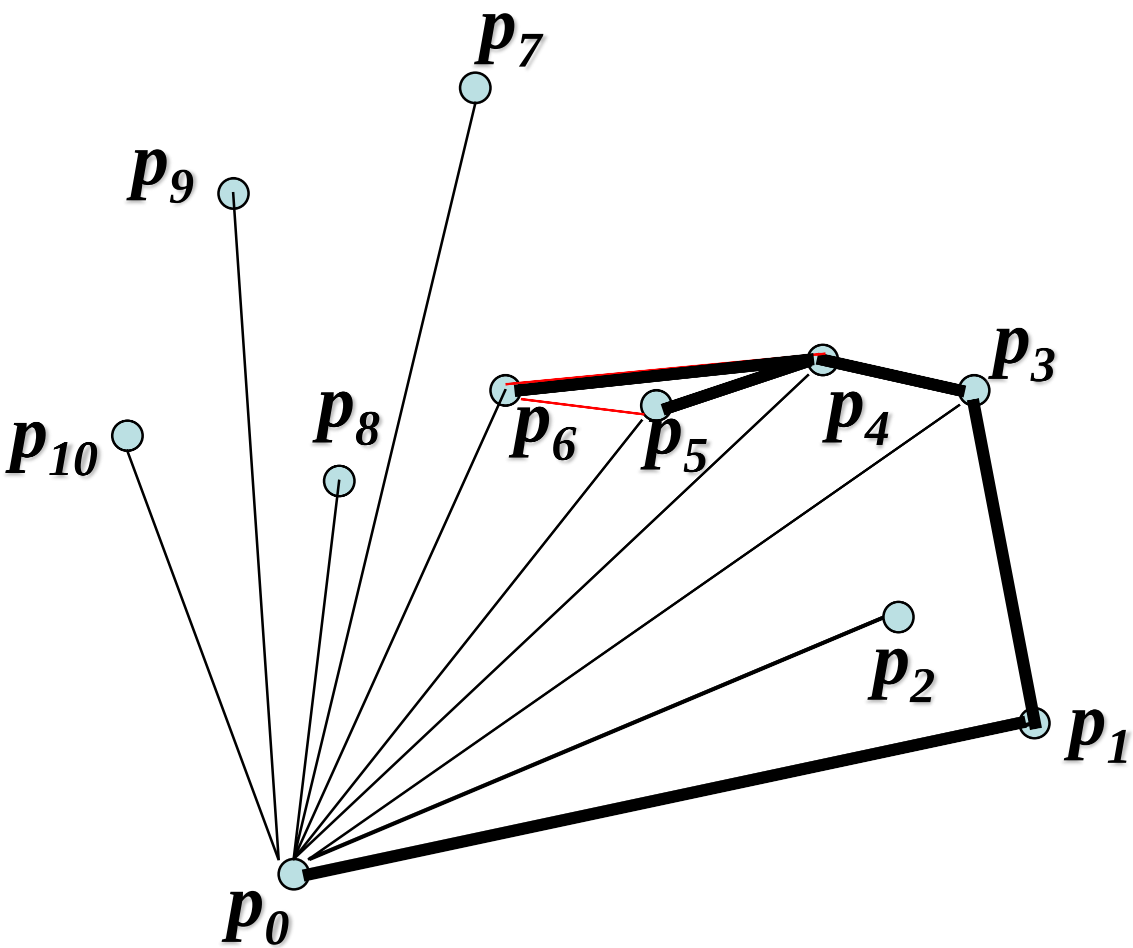


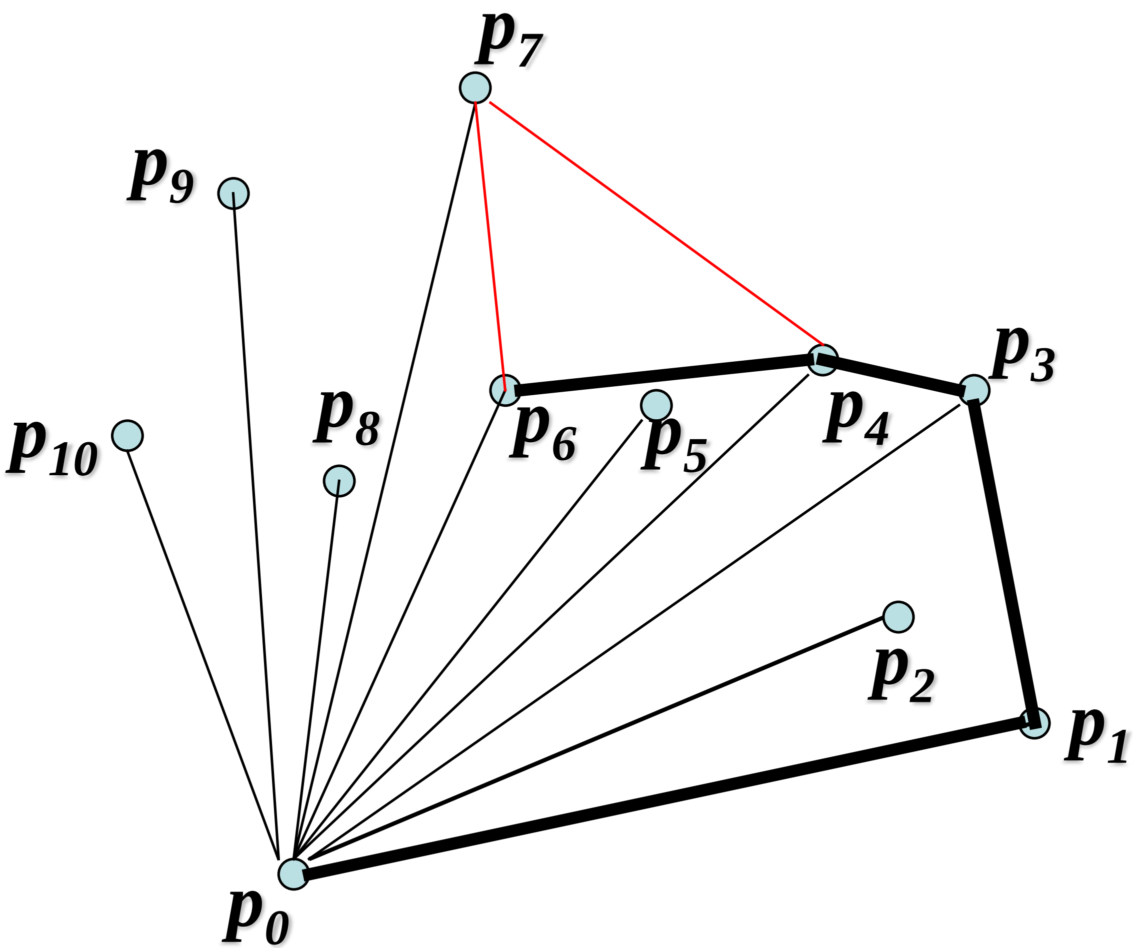
第3步

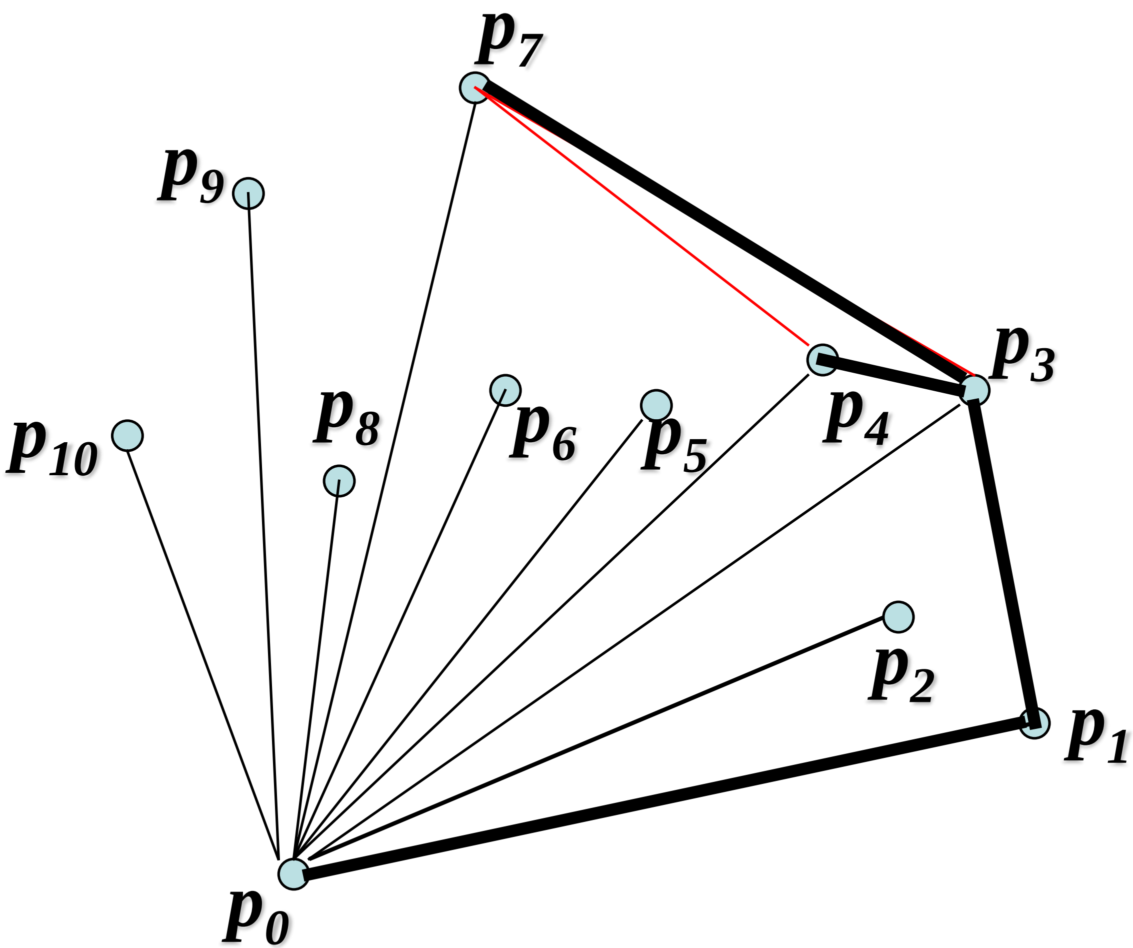


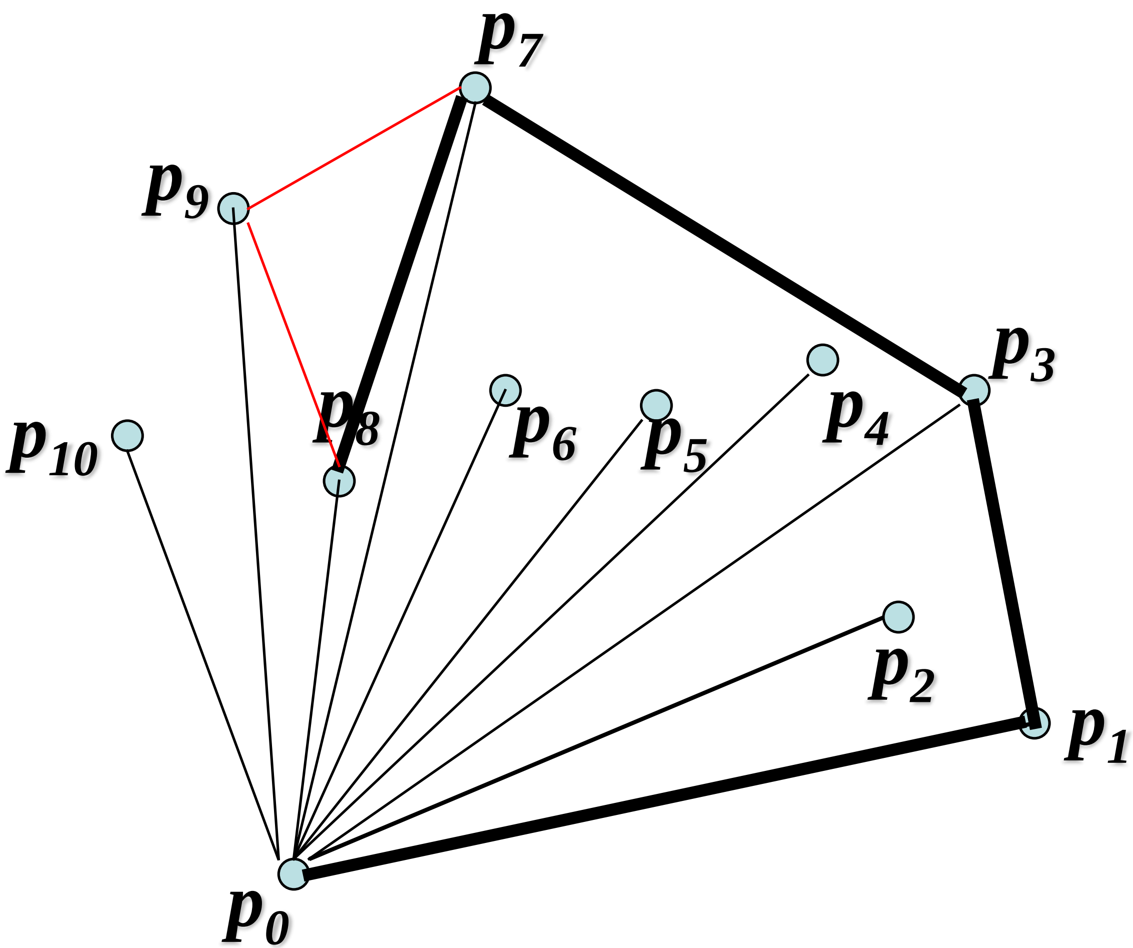
HITWH
SE

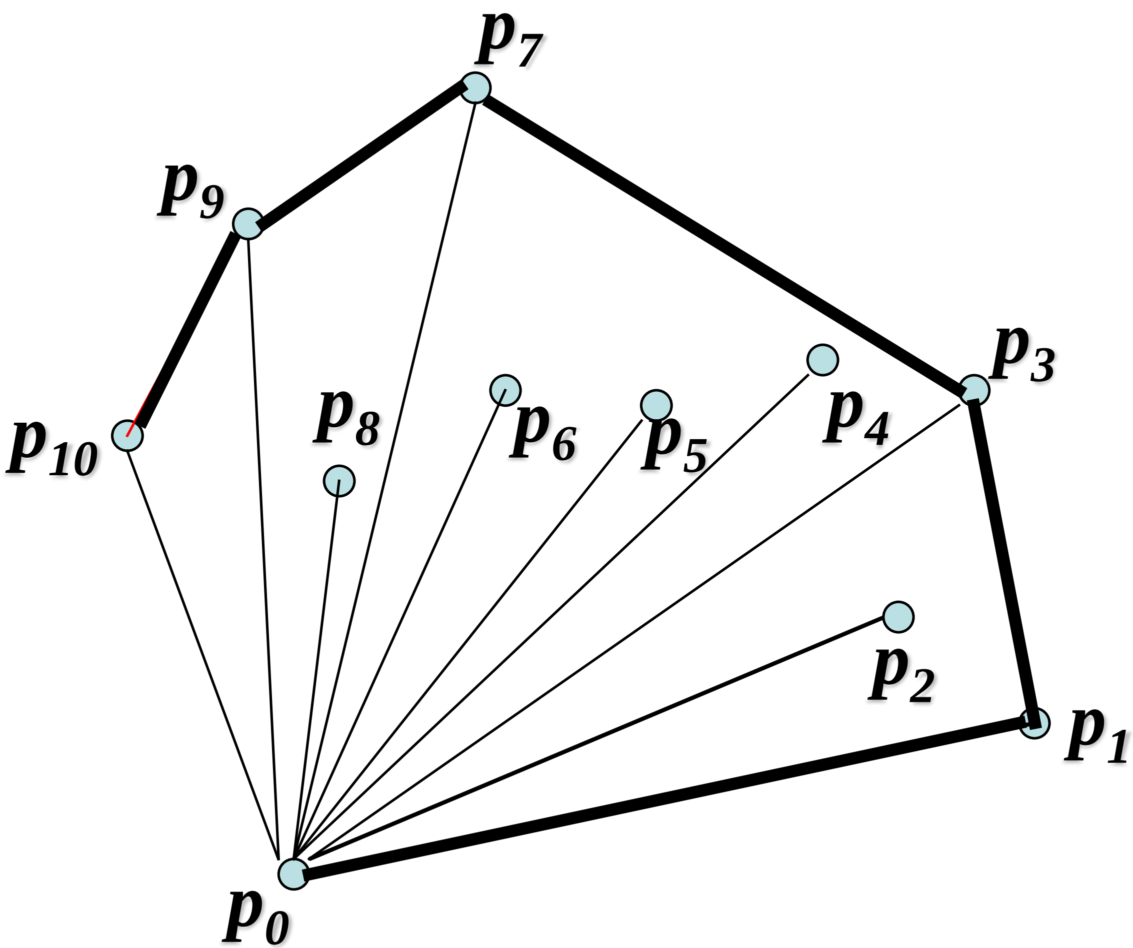



p7
p9
p8
p10
p6
p5
p4
p3
p2
p1
p0


p_7
p_9
p_3
p_8
p_4
p_{10}
p_6
p_5
p_2
p_1
p_0


p7
p9
p3
p4
p8
p6
p5
p10
p2
p1
p0

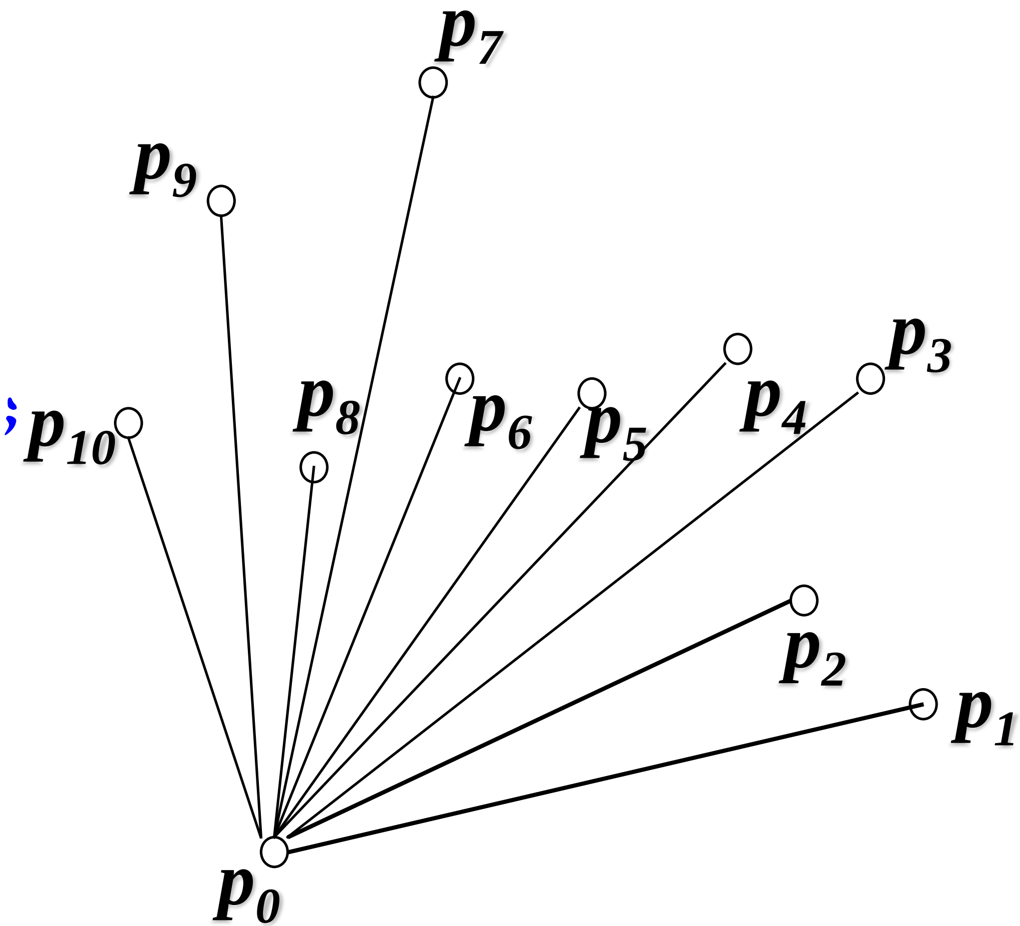

p_7
p_9
p_3
p_8
p_6
p_5
p_4
p_{10}
p_2
p_1
p_0


p_7
p_9
p_3
p_8
p_6
p_5
p_4
p_{10}
p_2
p_1
p_0

算法Graham-Scan(Q)

/* 栈S从底到顶存储按逆时针方向排列的CH(Q)顶点 */

1. 求Q中y-坐标值最小的点p_0;
2. 按照与p_0极角(逆时针方向)大小排序Q中其余点,
   结果为$<p_1, p_2, \ldots, p_n>$;
3. Push p_0, p_1, p_2 into S;
4. FOR i=3 TO n DO
5. While Next-to-top(S)、Top(S)和p_i形成非左移动 Do
6. Pop(S);
7. Push(p_i, S);
8. Rerurn S.



HITWH
SE

- 时间复杂性$T(n)$

1. 求Q中y-坐标值最小的点p_0;
2. 按照与p_0极角(逆时针方向)大小排序Q中其余点，结果为<$p_1, p_2, \ldots, p_n$>;
3. Push p_0, p_1, p_2 into S;
4. **FOR** i=3 TO n DO
5. **While** Next-to-top(S)、Top(S) 和p_i形成非左移动 **Do**
6. Pop(S);
7. Push(p_i, S);
8. Rerurn S.

- 第1步需要$O(n)$时间
- 第2步需要$O(n\log n)$时间
- 第3步需要$O(1)$时间
- 第4-7步需要$O(n)$时间
  - 因为每个点至多进栈一次出栈一次，每次需要常数计算时间
- $T(n)= O(n\log n)$

- 正确性分析

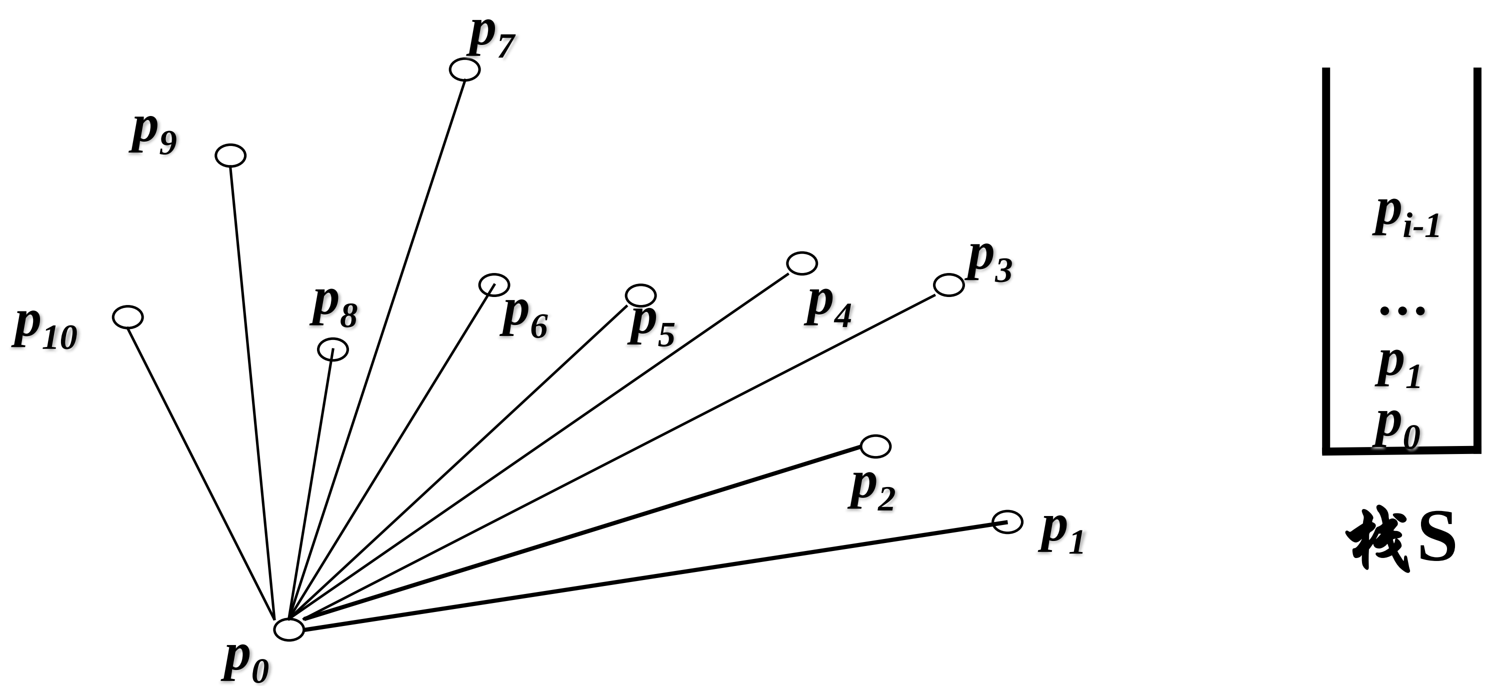
定理. 设n个二维点的集合Q是Graham-Scan算法的输入，$|Q|\geq 3$，算法结束时，栈S中自底到顶存储CH(Q)的顶点（按照逆时针顺序）.

证明: 使用循环不变量方法

```
Push p0, p1, p2 into S;
FOR  i=3  TO  n  DO
    While  Next-to-top(S)、Top(S)
              和pi形成非左移动   Do
                    Pop(S);
    Push(pi, S);
```

Loop invariant

在处理第i个顶点之前，栈S中自底到顶存储$CH(Q_{i-1})$的顶点.



```
3.  Push p0, p1, p2 into S;
4.  FOR  i=3  TO  n  DO
5.     While  Next-to-top(S)、Top(S)
              和pi形成非左移动   Do
6.                 Pop(S);
7.       Push(pi, S);
```

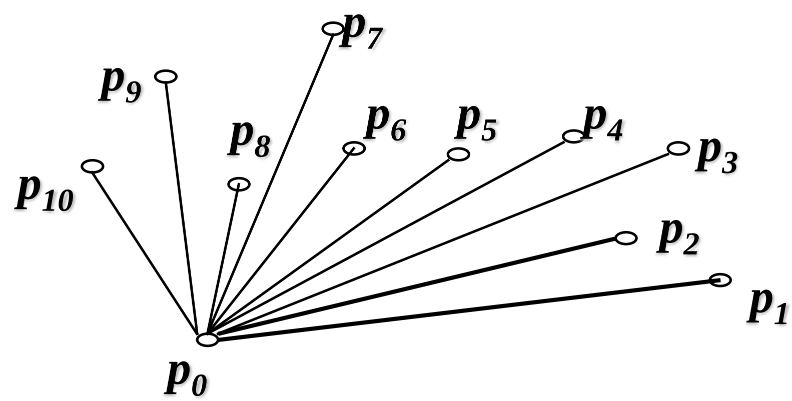
循环不变量

在处理第i个顶点之前，栈S自底到顶存储$CH(Q_{i-1})$的顶点.

- Proof by induction
  - Initialization: (第3步)
    - 处理$i=3$之前．栈S中包含了$Q_{i-1}=Q_2=\{p_0,p_1,p_2\}$中的顶点，这三个点形成了一个CH. 循环不变量为真.
  - Maintenance:
    - 设在处理第$i(i\geq 3)$个顶点之前，循环不变量为真，即：栈S中自底到顶存储$CH(Q_{i-1})$的顶点.
    - 往证：

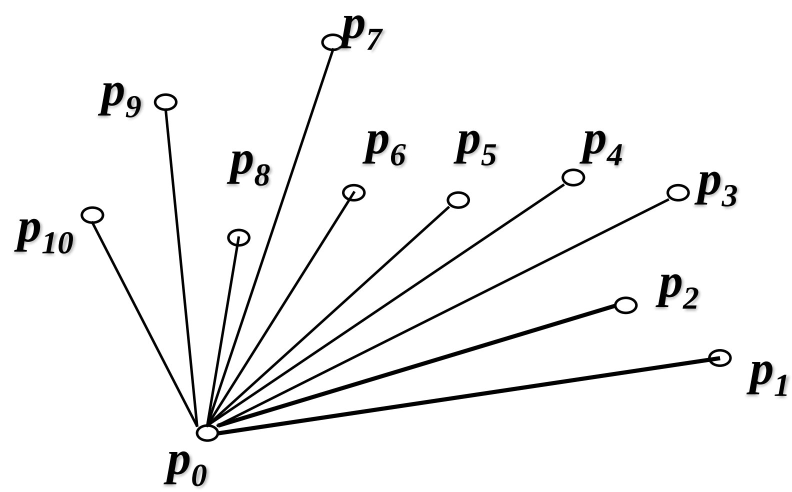
      算法执行5~7步之后，栈S中自底到顶存储$CH(Q_i)$的顶点.

```
3.  Push p0, p1, p2 into S;
4.  FOR  i=3  TO  n  DO
5.     While  Next-to-top(S)、Top(S)
              和pi形成非左移动   Do
6.                  Pop(S);
7.       Push(pi, S);
```



- 往证：算法执行5~7步后,栈S中自底到顶存储CH(Q_i)的顶点
  - 5~6步while循环执行结束后，第7步将p_i压入栈之前，设栈顶元素为p_j，次栈顶元素为p_k，则此时，栈中包含了与for循环的第j轮迭代后相同的顶点，即CH(Q_j)，循环不变量为真.
  - 执行第7步之后，p_i入栈，则栈S中包含了CH($Q_j \cup \{p_i\}$)中的顶点,且这些点仍按逆时针顺序，自底向上出现在栈中.　　CH($Q_j \cup \{p_i\}$)= CH(Q_i)?
  - 对于任意一个在第i轮迭代中被弹出的栈顶点p_t，设 p_r为紧靠p_t的次栈顶点，p_t被弹出当且仅当p_r、p_t、p_i构成非左移动。因此，p_t不是CH(Q_i)的一个顶点，即CH($Q_i - \{p_t\}$)= CH(Q_i)。
  - 设P_i为for循环第i轮迭代中被弹出的所有点的集合,则有CH($Q_i - P_i$)= CH(Q_i)
  - 又 $Q_i - P_i = Q_j \cup \{p_i\}$，故有CH($Q_j \cup \{p_i\}$)= CH($Q_i - P_i$)= CH($Q_i$)
  - 即得到：一旦将p_i压入栈后，栈S中恰包含CH(Q_i)中的顶点，且按照逆时针顺序，自底向上排列。

Push p_0, p_1, p_2 into S;
FOR i=3 TO n DO
  While Next-to-top(S)、Top(S)
    和p_i形成非左移动 **Do**
      Pop(S);
  Push(p_i, S);



– Termination:

- $i=n+1$,栈S中自底到顶存储CH(Q_n)的顶点,算法正确.

证毕.

Divide-and-conquer算法

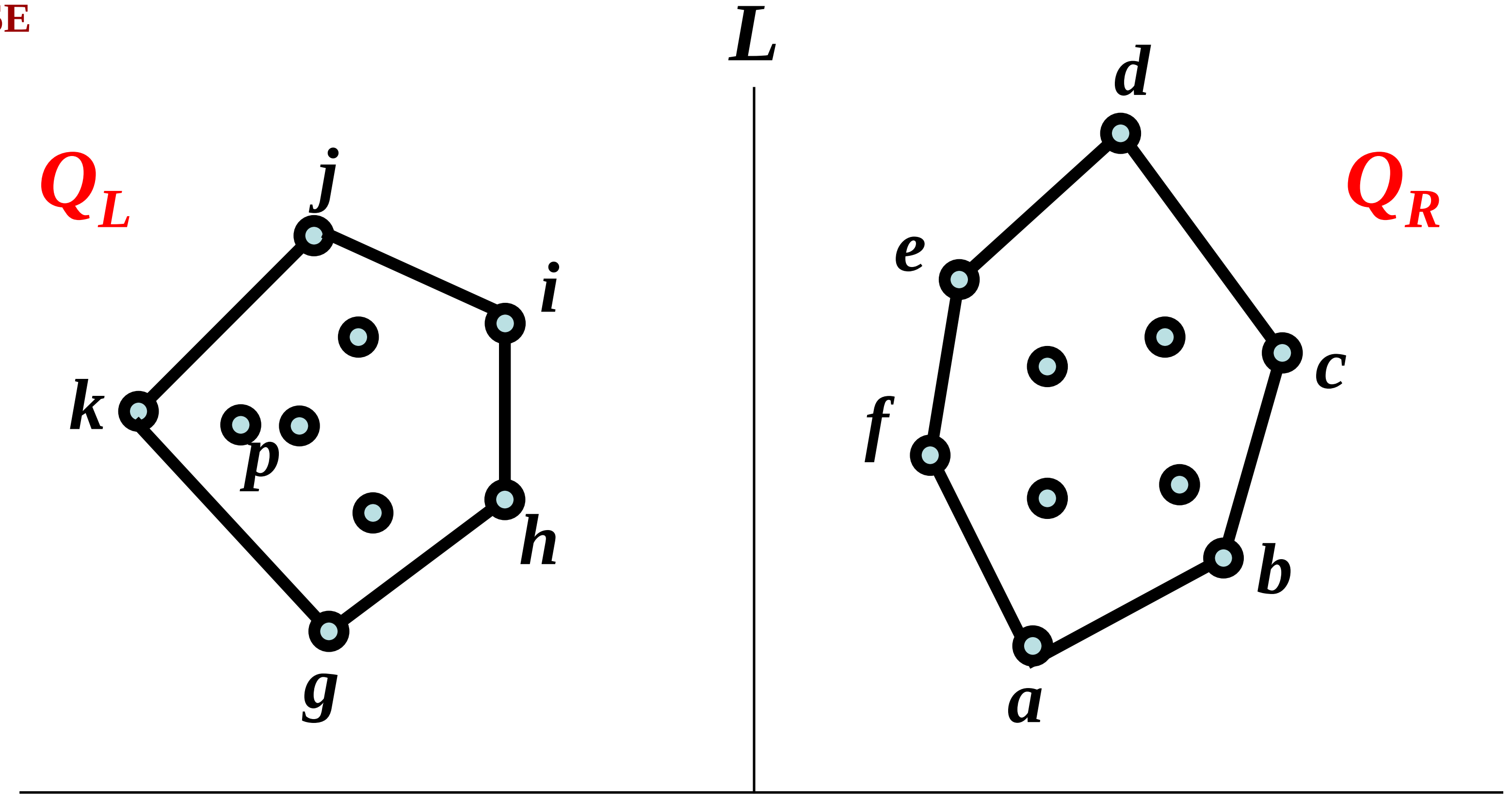
边界条件：(时间复杂性为$O(1)$)

1. 如果|Q|<3, 算法停止；
2. 如果|Q|=3, 按照逆时针方向输出CH(Q)的顶点;

Divide:(使用$O(n)$算法求中值)

1. 选择一个垂直于x-轴的直线把Q划分为基本相等的两个集合Q_L和Q_R, Q_L在Q_R的左边;

HITWH
SE



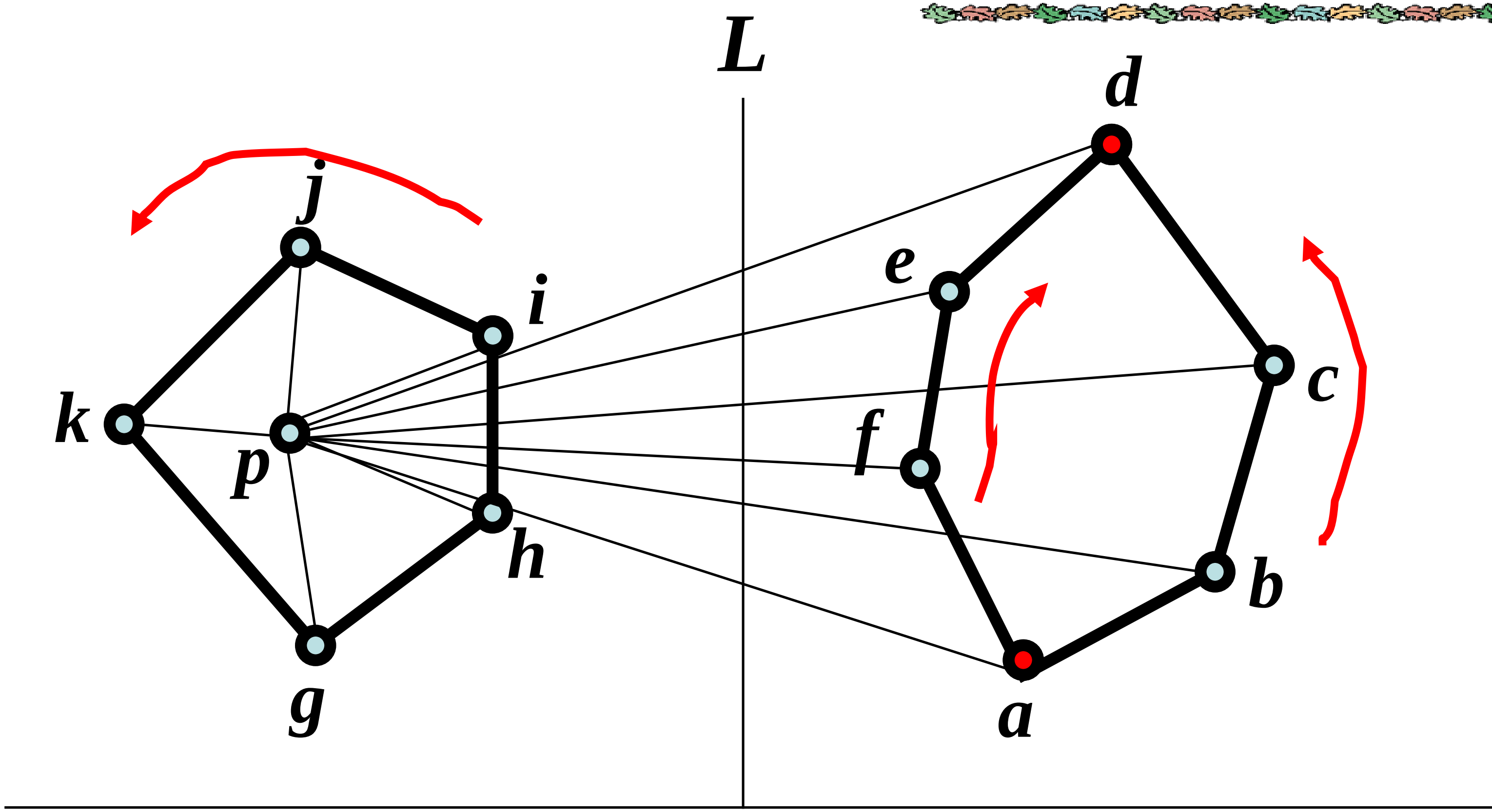
Conquer: (时间复杂性为$2T(n/2)$)

1. 递归地为Q_L和Q_R构造$CH(Q_L)$和$CH(Q_R)$;

Merge:

我们先通过一个例子来看Merge的思想

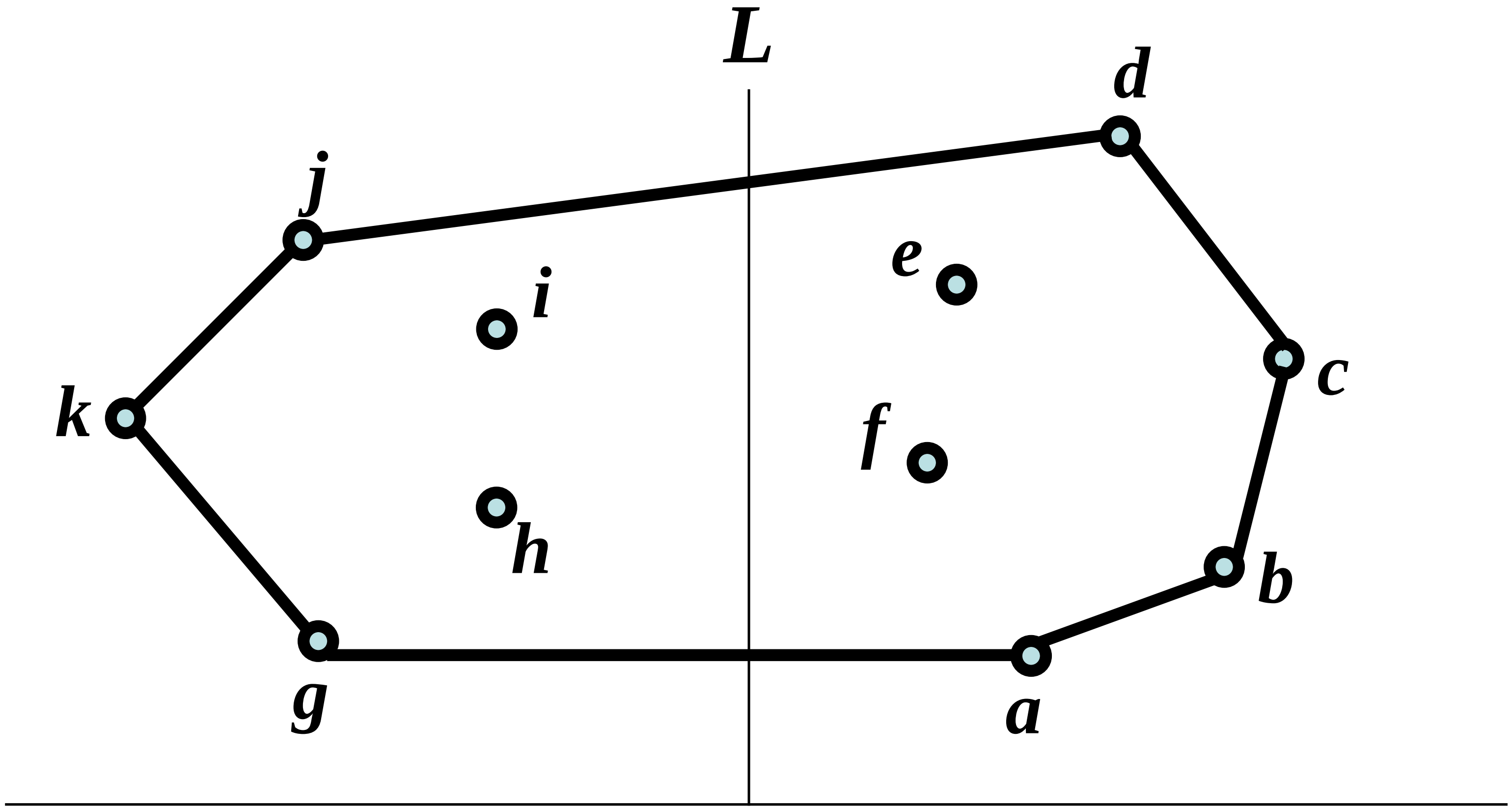
Merge实例



3个序列：<g, h, i, j, k>, <a, b, c, d>, <f, e>

合并以后：<g, h, a, b, f, c, e, d, i, j, k>

HITWH
SE



选择p点为y坐标最小的点
在合并的序列上运行Graham-Scan，排序部分合并三个有序序列即可

Merge:(时间复杂性为$O(n)$)

1. 找到Q_L和Q_R中y坐标最小的点p (假设在Q_L中);
2. 在CH(Q_R)中找与p的极角最大和最小顶点u和v;
3. 构造如下三个点序列:
   (1) 按逆时针方向排列的CH(Q_L)的所有顶点,
   (2) 按逆时针方向排列的CH(Q_R)从v到u的顶点,
   (3) 按顺时针方向排列的CH(Q_R)从v到u的顶点;
4. 合并上述三个序列;
5. 在合并的序列上应用Graham-Scan.

时间复杂性

- Preprocessing阶段
  - $O(1)$
- Divide阶段(使用$O(n)$算法求中值)
  - $O(n)$
- Conquer阶段
  - $2T(n/2)$
- Merge阶段
  - $O(n)$

- 总的时间复杂性

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

- 使用Master定理

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

HITWH
SE

3.7 快速傅里叶变换

问题定义

输入： $a_0, a_1, \ldots, a_{n-1}$， a_i是实数，$(0 \le i \le n\text{-}1)$

输出： $A_0, A_1, \ldots, A_{n-1}$， 使得，

$$A_j = \sum_{k=0}^{n-1} a_k e^{\frac{2\pi ijk}{n}}$$

其中：

(2) $0 \le j \le n\text{-}1$

(3) e是自然对数的底数

(4) $i = \sqrt{-1}$ 是虚数单位

蛮力法利用定义计算每个A_j，时间复杂度为$\Theta(n^2)$

$$A_j = \sum_{k=0}^{n-1} a_k e^{\frac{2\pi ijk}{n}}$$ 令 $w_n = e^{\frac{2\pi i}{n}}$， 有：$A_j = \sum_{k=0}^{n-1} a_k w_n^{jk}$

$$A_j = a_0 + a_1 w_n^{j} + a_2 w_n^{2j} + a_3 w_n^{3j} + a_4 w_n^{4j} + \ldots\ldots + a_{n-2} w_n^{(n-2)j} + a_{n-1} w_n^{(n-1)j}$$

$$= (a_0 + a_2 w_n^{2j} + a_4 w_n^{4j} + \ldots\ldots + a_{n-2} w_n^{(n-2)j})$$

$$+ (a_1 w_n^{j} + a_3 w_n^{3j} + a_5 w_n^{5j} + \ldots\ldots + a_{n-1} w_n^{(n-1)j})$$

$$= (a_0 + a_2 w_n^{2j} + a_4 w_n^{4j} + \ldots\ldots + a_{n-2} w_n^{(n-2)j})$$

$$+ w_n^{j} (a_1 + a_3 w_n^{2j} + a_5 w_n^{4j} + \ldots\ldots + a_{n-1} w_n^{(n-2)j})$$

算法的数学基础

$$A_j = (\underline{a_0 + a_2 w_n^{2j} + a_4 w_n^{4j} + \ldots\ldots + a_{n-2} w_n^{(n-2)j}})$$

奇数输入项的FFT

$$+ w_n^j (\underline{a_1 + a_3 w_n^{2j} + a_5 w_n^{4j} + \ldots\ldots + a_{n-1} w_n^{(n-2)j}})$$

偶数输入项的FFT

由于$w_n^2 = w_{n/2}$，以及$w_n^{n+k} = w_n^k$，

$$A_j = (\underbrace{a_0 + a_2 w_{n/2}^{j} + a_4 w_{n/2}^{2j} + \ldots\ldots + a_{n-2} w_{n/2}^{\frac{(n-2)j}{2}}}_{B_j})$$

将问题划分为两个子问题

$$+ w_n^j (\underbrace{a_1 + a_3 w_{n/2}^{j} + a_5 w_{n/2}^{2j} + \ldots\ldots + a_{n-1} w_{n/2}^{\frac{(n-2)j}{2}}}_{C_j})$$

于是，$A_j = B_j + w_n^j C_j$　还可证明，$A_{j+n/2} = B_j + w_n^{j+n/2} C_j$

分治算法过程

划分：将输入拆分成$a_0,a_2,\ldots,a_{n-2}$和$a_1,a_3,\ldots,a_{n-1}$.

递归求解：递归计算$a_0,a_2,\ldots,a_{n-2}$的变换$B_0,B_1,\ldots,B_{n/2-1}$

递归计算$a_1,a_3\ldots,a_{n-1}$的变换$C_0,C_1,\ldots,C_{n/2-1}$

合并：根据$A_j= B_j+C_j\cdot W_n^{\,j}\ (j<n/2)$ 和$A_j=B_{j-n/2}+C_{j-n/2}\cdot W_n^{\,j}$ $(n/2\le j<n-1)$依次求得$A_0, A_1, \ldots, A_{n-1}$.

分治算法过程

例如：计算8个点$a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7$的FFT.

计算4个点
a_0, a_2, a_4, a_6的FFT.

计算4个点
a_1, a_3, a_5, a_7的FFT.

计算2个点
a_0, a_4的FFT.

计算2个点
a_2, a_6的FFT.

计算2个点
a_1, a_5的FFT.

计算2个点
a_3, a_7的FFT.

算法FFT($a_0,a_2,\ldots,a_{n-1}$, n)

输入： $a_0,a_1,\ldots,a_{n-1}$， $n=2^k$

输出： $a_0,a_1,\ldots,a_{n-1}$的傅里叶变换$A_0,\ldots,A_{n-1}$

1. $W \leftarrow \exp(2\pi i/n)$;
2. If (n=2) Then
3、 $A_0 \leftarrow a_0+a_1$;
4. $A_1 \leftarrow a_0-a_1$;
5. 输出A_0,A_1,算法结束；
6. $B_0,B_1,\ldots,B_{n/2-1} \leftarrow$FFT($a_0,a_2,\ldots,a_{n-2}$, $n/2$)；
7. $C_0,C_1,\ldots,C_{n/2-1} \leftarrow$FFT($a_1,a_3,\ldots,a_{n-1}$, $n/2$)；
8. For j=0 To $n/2-1$
9. $A_j \leftarrow B_j+C_j \cdot W^j$;
10. $A_{j+n/2} \leftarrow B_j+C_j \cdot W^{j+n/2}$;
11. 输出$A_0,A_1,\ldots,A_{n-1}$,算法结束；

$T(n)=\theta(1)$ If $n=2$

$T(n)=2T(n/2)+\theta(n)$ If $n>2$

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

$$T(n)=\theta(1) \qquad \text{if } n\le c$$
$$T(n)=aT(n/b)+D(n)+C(n) \qquad \text{if } n>c$$

3.8 整数乘法

优化划分阶段，降低$T(n)=aT(n/b)+f(n)$中的a

问题定义

输入：n位二进制整数X和Y
输出：X和Y的乘积

通常，计算X*Y时间复杂性为$O(n^2)$，
我们给出一个复杂性为$O(n^{1.59})$的算法。

算法的思想

	n/2位	n/2位
X=	A	B

	n/2位	n/2位
Y=	C	D

$$XY = (A2^{n/2} + B)(C2^{n/2} + D)$$
$$= AC2^{n} + AD2^{n/2} + BC2^{n/2} + BD$$
$$= AC2^{n} + ((A-B)(D-C)+AC+BD)2^{n/2} + BD$$

如此计算需要

$T(n)=4T(n/2)+O(n)=O(n^2)$

时间复杂性

$T(n)=\theta(1)$ if $n=1$

$T(n)=3T(n/2)+O(n)$ if $n>1$

使用Master定理

$T(n)=O(n^{\log_2 3})=O(n^{1.59})$