

# The Divide-and-Conquer Strategy

個個擊破法

# Outlines

- **The 2-Dimensional Maxima Finding Problem**
- **The Closest Pair Problem**
- **The Convex Hull Problem**
- **The Voronoi Diagrams Constructed by the Divide-and-Conquer Strategy**
- **Applications of the Voronoi Diagrams**
- **Matrices Multiplication**

# Introduction

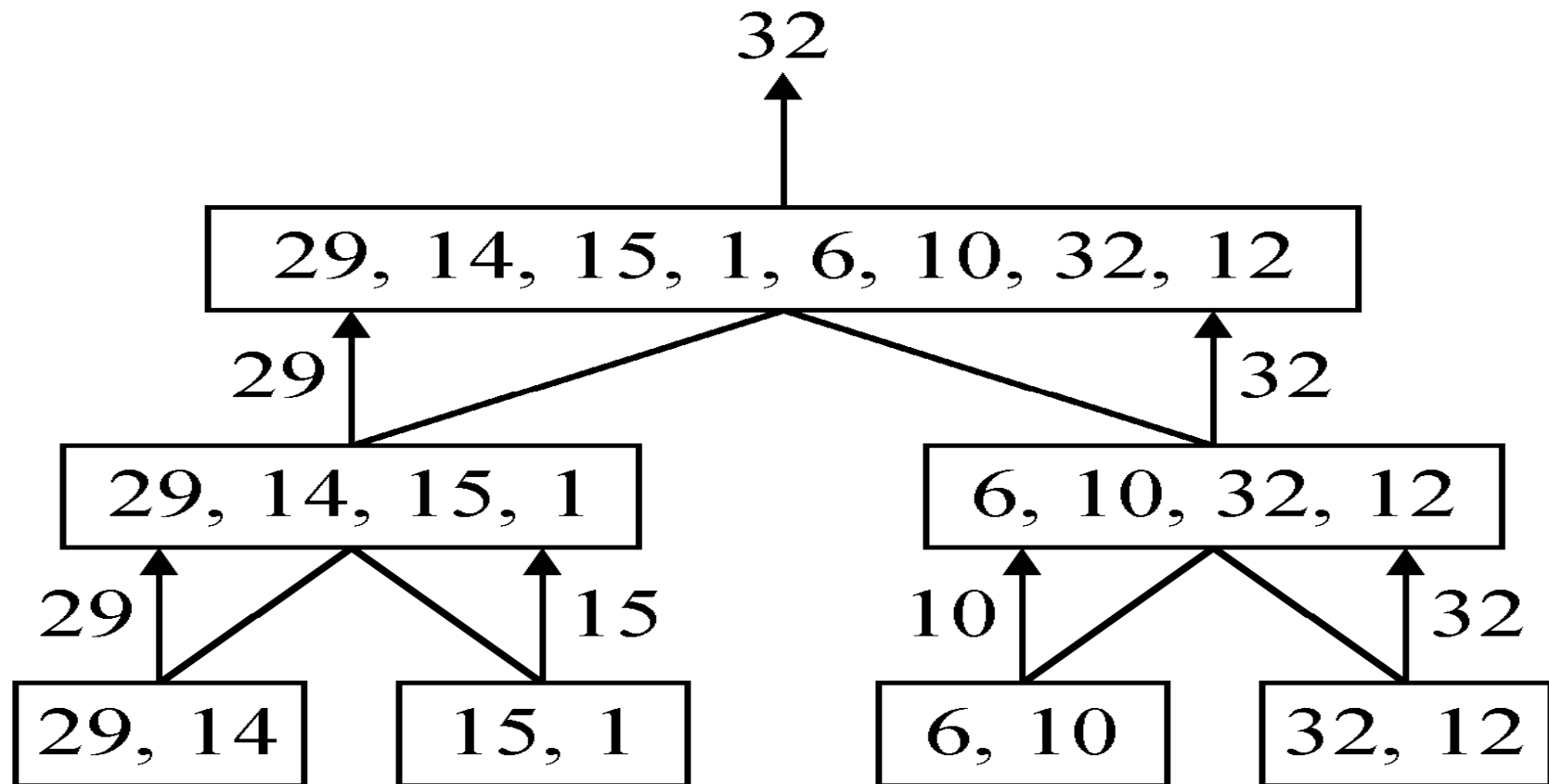
- Divide-and-conquer strategy
  - first **divides** a problem into two smaller sub-problems and each sub-problem is identical to its original problem, except its **input size is smaller**.
  - Both sub-problems are solved and the sub-solutions are finally **merged** into the final solution.
- These two sub-problems themselves can be solved by the divide-and-conquer strategy again.
- Or, to put it in another way, these two sub-problems are *solved recursively*.

# A simple example

- **Finding the maximum of a set  $S$  of  $n$  numbers.**
- Dividing the input into two sets, each set consisting of  $n/2$  numbers.
- Let us call these two sets  $S_1$  and  $S_2$ .
- Find the maximums of  $S_1$  and  $S_2$  respectively.
- Let the maximum of  $S_i$  be denoted as  $X_i$ ,  $i = 1, 2$ .
- Then the maximum of  $S$  can be found by comparing  $X_1$  and  $X_2$ . Whichever is the larger is the maximum of  $S$ .

# A simple example

- **Problem:** Finding the **maximum** of a set  $S$  of  $n$ .



# Time Complexity

- In general, the complexity  $T(n)$  of a divide-and-conquer algorithm is determined by the following formulas:

$$T(n) = \begin{cases} 2T(n/2) + S(n) + M(n) & , n \geq c \\ b & , n < c \end{cases}$$

- where
  - $S(n)$  denotes the time steps needed to **split** the problem into two sub-problems,
  - $M(n)$  denotes the time steps needed to **merge** two sub-solutions and
  - $b$  is a constant.

$$S(n)=M(n)=1, c=2, b=1$$

- Time complexity:

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

- Calculation of  $T(n)$ :

Assume  $n = 2^k$ ,

$$\begin{aligned} T(n) &= 2T(n/2)+1 \\ &= 2(2T(n/4)+1)+1 \\ &= 4T(n/4)+2+1 \end{aligned}$$

⋮

$$\begin{aligned} &= 2^{k-1}T(2)+2^{k-2}+\dots+4+2+1 \\ &= 2^{k-1}+2^{k-2}+\dots+4+2+1 \\ &= 2^k-1 = n-1 \end{aligned}$$

# A general divide-and-conquer algorithm

Step1: If the problem size is **small**, solve this problem directly; otherwise, **split** the original problem into **2** sub-problems with equal sizes.

Step2: **Recursively** solve these 2 sub-problems by applying this algorithm.

Step3: **Merge** the solutions of the 2 sub-problems into a solution of the original problem.

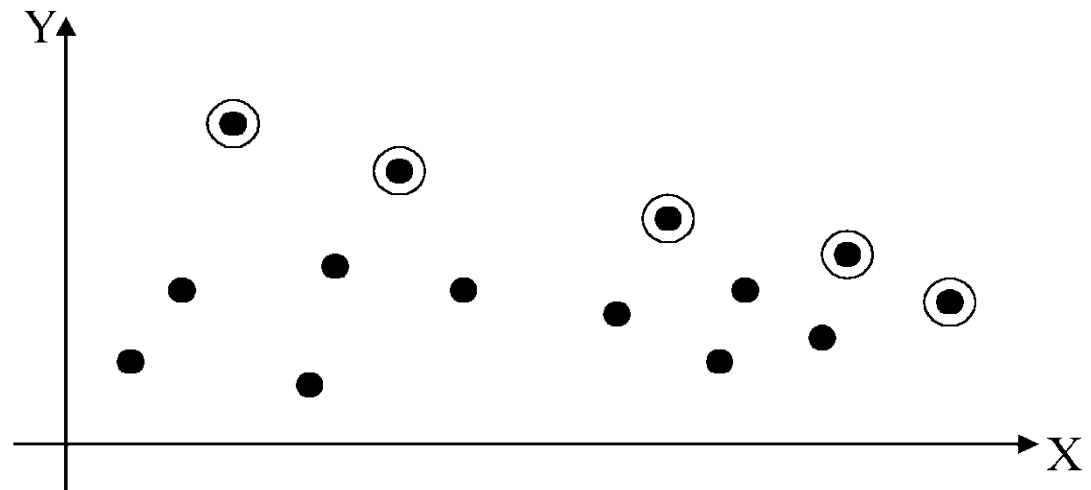


# 2-D maxima finding problem

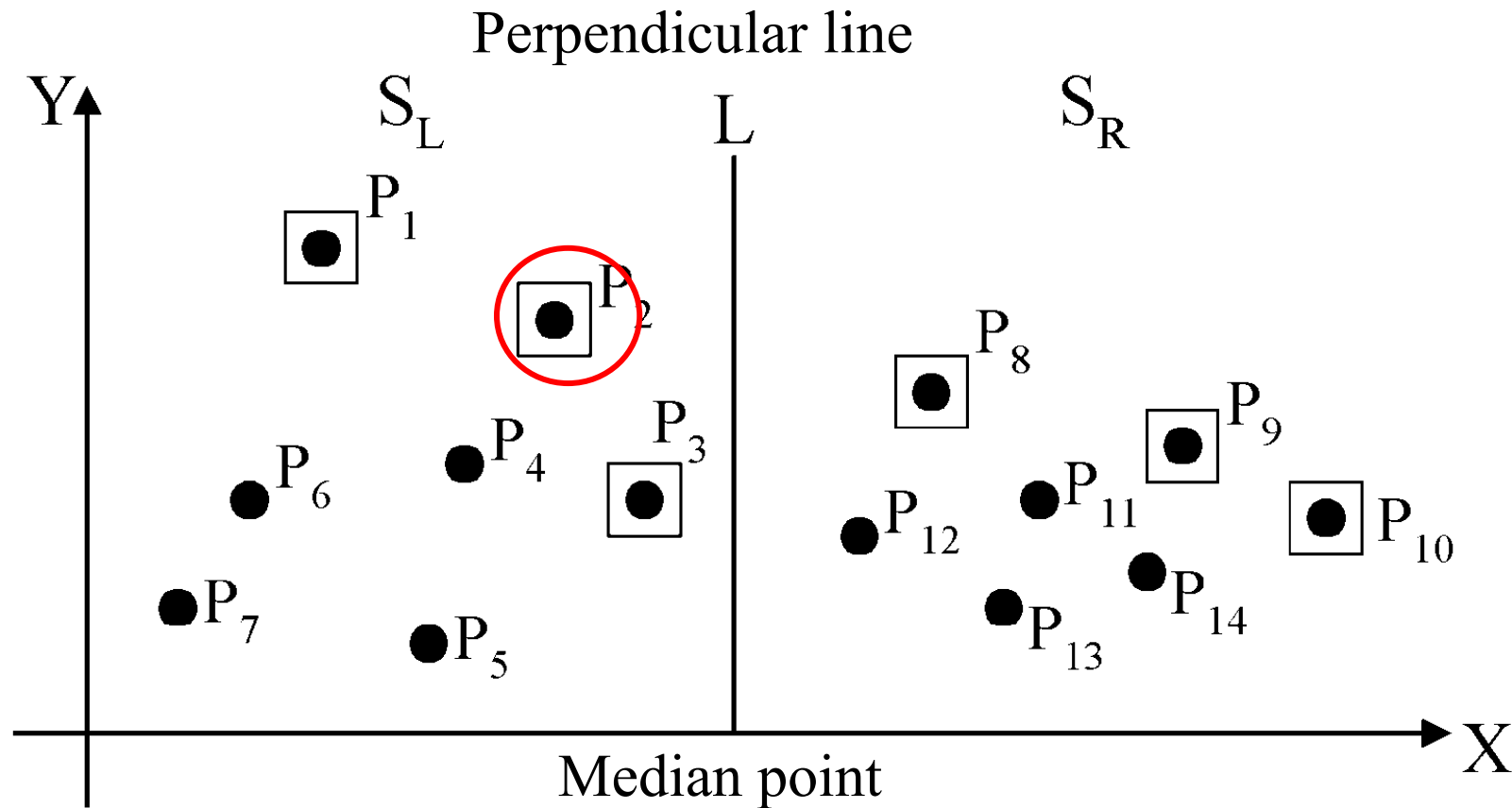
## 2-D maxima finding problem

- **Def** : A point  $(x_1, y_1)$  dominates  $(x_2, y_2)$  if  $x_1 > x_2$  and  $y_1 > y_2$ . A point is called a maxima if no other point dominates it
- **Maxima finding problem**: find the maximal points among these  $n$  points.
- Straightforward method : Compare every pair of points.

Time complexity:  
 $O(n^2)$



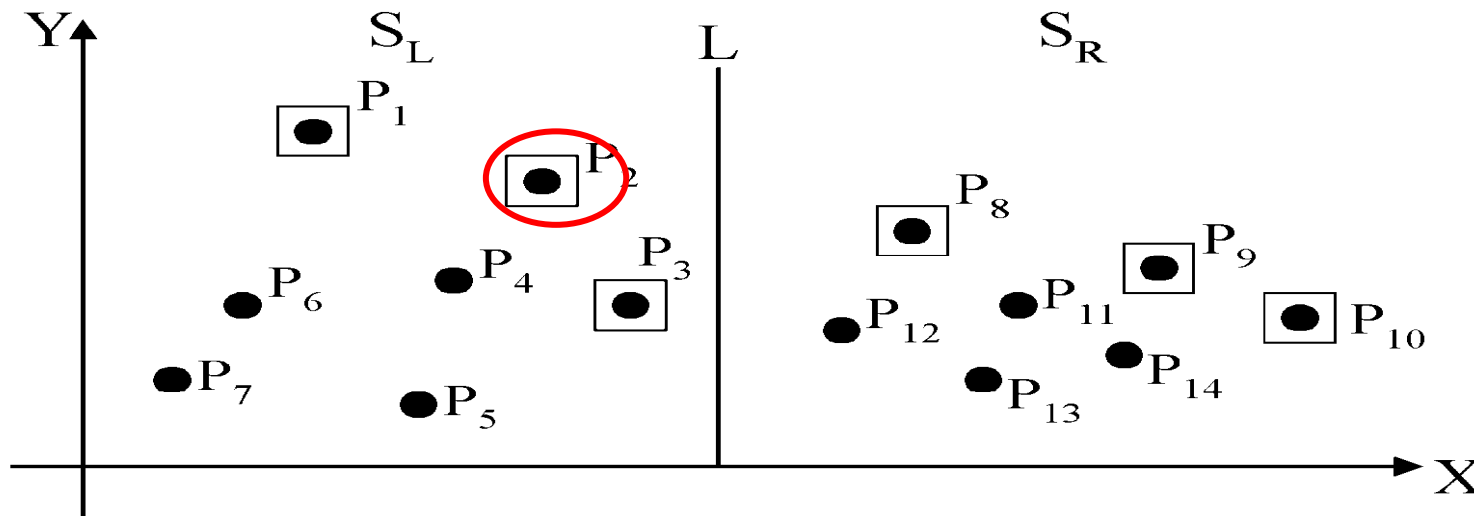
# Divide-and-conquer for maxima finding



The maximal points of  $S_L$  and  $S_R$

# Merge Step

- The merging process is rather simple.
- Since the **x-value** of a point in  $S_R$  is always larger than the x-value of every point in  $S_L$ .
- A point in  $S_L$  is a maxima if and only if its y-value is **not less** than the y-value of a maxima of  $S_R$ .



## The algorithm:

- Input: A set of  $n$  planar points.
- Output: The maximal points of  $S$ .

Step 1: If  $S$  contains only one point, return it as the maxima. Otherwise, find a **line**  $L$  perpendicular to the X-axis which separates the set of points into two subsets  $S_L$  and  $S_R$ , each of which consisting of  $n/2$  points.

Step 2: **Recursively** find the maximal points of  $S_L$  and  $S_R$ .

Step 3: Find the **largest y**-value of  $S_R$ . Project the maximal points of  $S_L$  onto  $L$ . Discard each of the maximal points of  $S_L$  if its y-value is less than the largest y-value of  $S_R$ .

- Time complexity:  $T(n)$

Step 1:  $O(n)$  median finding for **line L**

Step 2:  $2T(n/2)$

Step 3:  $O(n \log n)$  :sorting  $n$  points according to their  $y$ -value.

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

Assume  $n = 2^k$

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

# Improvement

- We note that our divide-and-conquer strategy is dominated by sorting in the merging steps.
- Somehow we are not doing a very efficient job because **sorting should be done once and for all**.
- That is, we should **conduct a presorting  $O(n \log n)$** .
- If this is done, the merging complexity is  $O(n)$  and the total number of time steps needed is  **$O(n \log n) + T(n)$**

where

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

- and  $T(n)$  can be easily found to be  $O(n \log n)$ . Thus the total time-complexity of using the divide-and-conquer strategy to find maximal points with presorting is  **$O(n \log n)$** .

# Recurrence



# Recurrence

**Recurrence:** an equation that describes a function in terms of its value on smaller functions

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

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

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

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

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

# Solve Recurrence

- **Iteration Method**
- **Recursion-Tree Method**
- **Master Theorem**

# Iteration Method

- Expand the recurrence (substitute some terms)
- Work some algebra to express as a summation
- Evaluate the summation

## Iteration Method

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

- $T(n) = c + T(n-1)$   
 $= c + c + T(n-2) = 2c + T(n-2)$   
 $= 2c + c + T(n-3) = 3c + T(n-3)$   
 $= \dots$   
 $= kc + T(n-k)$                       Set  $k = n$   
 $= nc + T(0) = nc$

$T(n) = \Theta(n)$

## Iteration Method

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

- $T(n) = n + T(n-1)$   
 $= n + n-1 + T(n-2)$   
 $= n + n-1 + n-2 + T(n-3)$   
 $= \dots$   
 $= n + n-1 + n-2 + \dots + n-(k-1) + T(n-k)$   
 $= n + n-1 + n-2 + \dots + 1 + T(0)$   
 $= \frac{n(n+1)}{2}$

$T(n) = \Theta(n^2)$

# Recursion-Tree Method

# Recursion-Tree Method

- Expand the recurrence
- Construct a recursion-tree
- Sum the costs

# Merge Sort

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

Rewrite:

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

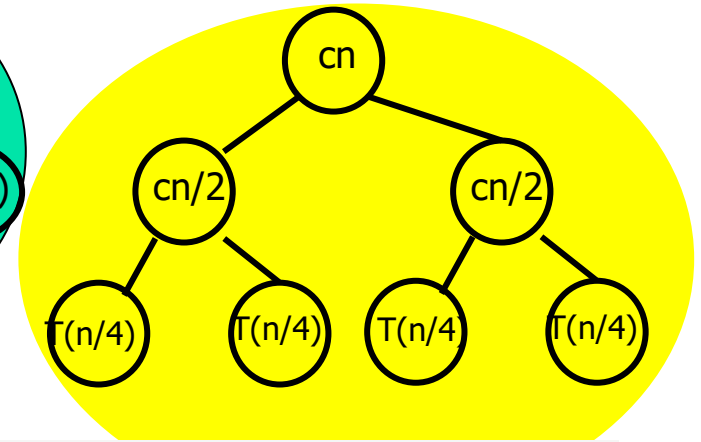
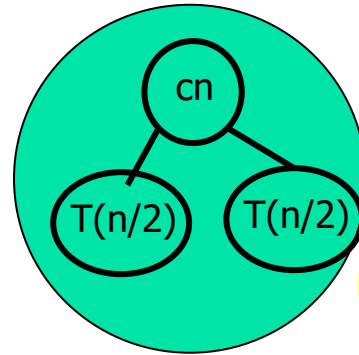


Becomes A Recursion Tree



# Merge Sort (Recursion Tree)

$T(n)$



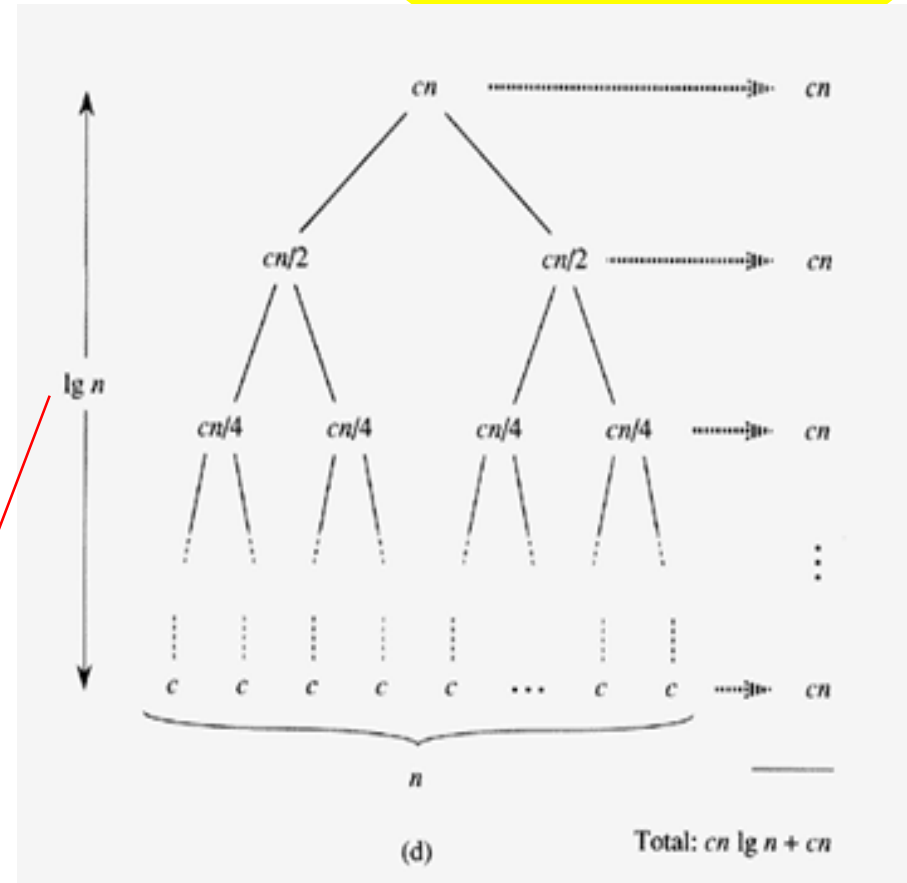
$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + cn & n > 1 \end{cases}$$

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

$$= cn + 2\left(\frac{cn}{2}\right) + 4T(n/4)$$

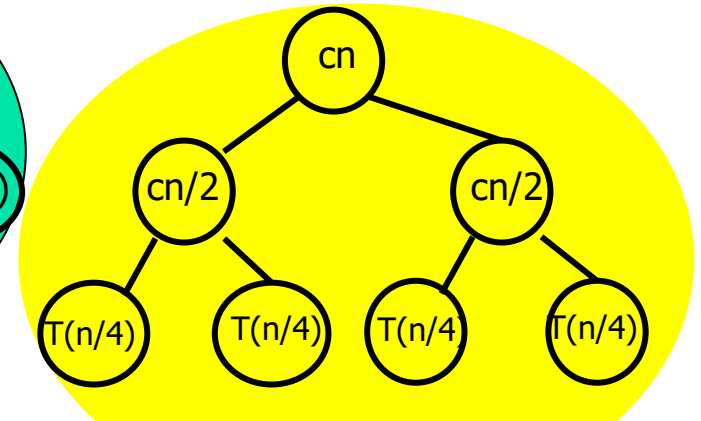
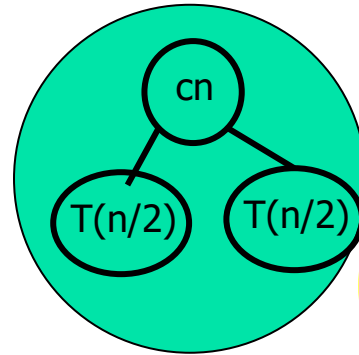
$$= cn + 2\left(\frac{cn}{2}\right) + 4\left(\frac{cn}{4}\right) + \dots?$$

when  $\frac{n}{2^i} = 1, i = \lg n$



# Merge Sort (Recursion Tree)

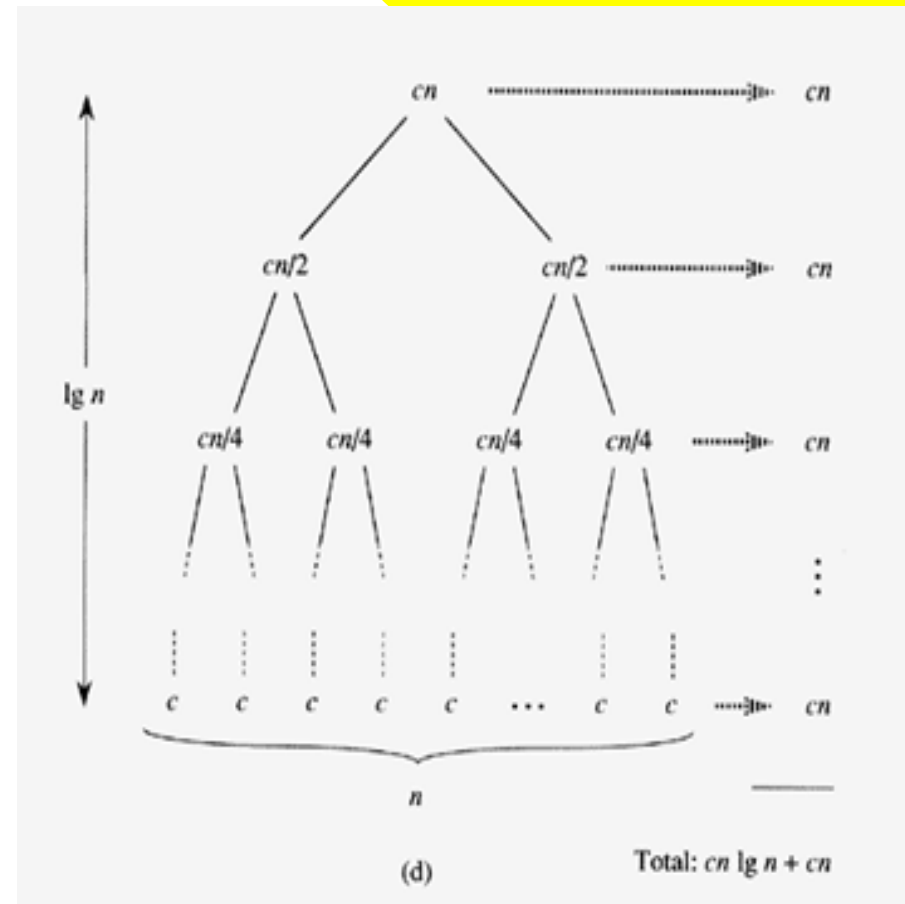
$T(n)$



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

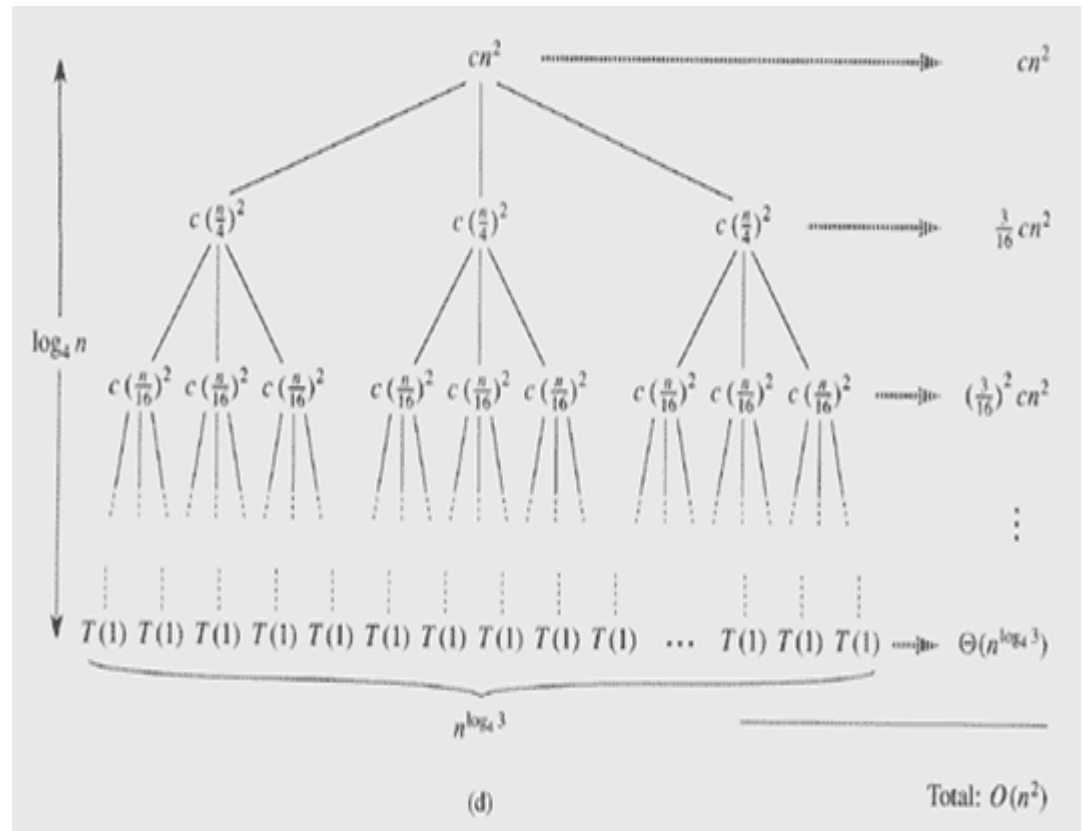
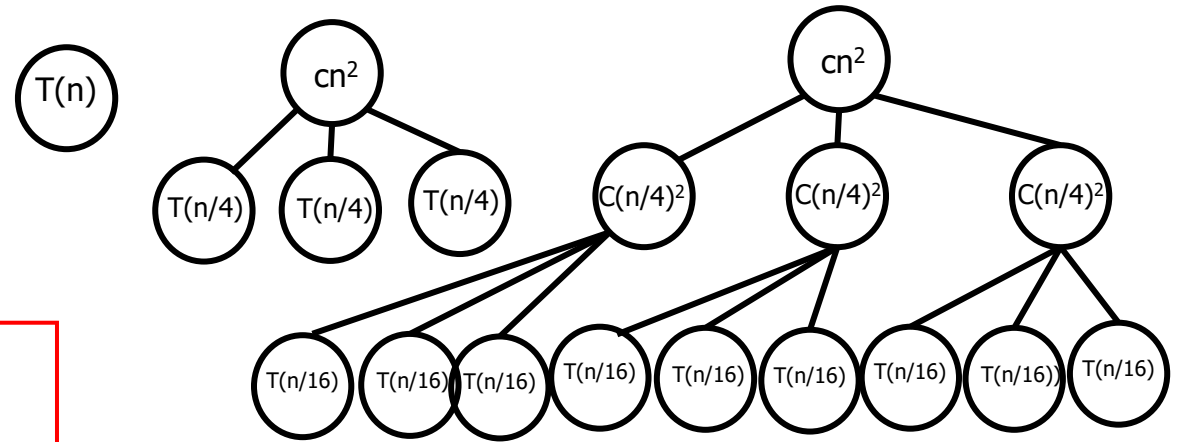
$$T(n) = cn \lg n + cn$$

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



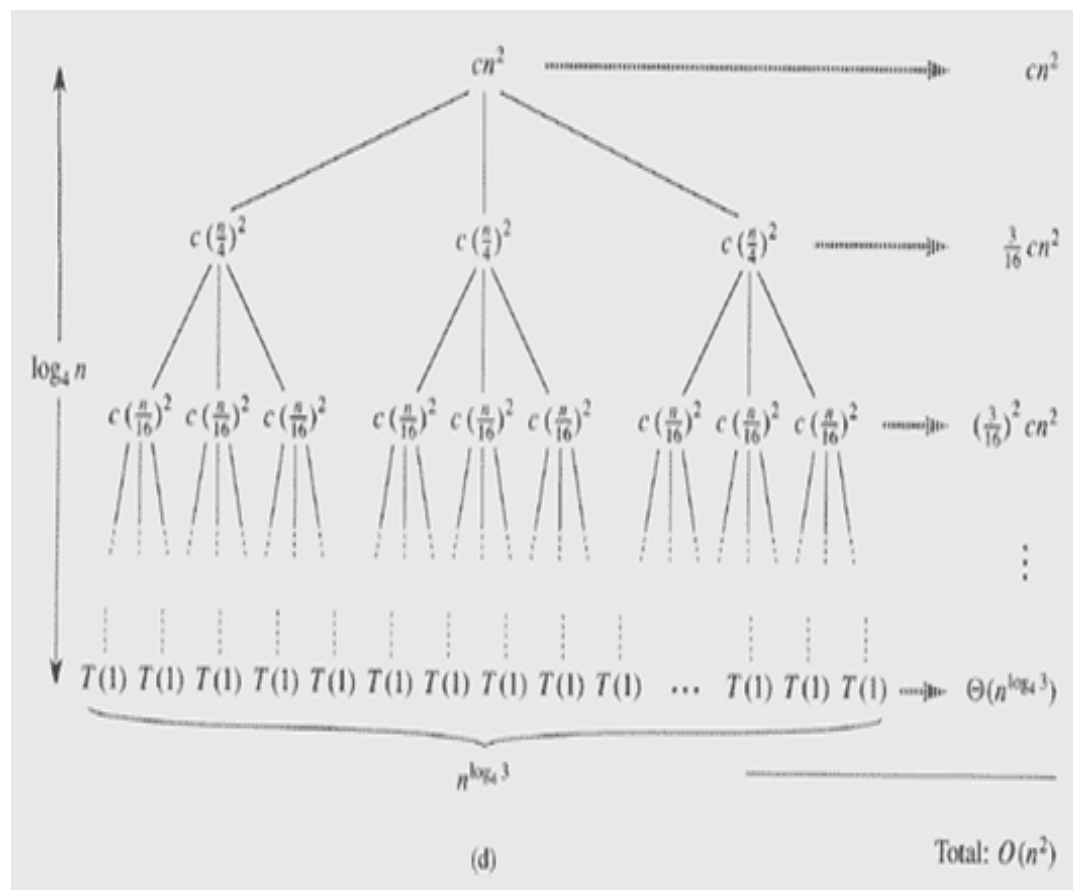
# Recursion-Tree Method (Example)

$$\begin{aligned}
 T(n) &= 3T(n/4) + cn^2 \\
 &= cn^2 + 3T(n/4) \\
 &= cn^2 + \left(\frac{3}{16}\right)cn^2 + 9T(n/16) \\
 &= cn^2 + \left(\frac{3}{16}\right)cn^2 + \left(\frac{3}{16}\right)^2 cn^2 \\
 &+ \dots + \left(\frac{3}{16}\right)^{i-1} cn^2 + 3^i T(n/4^i) \\
 \frac{n}{4^i} &= 1, i = \log_4 n
 \end{aligned}$$



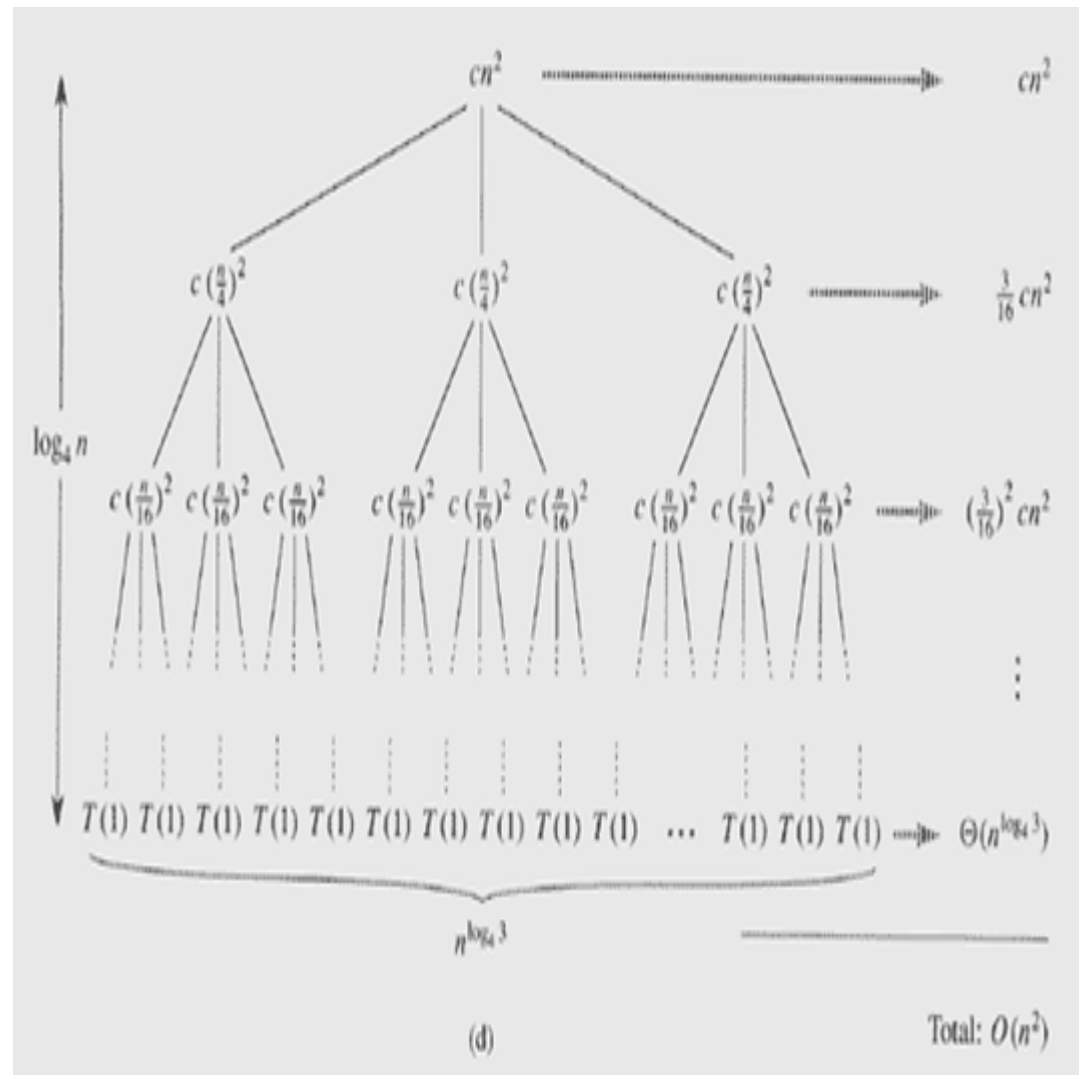
## Recursion-Tree Method (Example)

$$\begin{aligned}
 T(n) &= 3T(n/4) + cn^2 \\
 &= cn^2 + 3T(n/4) \\
 &= cn^2 + \left(\frac{3}{16}\right)cn^2 + 9T(n/16) \\
 &= cn^2 + \left(\frac{3}{16}\right)cn^2 + \left(\frac{3}{16}\right)^2 cn^2 \\
 &+ \dots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + 3^{\log_4 n} T(1) \\
 &\quad \swarrow \\
 3^{\log_4 n} T(1) &= n^{\log_4 3} \Theta(1) \\
 &= \Theta(n^{\log_4 3})
 \end{aligned}$$

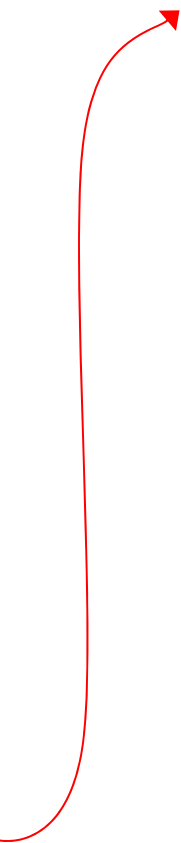


# Recursion-Tree Method (Example)

$$\begin{aligned}
 T(n) &= 3T(n/4) + cn^2 \\
 &= cn^2 + 3T(n/4) \\
 &= cn^2 + \left(\frac{3}{16}\right)cn^2 + 9T(n/16) \\
 &= cn^2 + \left(\frac{3}{16}\right)cn^2 + \left(\frac{3}{16}\right)^2 cn^2 \\
 &+ \dots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + 3^{\log_4 n} T(1) \\
 &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})
 \end{aligned}$$



## Recursion-Tree Method (Example)

$$\begin{aligned}T(n) &= 3T(n/4) + cn^2 \\&= cn^2 + 3T(n/4) \\&= cn^2 + \left(\frac{3}{16}\right)cn^2 + 9T(n/16) \\&= cn^2 + \left(\frac{3}{16}\right)cn^2 + \left(\frac{3}{16}\right)^2 cn^2 \\&\quad + \dots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + 3^{\log_4 n} T(1) \\&= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})\end{aligned}$$

$$= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3})$$

Recall

$$\boxed{\sum_{i=0}^n r^i = \frac{r^{n+1} - 1}{r - 1}}$$

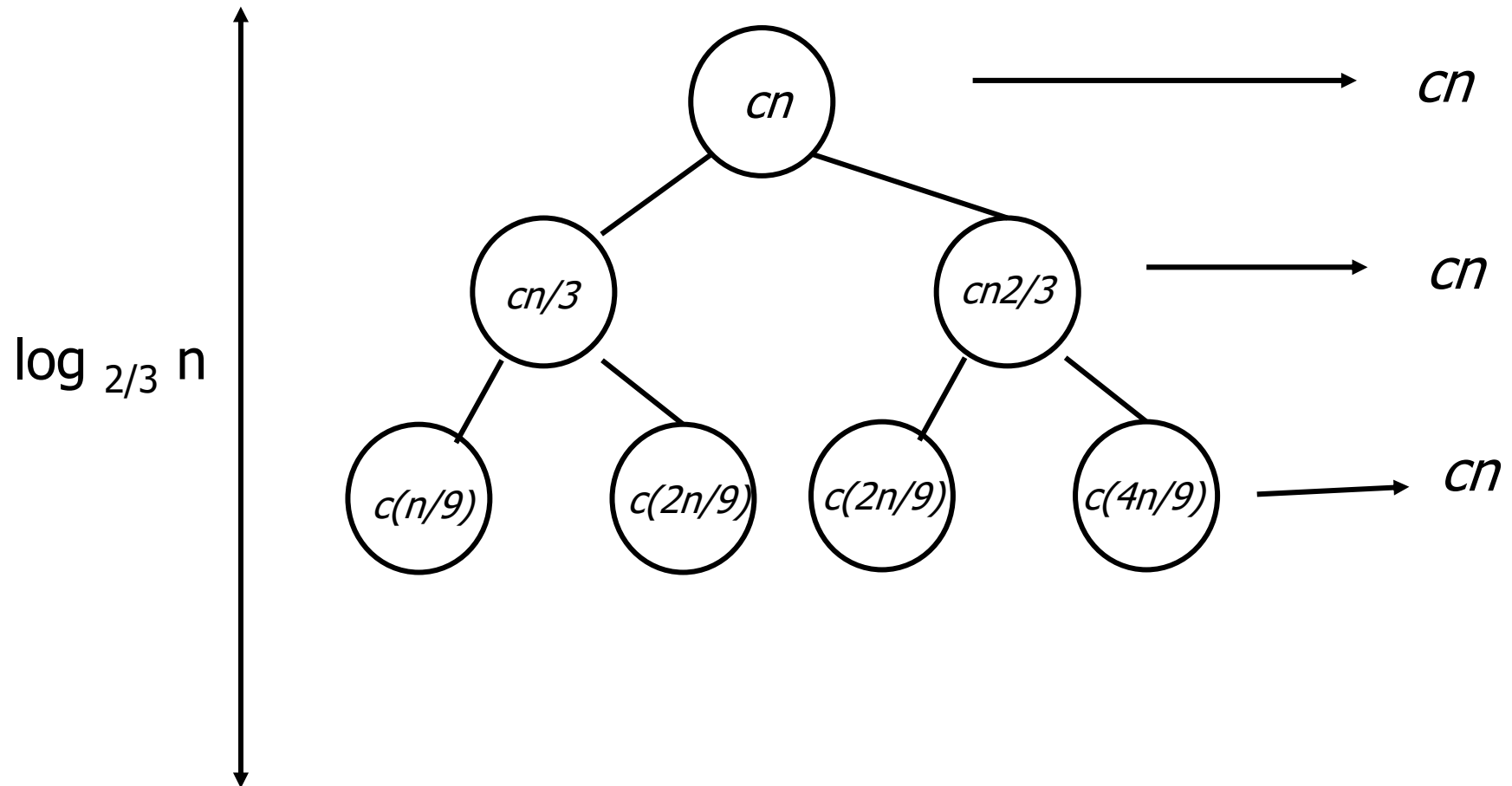
## Recursion-Tree Method (Example)

$$\begin{aligned}T(n) &= 3T(n/4) + cn^2 \\&= cn^2 + 3T(n/4) \\&= cn^2 + \left(\frac{3}{16}\right)cn^2 + 9T(n/16) \\&= cn^2 + \left(\frac{3}{16}\right)cn^2 + \left(\frac{3}{16}\right)^2 cn^2 \\&\quad + \dots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + 3^{\log_4 n} T(1) \\&= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})\end{aligned}$$

$$\begin{aligned}&= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}) \\&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\&= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) \\&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3})\end{aligned}$$

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

$$T(n) = T(n/3) + T(2n/3) + cn$$



Total:  $O(n \log n)$



# Master Theorem

# Master Theorem\*\*\*

- Provide a “**cookbook**” method for solving recurrences
- Divide-and-conquer algorithm

An algorithm that divides the problem of size  $n$  into  $a$  subproblems, each of size  $n / b$

# Master Theorem

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

$$a \geq 1 \text{ and } b > 1$$

1. If  $f(n) = O(n^{\log_b a - \varepsilon})$ , then  $T(n) = \Theta(n^{\log_b a})$

2. If  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \lg n)$

3. If  $f(n) = \Omega(n^{\log_b a + \varepsilon})$ , and if  $af(n/b) \leq cf(n)$ ,  
then  $T(n) = \Theta(f(n))$

$$\varepsilon > 0, c < 1$$

- "**Polynomially larger**" means that the ratio of the functions **falls between two polynomials**, asymptotically.
- Specifically,  $f(n)$  is polynomially greater than  $g(n)$  **if and only if** there exist generalized polynomials (**fractional exponents** are allowed)  $p(n)$ ,  $q(n)$  such that the following inequality holds asymptotically:

$$p(n) \leq \frac{f(n)}{g(n)} \leq q(n)$$

- Example:

- $(n, n \log n), (n^2, n \log n)$  (yes)

For the second problem, we have the ratio is equal to  $n \log(n)$ . It is the case that  $n \leq n \log(n) \leq n^2$  asymptotically, so it is polynomially bounded and therefore  $n^2$  is polynomially larger.

$\frac{n^2}{n \log(n)} = \frac{n}{\log(n)}$ , and we have that (asymptotically)

$$n^{\frac{1}{3}} \leq \frac{n}{\log(n)} \leq n$$

# Notes on Master Theorem

- Some technicalities:

In case 1,  $f(n)$  must be *polynomially smaller* than  $n^{\log_b a}$  by a factor of  $n^\varepsilon$ ,  $\varepsilon > 0$

In case 3,  $f(n)$  must be *polynomially larger* than  $n^{\log_b a}$  by a factor of  $n^\varepsilon$ ,  $\varepsilon > 0$

- The three cases doesn't cover all possibilities of  $f(n)$ .
- *Can't use Master Theorem*

# Examples

**Meaning of polynomial differences.**  $f(n)$  is polynomially smaller than  $g(n)$  if  $f(n) = O(g(n)/n^\epsilon)$  for some  $\epsilon > 0$ .  $f(n)$  is polynomially larger than  $g(n)$  if  $f(n) = \Omega(g(n)n^\epsilon)$  for some  $\epsilon > 0$ . We will never say  $f(n)$  is polynomially equal to  $g(n)$ .

Here are some examples.

- $f(n) = 1$  and  $g(n) = n^2$ . Then  $f(n)$  is polynomially smaller than  $g(n)$ . This is what you believed and it is correct.
- $f(n) = g(n) = n^2$ . Then  $f(n)$  is not polynomially smaller nor polynomially larger than  $g(n)$ .
- $f(n) = n^{1+\frac{1000}{\log n}}$  and  $g(n) = n^2$ . Then  $f(n)$  is polynomially smaller than  $g(n)$ .
- $f(n) = \frac{n}{\log n}$  and  $g(n) = n$ . Then  $f(n)$  is not polynomially smaller nor polynomially larger than  $g(n)$ .

$$T(n) = 4T\left(\frac{n}{2}\right) + 1$$

$a = 4, b = 2, n^{\log_2 4} = n^2$  is polynomial larger than the constant function 1. This is case 1 of the master's theorem.

$$T(n) = 4T\left(\frac{n}{2}\right) + \frac{n^2}{\log n}$$

$a = 4, b = 2, \frac{n^2}{\log n}$  is not polynomially smaller nor polynomially larger than  $n^2$ . None of the three cases of master's theorem can be applied. However, the extension of case 2, case 2b can be applied.

<https://cs.stackexchange.com/questions/105149/meaning-of-polynomially-larger-or-smaller-in-the-context-of-the-master-method>

## Using the Master Method

$$T(n) = aT(n/b) + f(n)$$
$$a \geq 1 \text{ and } b > 1$$

$$T(n) = 9T(n/3) + n$$

$$a = 9, b = 3, f(n) = n$$

$$n^{\log_b a} = n^{\log_3 9} = n^2 = \Theta(n^2)$$

$$f(n) = O(n^{\log_3 9 - \varepsilon}), \text{ where } \varepsilon = 1$$

Use Case 1: If  $f(n) = O(n^{\log_b a - \varepsilon})$ , then  $T(n) = \Theta(n^{\log_b a})$

$$T(n) = \Theta(n^2)$$

## Using the Master Method

$$T(n) = aT(n/b) + f(n)$$
$$a \geq 1 \text{ and } b > 1$$

$$T(n) = T(2n/3) + 1$$

$$a = 1, b = 3/2, f(n) = 1$$

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$$

$$f(n) = \Theta(n^{\log_b a}) = \Theta(1)$$

Use Case 2: *If  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \lg n)$*

$$T(n) = \Theta(\lg n)$$



## Using the Master Method

$$T(n) = aT(n/b) + f(n)$$
$$a \geq 1 \text{ and } b > 1$$

$$T(n) = 3T(n/4) + n \lg n$$

$$a = 3, b = 4, f(n) = n \lg n$$

$$n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$$

$$f(n) = \Omega(n^{\log_4 3 + \varepsilon}), \text{ where } \varepsilon \approx 0.2$$

$$af(n/b) = 3(n/4) \lg(n/4) \leq (3/4)n \lg n = cf(n), \text{ for } c = 3/4$$

Use Case 3: If  $f(n) = \Omega(n^{\log_b a + \varepsilon})$ , and if  $af(n/b) \leq cf(n)$ ,  
then  $T(n) = \Theta(f(n))$

$$T(n) = \Theta(n \lg n)$$

## Using the Master Method

$$T(n) = aT(n/b) + f(n)$$
$$a \geq 1 \text{ and } b > 1$$

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

$$a = 2, b = 2, f(n) = n \lg n$$

But  $f(n) = n \lg n$  is not polynomially larger than  $n^{\log_b a} = n$

$f(n) / n^{\log_b a} = (n \lg n) / n = \lg n$  is asymptotically less than  $n^\epsilon, \epsilon > 0$

**Master Method doesn't Apply**

## Extended Master Method

$$T(n) = aT(n/b) + f(n)$$
$$a \geq 1 \text{ and } b > 1$$

If  $f(n) = \Theta(n^{\log_b a} \lg^k n)$ ,  $k \geq 0$ ,  
then  $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$

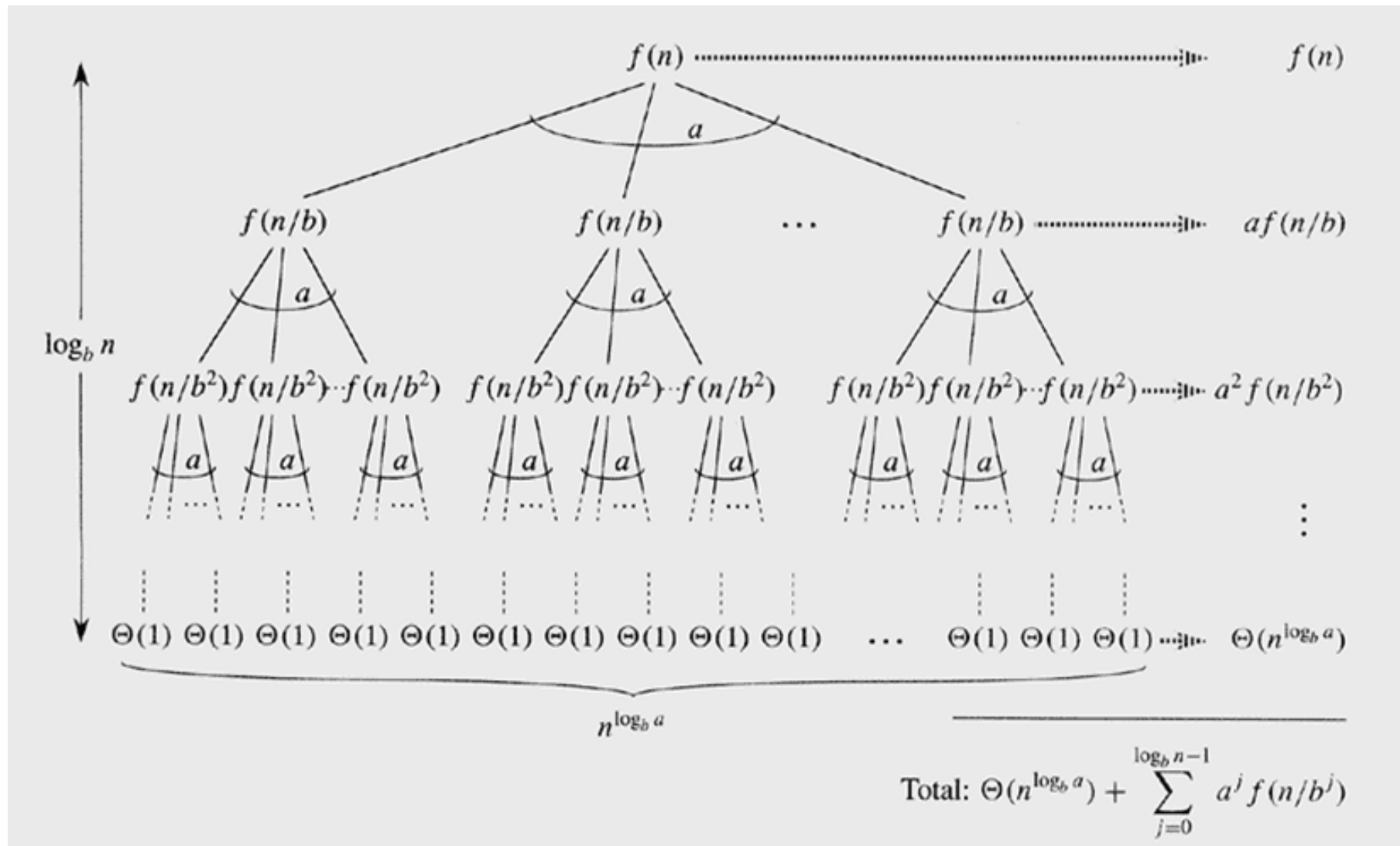
Back to the previous recurrence:

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

$$f(n) = \Theta(n^{\log_b a} \lg n), \text{ and } k = 1$$

$$T(n) = \Theta(n \lg^2 n)$$

# Proof of Master Theorem (Lemma 4.2)



# The closest pair problem

# The closest pair problem

- Given a set  $S$  of  $n$  points, find a pair of points which are closest together.

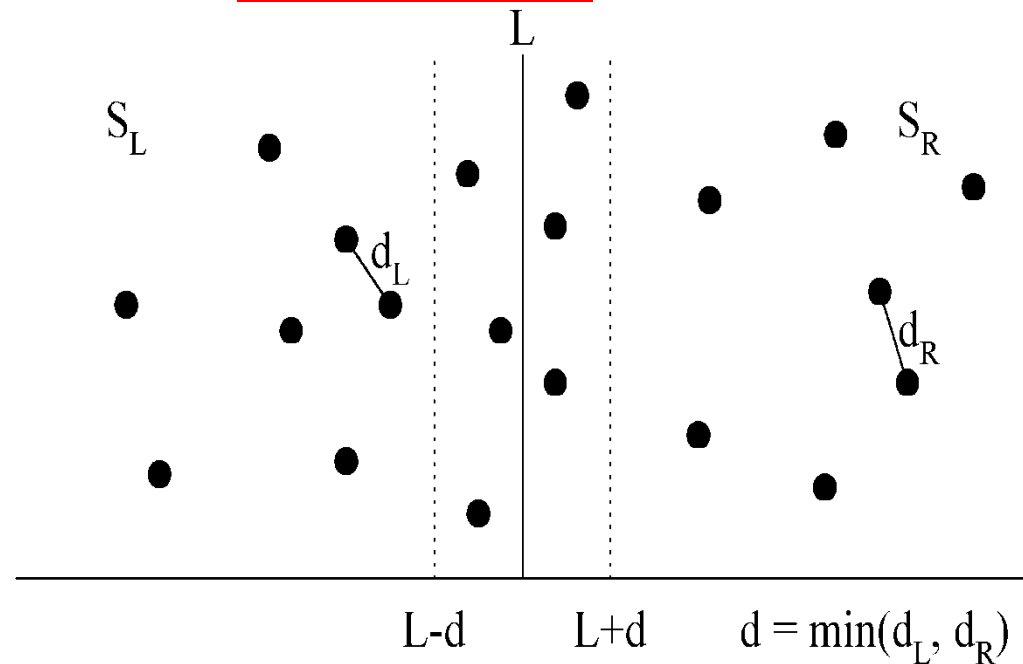
- 1-D version :

Solved by sorting

Time complexity :

$O(n \log n)$

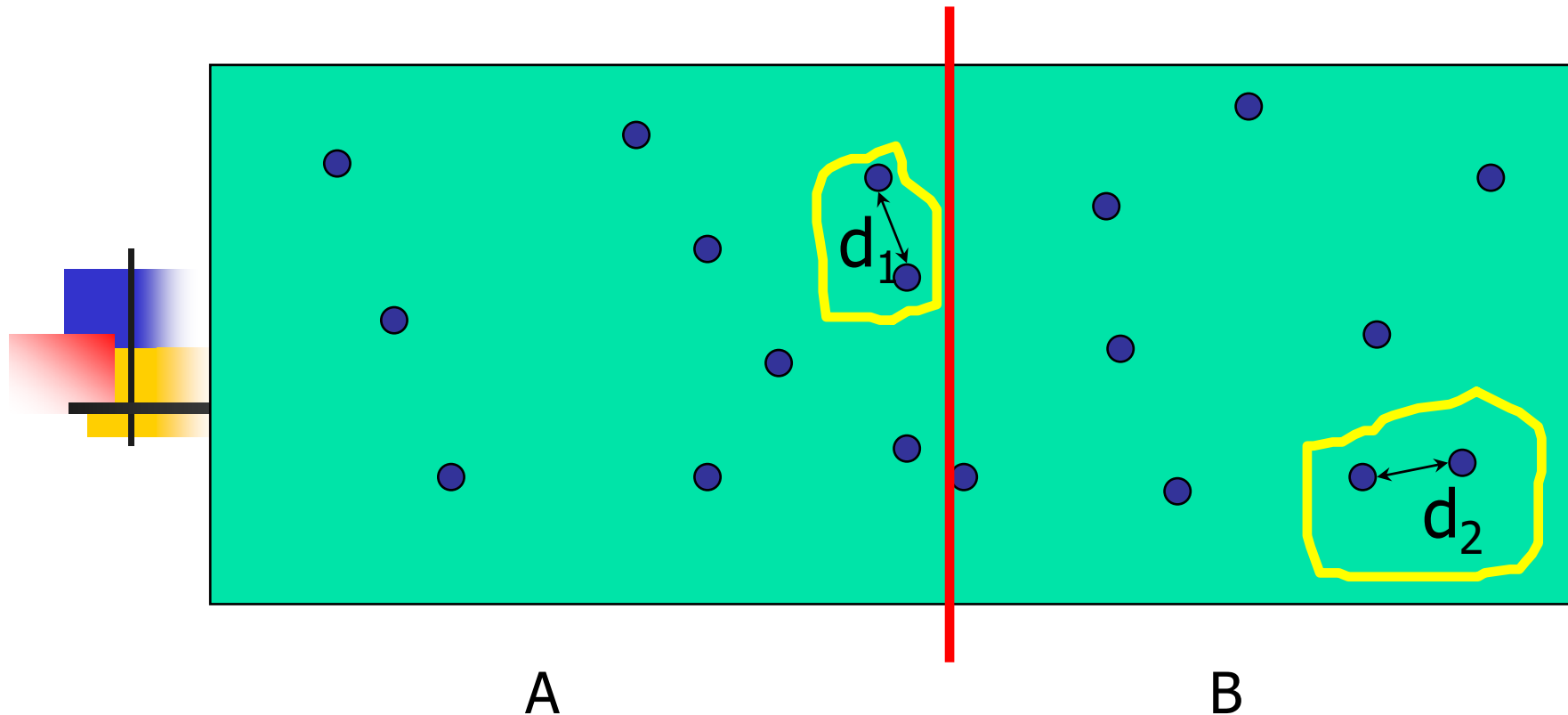
- 2-D version



# Divide-and-Conquer Solution

- We first partition the set  $S$  into  $S_L$  and  $S_R$ , and the number of points in  $S_L$  is equal to that in  $S_R$ .
- Find a vertical line  $L$  perpendicular to the x-axis such that  $S$  is cut into two equal sized subsets.
- Solving the closest pair problems in  $S_L$  and  $S_R$  respectively, we shall obtain  $d_L$  and  $d_R$  where  $d_L$  and  $d_R$  denote the distances of the closest pairs in  $S_L$  and  $S_R$  respectively.
- Let  $d = \min(d_L, d_R)$ .
- If the closest pair  $(P_a, P_b)$  of  $S$  consists of a point in  $S_L$  and a point in  $S_R$ , then  $P_a$  and  $P_b$  must lie within a *slab* centered at line  $L$  and bounded by lines  $L-d$  and  $L+d$ .
- **Merge Step**: examine points in slab.

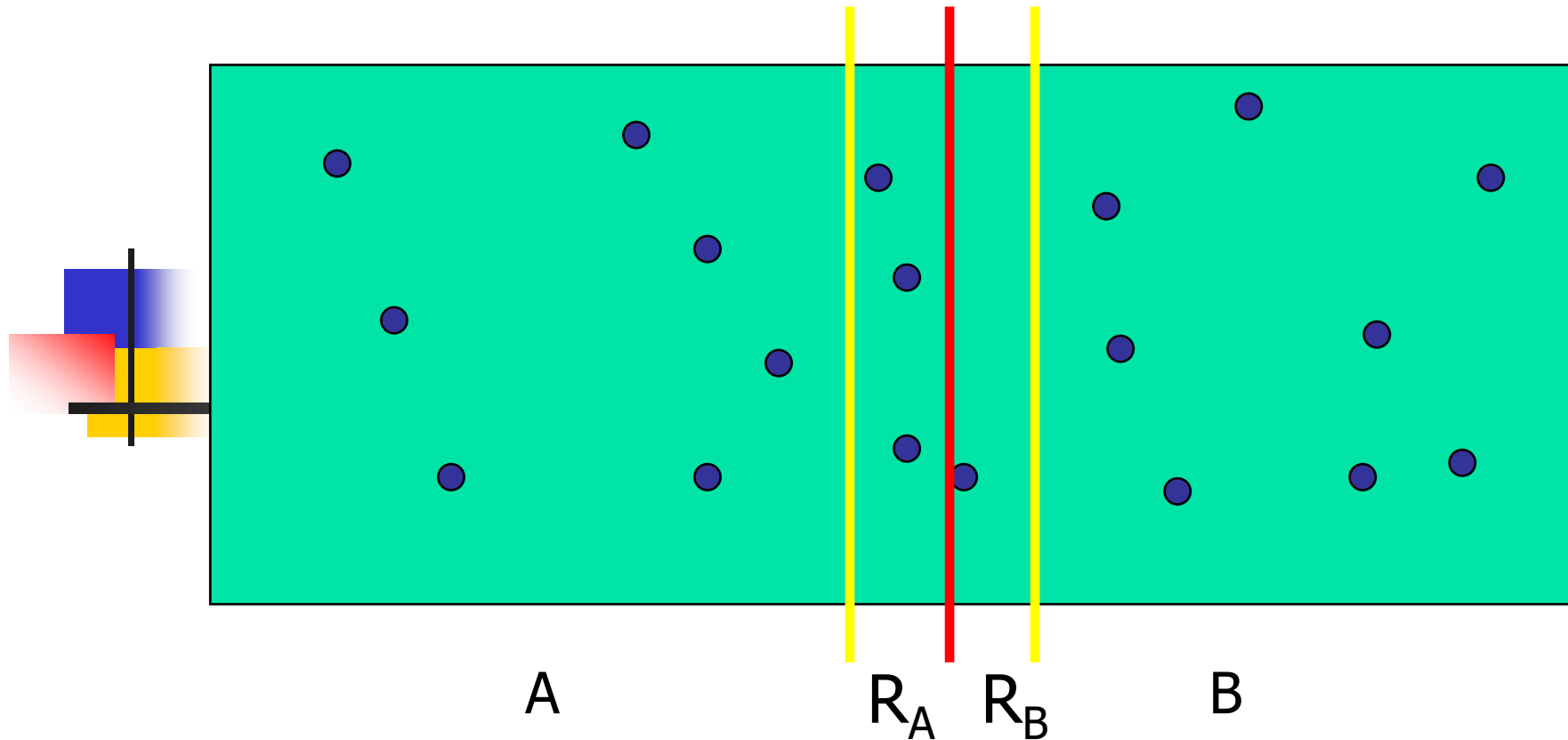
# Example



- Find  $L$  and the closest pair in A and B.
- Let  $d_1, d_2$  be the distance between the points in this pair.



# Example

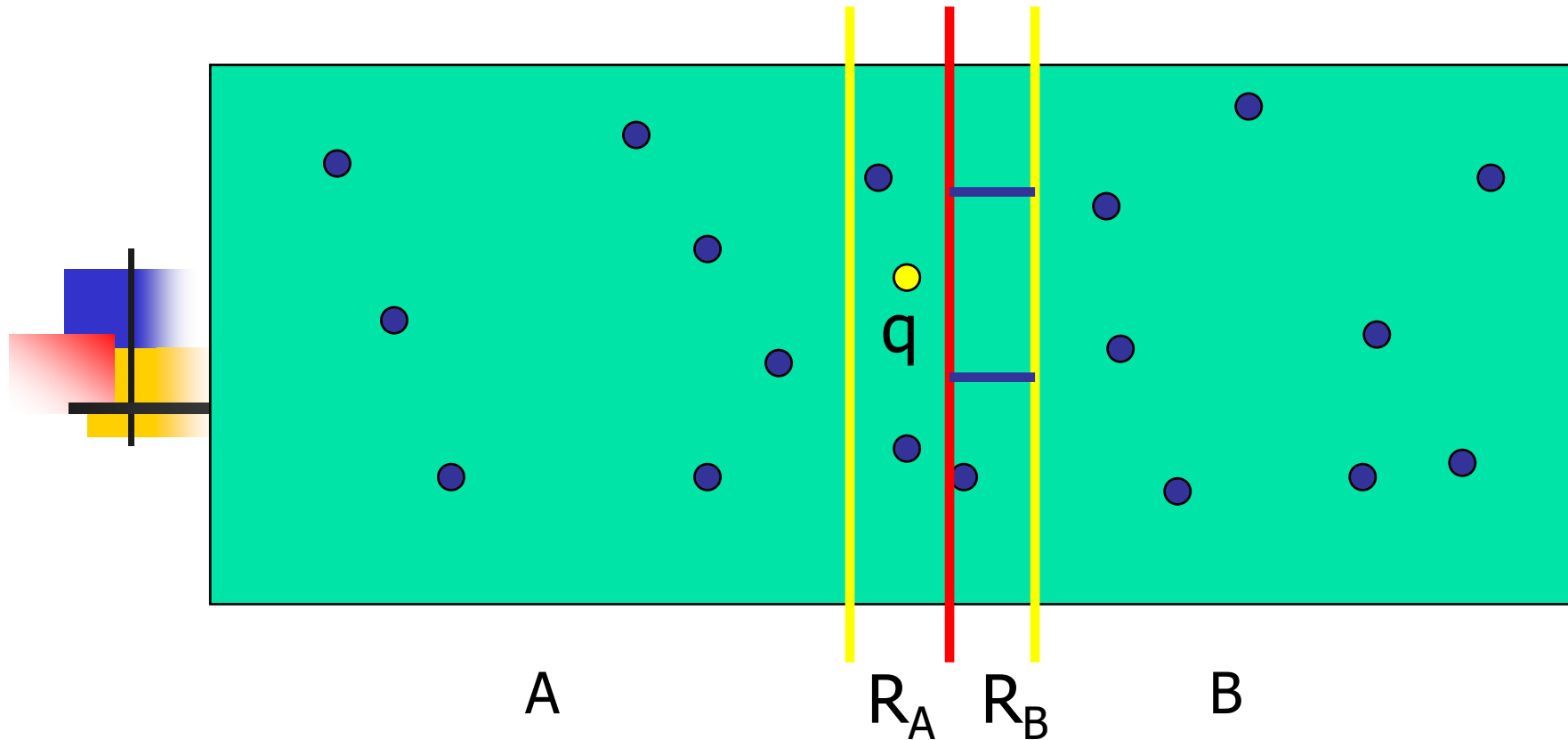


- Let  $d = \min\{d_1, d_2\}$ .
- Candidates lie within  $d$  of the dividing line.
- Call these regions  $R_A$  and  $R_B$ , respectively.

# Points in Slab

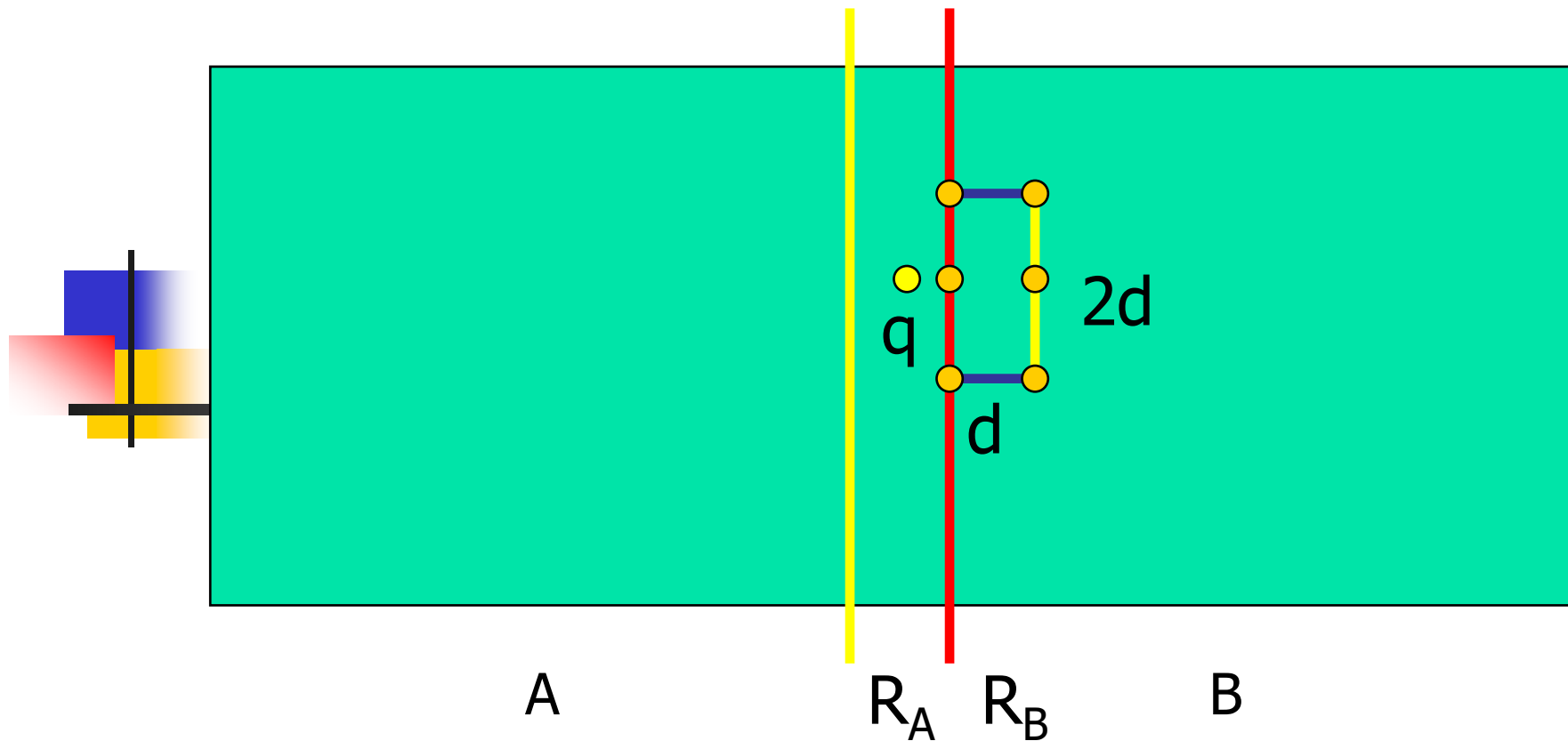
- During the merging step, we may examine only points in the slab.
- Although in average, the number of points within the slab may not be too large, in the **worst case**, there can be as many as  $n$  points within the slab.
- Thus the **brute-force way** to find the closest pair in the slab needs calculating  $n^2/4$  distances and comparisons.
- This kind of merging step will not be good for our divide-and-conquer algorithm.
- Fortunately, as will be shown in the following, the **merging step can be accomplished in  $O(n)$  time. (how?)**

# Example



- Let  $q$  be a point in  $R_A$ .
- $q$  need be paired only with those points in  $R_B$  that are within  $d$  of  $q.y$  (interval  $[q.y-d, q.y+d]$ ).
- Points that are to be paired with  $q$  are in a  $d \times 2d$  rectangle of  $R_B$  (comparing region of  $q$ ).

# Example



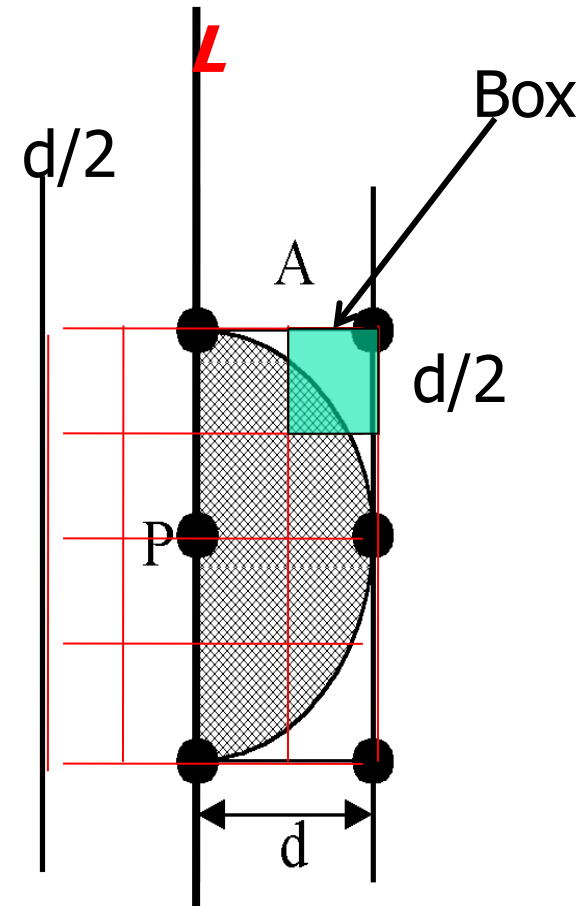
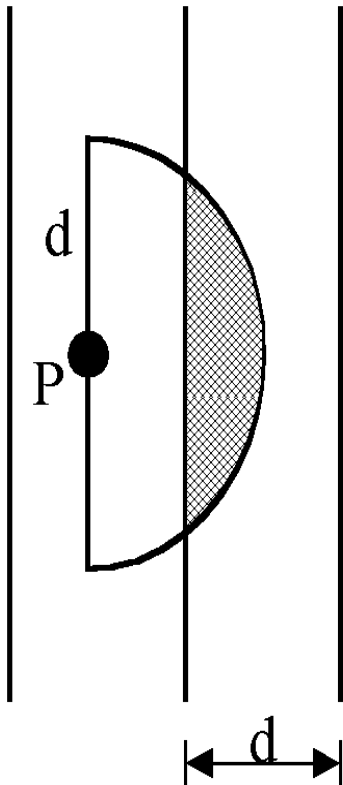
- So the comparing region of  $q$  has at most 8 points.
- So number of pairs to check is  $\leq 8 |R_A| = O(n)$ .

# Merge Step

- For each point  $P$  in the slab, we only have to examine limited number of points in the other half of the slab.
- Without losing generality, we may assume that  $P$  is within the left-half of the slab.
- Let the  $y$ -value of  $P$  be denoted as  $y_p$ . For  $P$ , we only have to examine points in the other half of the slab whose  $y$ -values are within  $y_p + d$  and  $y_p - d$ .
- There will be **at most eight** such points as discussed above **(why?)**.
- **$O(n)$**  Step

**Sort points by x-values and sort points by y-values.**

- at most 8 points in area A:



One box contains one point.

If  $s, s' \in S$  have the property that  $d(s, s') < d$ , then  $s$  and  $s'$  are within 15 positions of each other in the sorted list  $S_y$ .

The algorithm:

- Input: A set of  $n$  planar points.
- Output: The distance between two closest points.

Step 1: Sort points in  $S$  according to their  $y$ -values and  $x$ -values.

Step 2: If  $S$  contains only one point, return infinity( $\infty$ ) as their distance.

Step 3: Find a median line  $L$  perpendicular to the  $X$ -axis to divide  $S$  into two subsets, with equal sizes,  $S_L$  and  $S_R$ .

Step 4: Recursively apply Step 2 and Step 3 to solve the closest pair problems of  $S_L$  and  $S_R$ . Let  $d_L(d_R)$  denote the distance between the closest pair in  $S_L$  ( $S_R$ ). Let  $d = \min(d_L, d_R)$ .

Step 5: For a point P in the half-slab bounded by L-d and L, let its y-value be denoted as  $y_P$ . For each such P, find all points in the half-slab bounded by L and L+d whose y-value fall within  $y_P + d$  and  $y_P - d$ . If the distance  $d'$  between P and a point in the other half-slab is less than d, let  $d=d'$ . The final value of d is the answer.

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

Step 1:  $O(n \log n)$

Steps 2~5:

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

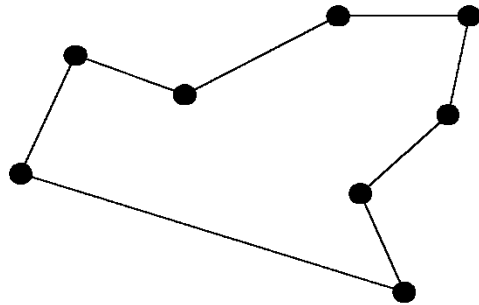
$$\Rightarrow T(n) = O(n \log n)$$



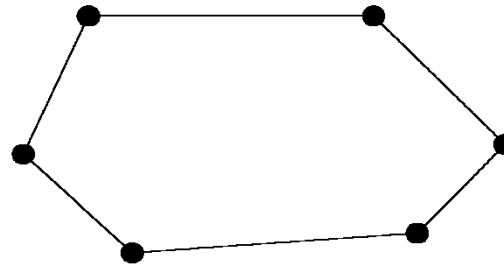
# The convex hull problem

# The convex hull problem

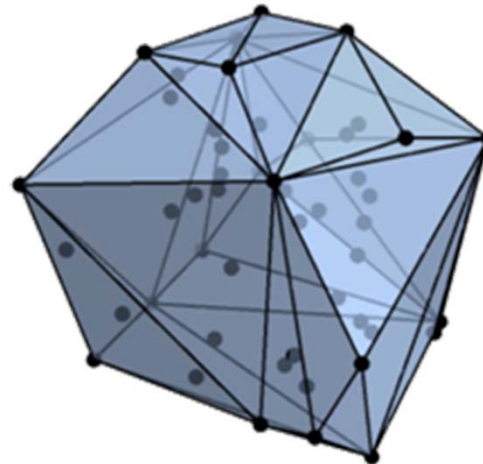
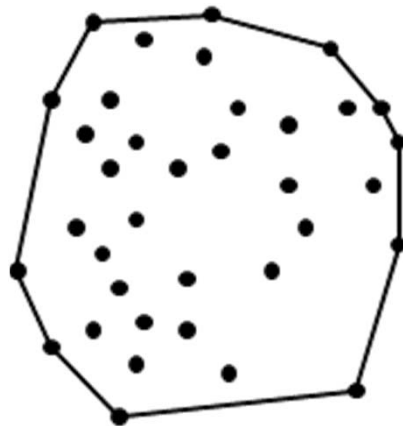
concave polygon:



convex polygon:

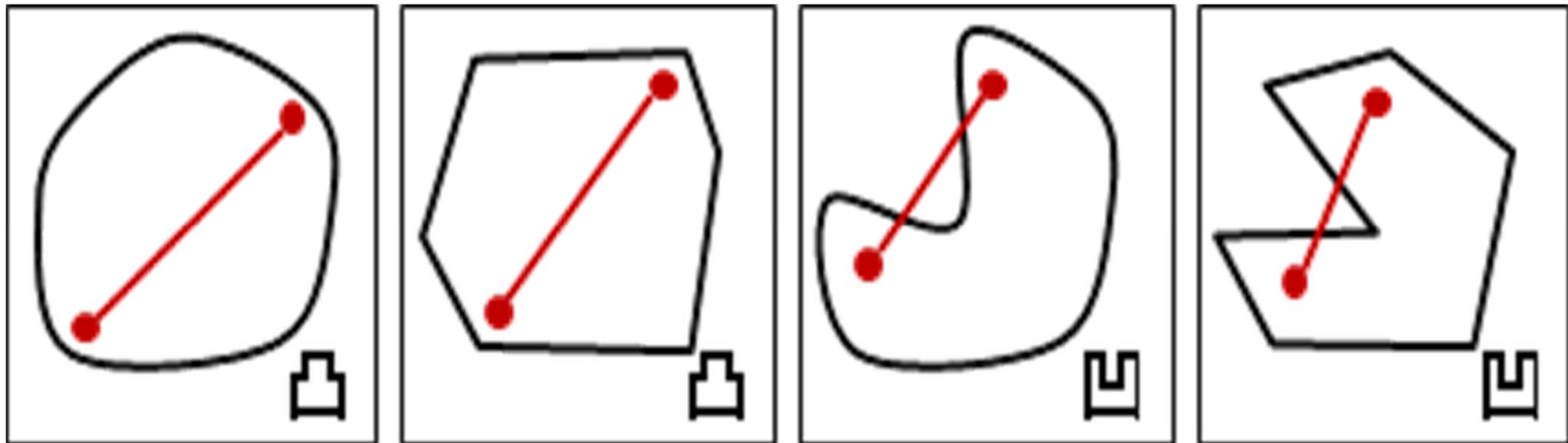


- The convex hull of a set of planar points is the smallest convex polygon containing all of the points.



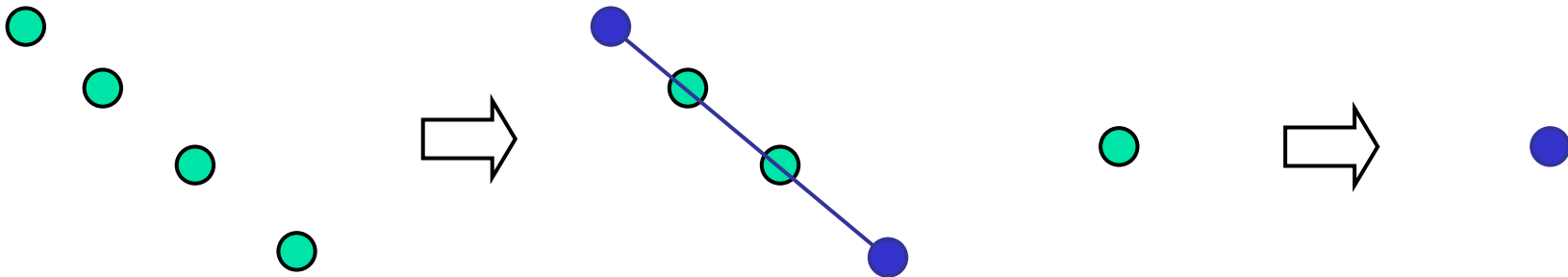
# Convex Polygon

- A convex polygon is a nonintersecting polygon whose **internal angles are all convex** (i.e., less than  $\pi$ )
- In a convex polygon, a segment joining two vertices of the polygon **lies entirely inside the polygon**.
- The convex hull of a set of points is the smallest convex polygon containing the points



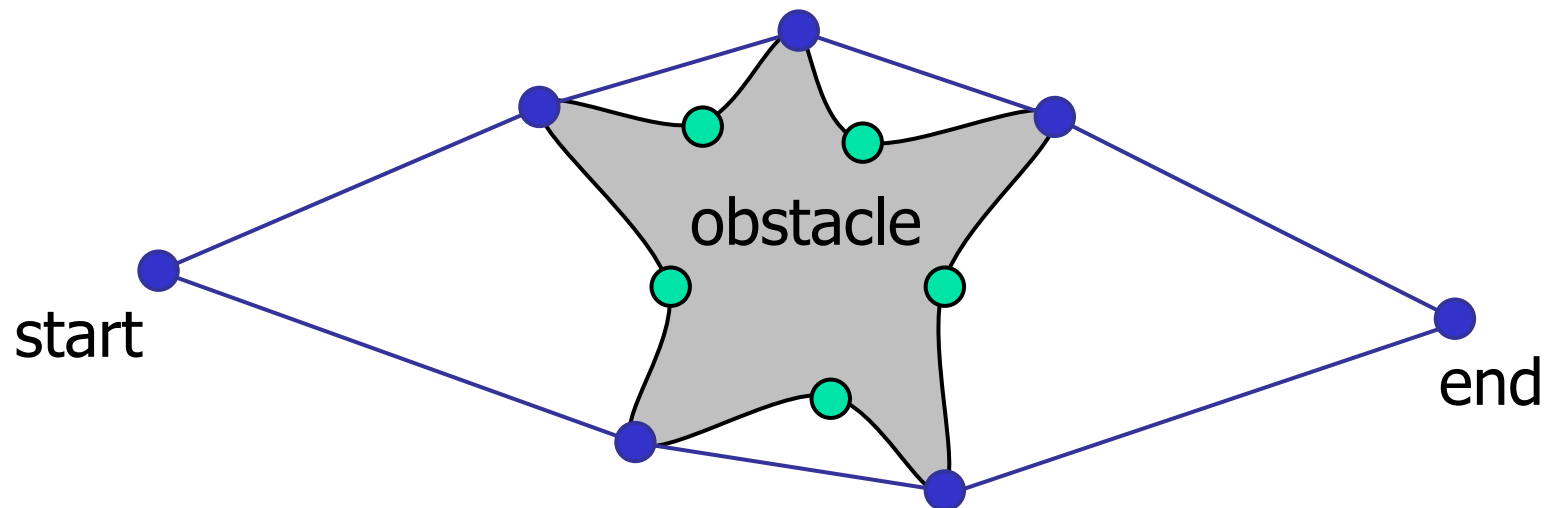
# Special Cases

- The convex hull is a segment
  - Two points
  - All the points are collinear
- The convex hull is a point
  - there is one point
  - All the points are coincident



# Applications

- Motion planning
  - Find an optimal route that avoids obstacles for a robot
- Geometric algorithms
  - Convex hull is like a two-dimensional sorting

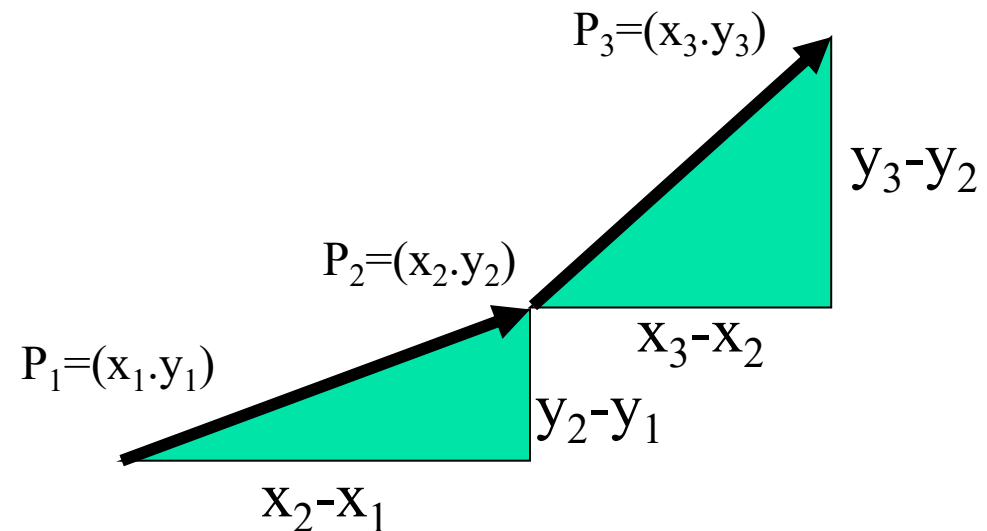


- The **orientation** of a triplet  $(p_1, p_2, p_3)$  of points in the plane is counterclockwise, clockwise, or collinear, depending on whether,  $\Delta(p_1, p_2, p_3)$  is positive, negative, or zero, respectively.
- Assume  $p_1=(x_1, y_1)$ ,  $p_2=(x_2, y_2)$ , and  $p_3=(x_3, y_3)$  and  $x_1 < x_2 < x_3$

Proof)

$$\begin{aligned}
 \text{area of } \Delta &= 1/2 \Delta(p_1, p_2, p_3) \\
 &= 1/2 \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \\
 &= x_1 y_2 + x_2 y_3 + x_3 y_1 \\
 &\quad - (x_3 y_2 + x_1 y_3 + x_2 y_1) = 0
 \end{aligned}$$

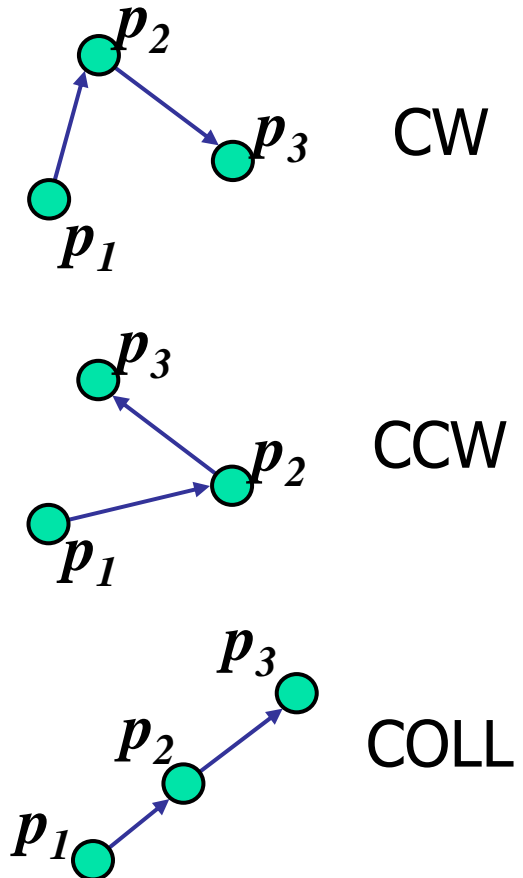
**collinear**



Left turn if  $\frac{y_2 - y_1}{x_2 - x_1} < \frac{y_3 - y_2}{x_3 - x_2}$

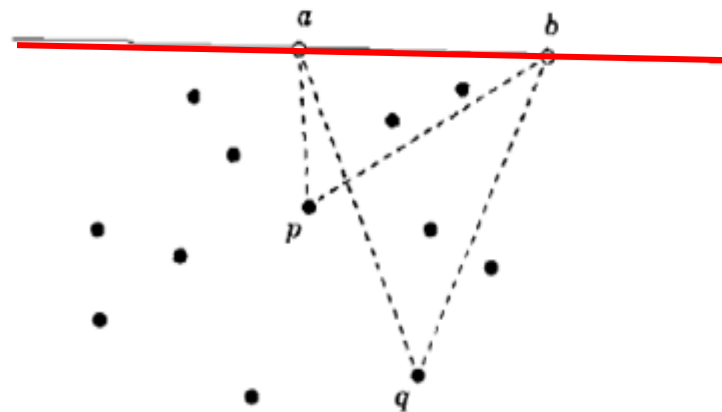
# Orientation

- The orientation of three points in the plane is clockwise, counterclockwise, or collinear
- **orientation**( $p_1, p_2, p_3$ )
  - clockwise (CW, right turn)
  - counterclockwise (CCW, left turn)
  - collinear (COLL, no turn)
- The orientation of three points is characterized by the **sign** of the determinant  $D(p_1, p_2, p_3)$ , whose absolute value is **twice** the area of the triangle with vertices  $p_1, p_2$  and  $p_3$

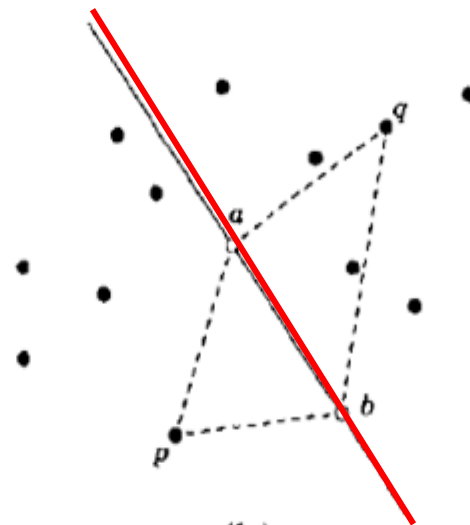


**Theorem** : Let  $S$  be a set of planar points with convex hull  $H$ . Then

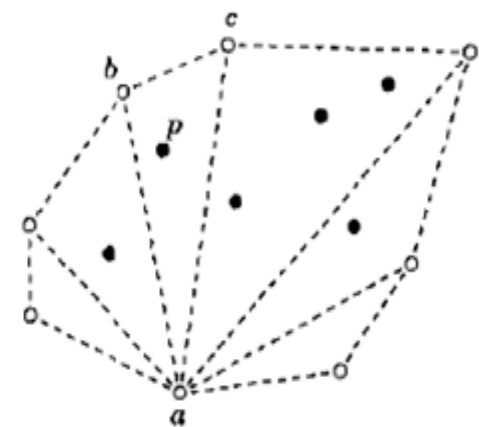
- A pair of points  $a$  and  $b$  of  $S$  form an edge of  $H$  if and only if all the other points of  $S$  are contained on one side of the line through  $a$  and  $b$ .
- A point  $p$  of  $S$  is a vertex of  $H$  if and only if there exists a line  $l$  through  $p$ , such that all the other points of  $S$  are contained in the same half-plane delimited by  $l$  (that is, they are all on the same side of  $l$ ).
- A point  $p$  of  $S$  is not a vertex of  $H$  if and only if  $p$  is contained in the interior of a triangle formed by three other points of  $S$  or in the interior of a segment formed by two other points of  $S$ .



(a)



(b)

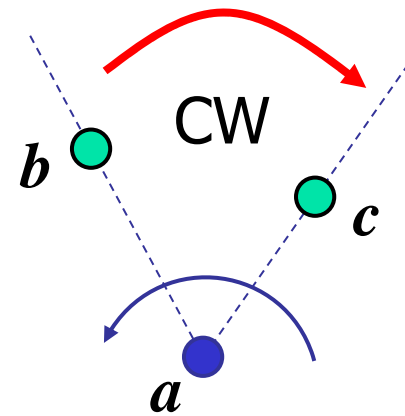
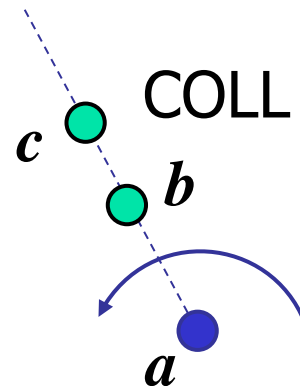
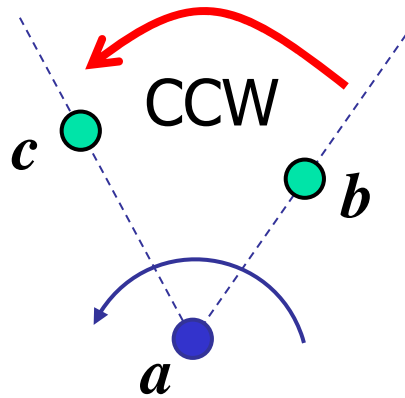


(c)



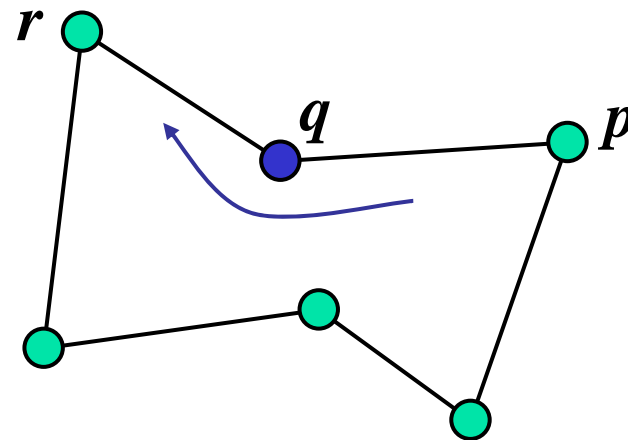
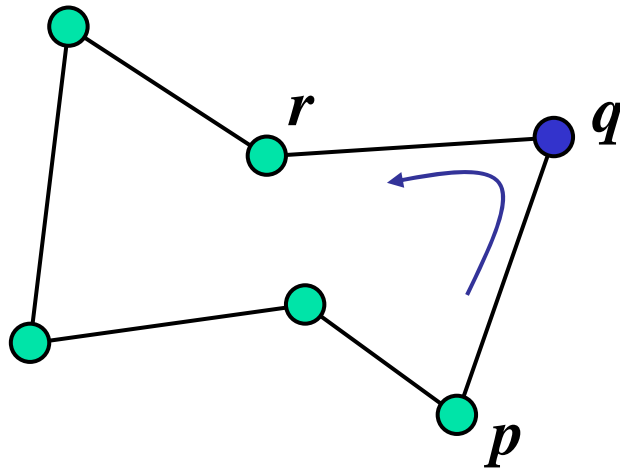
# Sorting by Angle

- Computing angles from coordinates is complex and leads to numerical inaccuracy
- We can **sort a set of points by angle** with respect to the anchor point  $a$  using a comparator based on the orientation function
  - $b < c \Leftrightarrow \text{orientation}(a, b, c) = \text{CCW}$
  - $b = c \Leftrightarrow \text{orientation}(a, b, c) = \text{COLL}$
  - $b > c \Leftrightarrow \text{orientation}(a, b, c) = \text{CW}$



# Removing Nonconvex Vertices

- Testing whether a vertex is convex can be done using the orientation function
- Let  $p$ ,  $q$  and  $r$  be three consecutive vertices of a polygon, in counterclockwise order
  - $q$  convex  $\Leftrightarrow \text{orientation}(p, q, r) = \text{CCW}$
  - $q$  nonconvex  $\Leftrightarrow \text{orientation}(p, q, r) = \text{CW}$  or  $\text{COLL}$

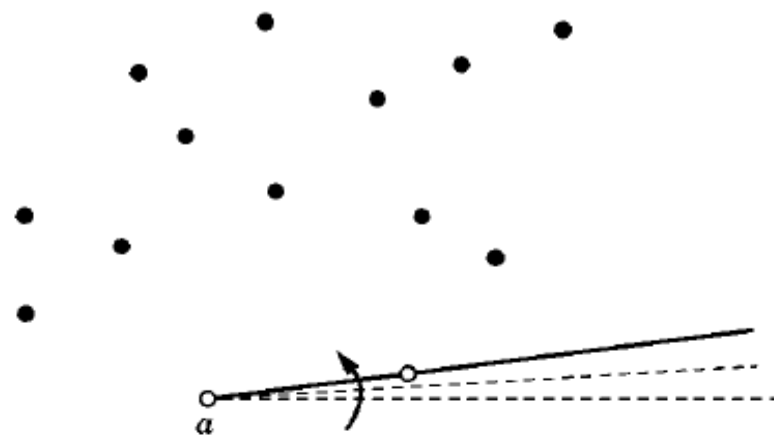


# Gift Wrapping Algorithm

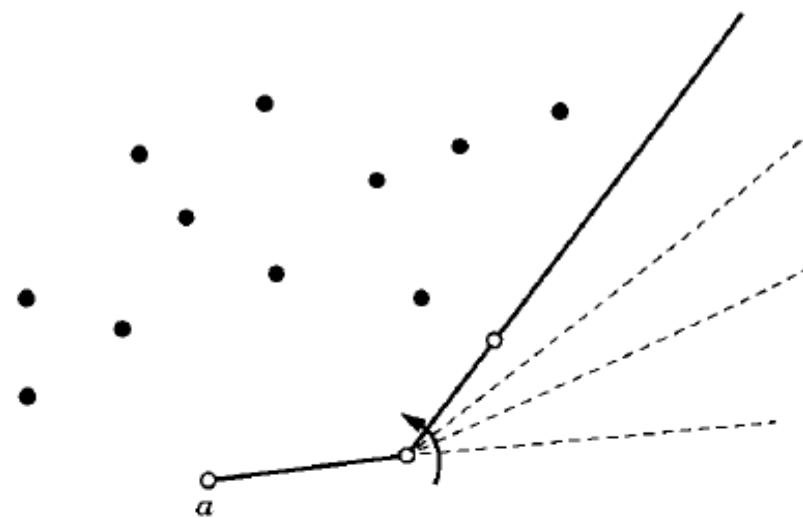
- We can identify a particular point, say one with minimum y-coordinate, that provides an initial starting configuration for an algorithm that computes the convex hull.
- The **gift wrapping algorithm** for computing the convex hull of a set of points in the plane is based on just such a starting point.

# Gift Wrapping

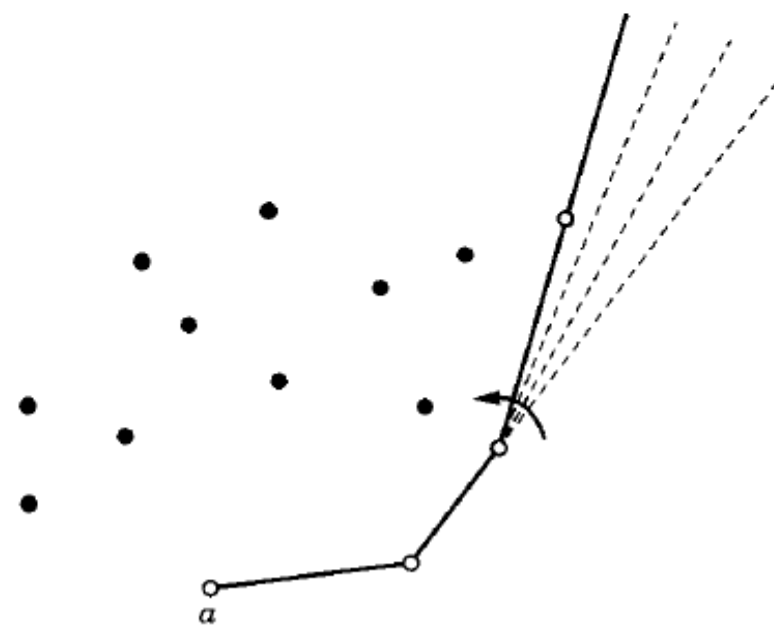
- View the points as pegs implanted in a level field, and imagine that we tie a rope to the peg corresponding to the point  $a$  with minimum y-coordinate (and minimum x-coordinate if there are ties). Call  $a$  the **anchor point**, and note that  $a$  is a vertex of the convex hull.
- Pull the rope to the right of the anchor point and rotate it counterclockwise until it touches another peg, which corresponds to the next vertex of the convex hull.
- Continue rotating the rope counterclockwise, identifying a new vertex of the convex hull at each step, until the rope gets back to the anchor point.



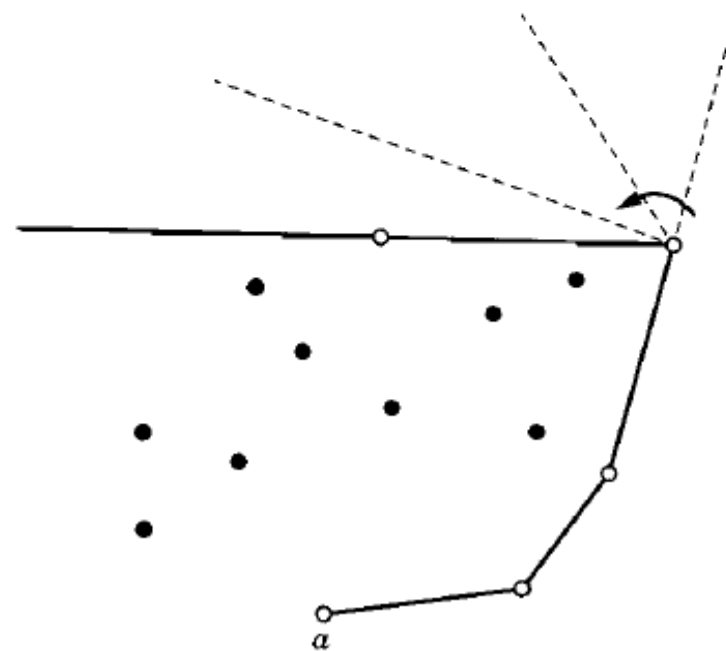
(a)



(b)



(c)



(d)

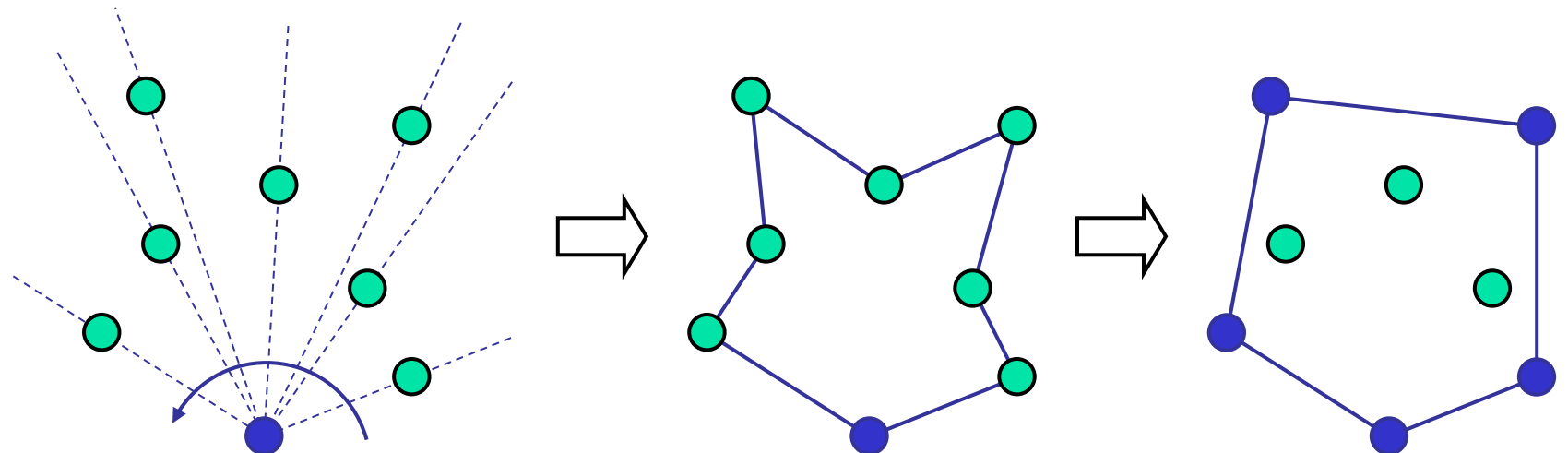
# Graham's Scan Algorithm

- The following method computes the convex hull of a set of points

**Phase 1:** Find the lowest point (anchor point) on the hull. (the point with the minimum y value)

**Phase 2:** Form a nonintersecting polygon by sorting the points counterclockwise around the anchor point

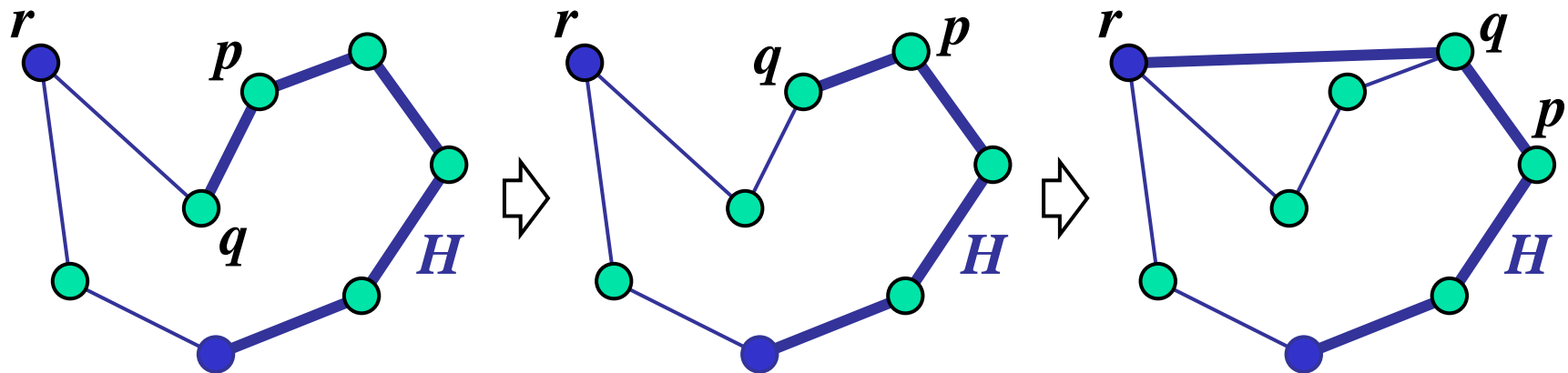
**Phase 3:** While the polygon has a nonconvex vertex, remove it

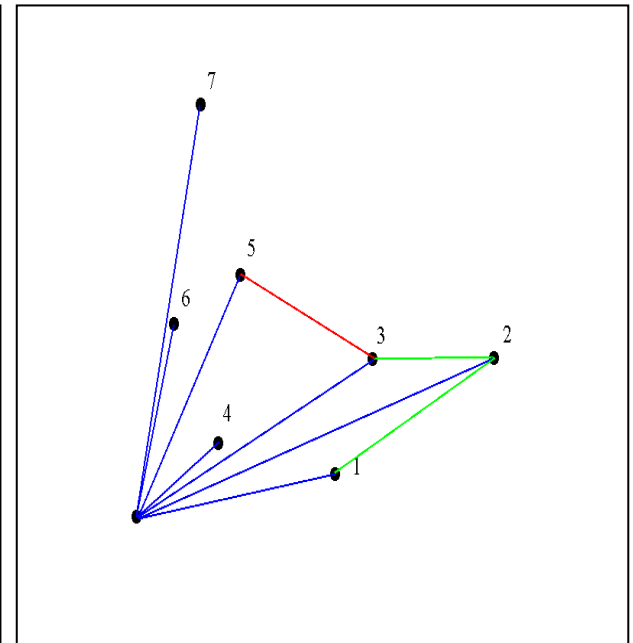
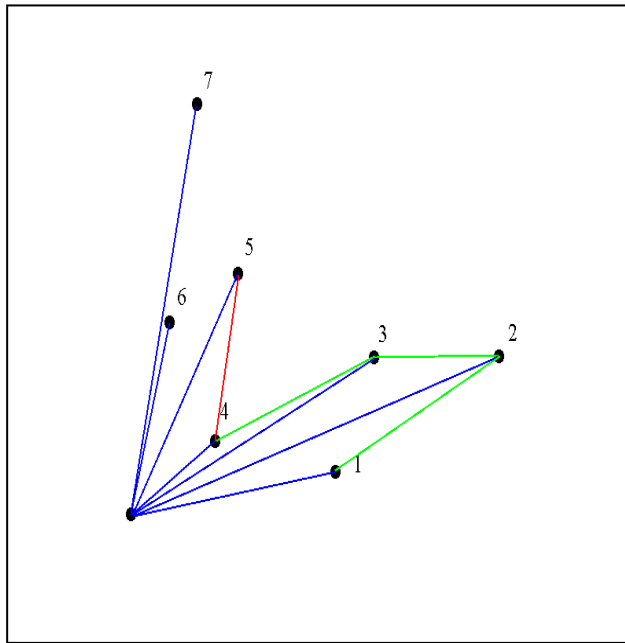
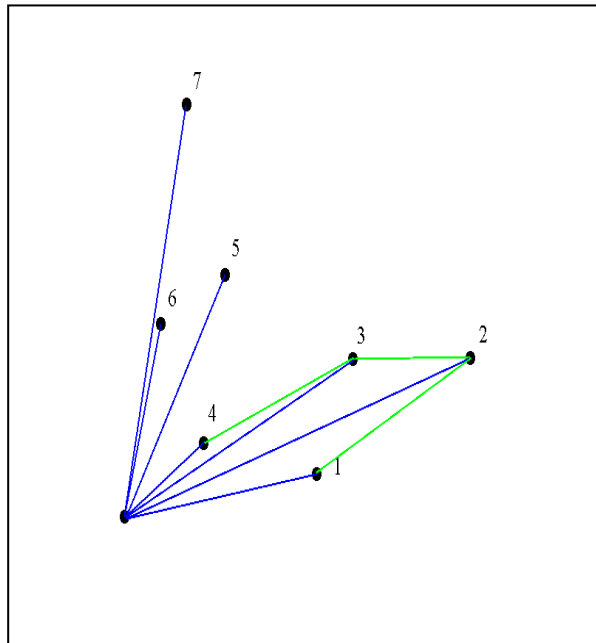
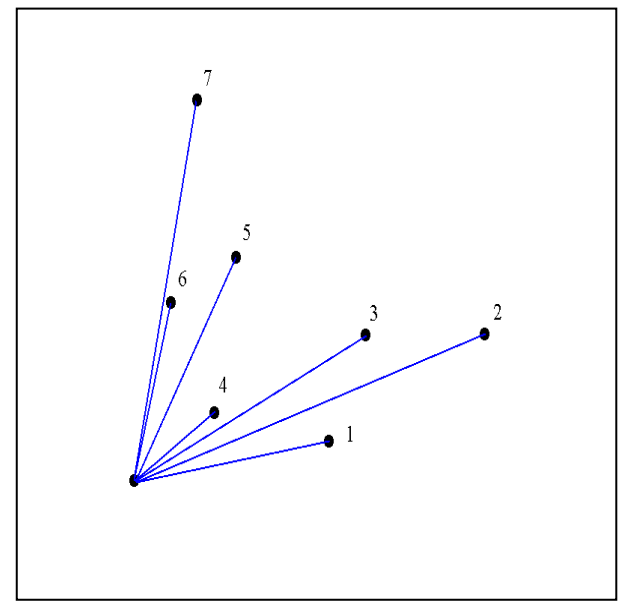
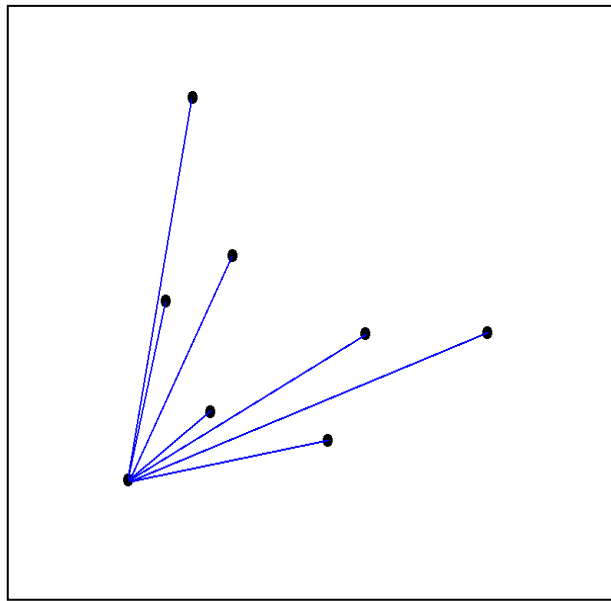


# Graham Scan

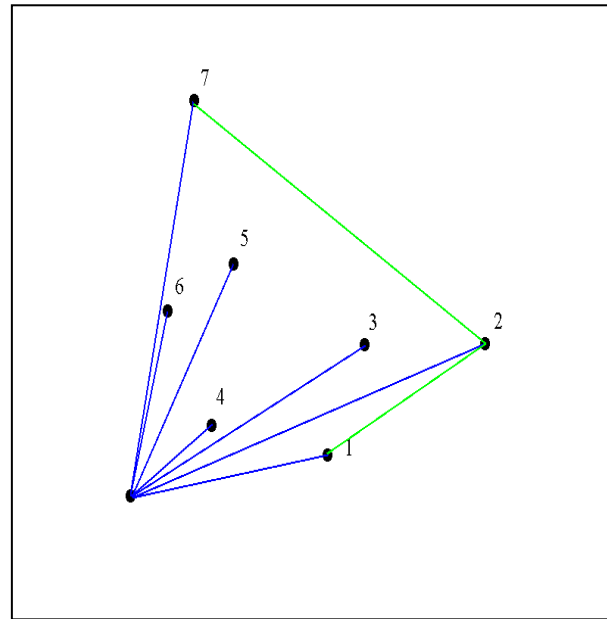
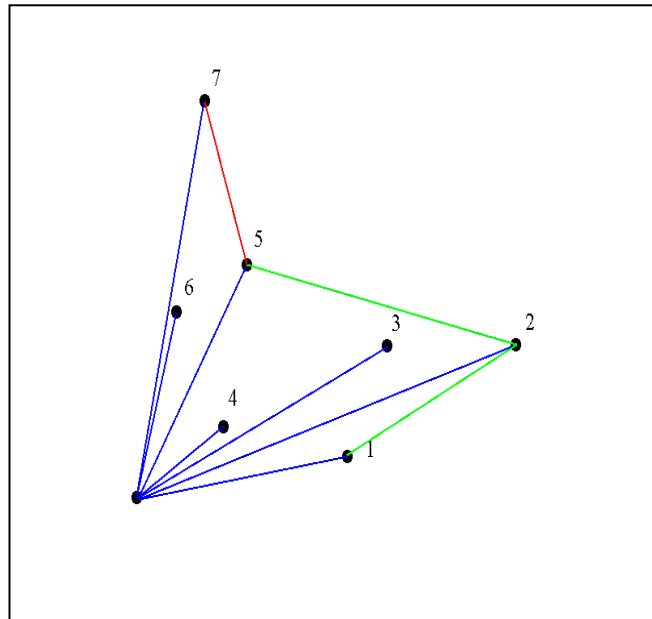
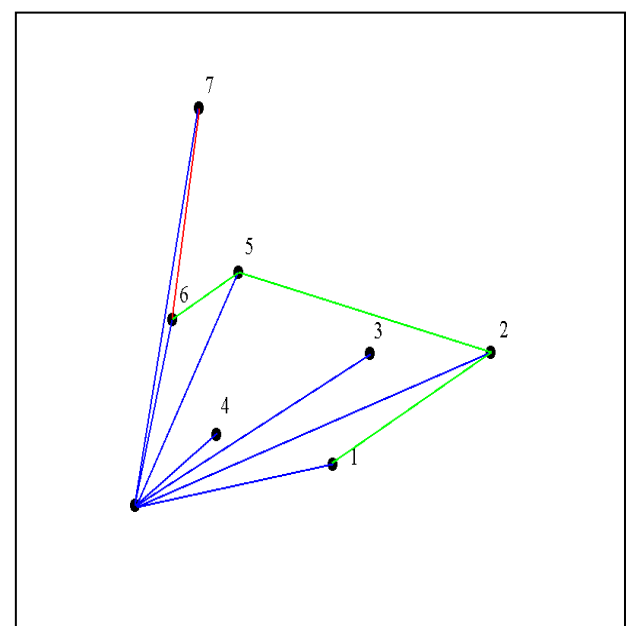
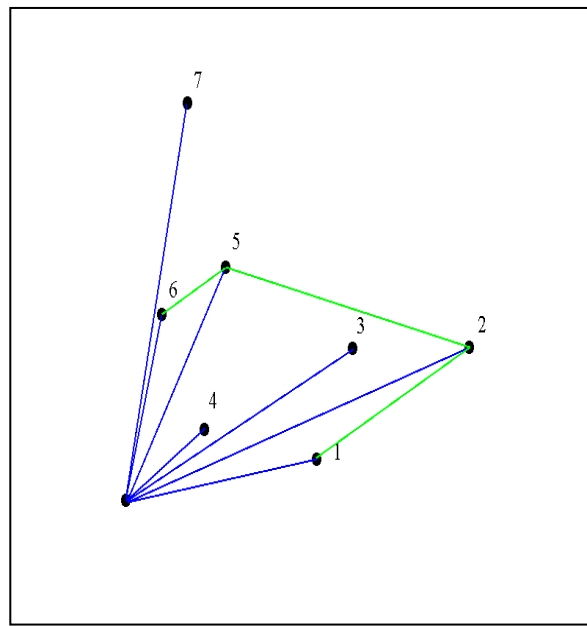
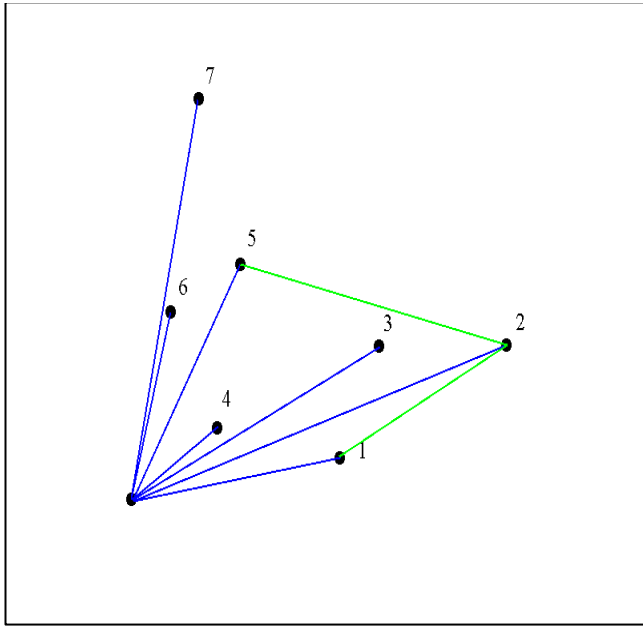
- The Graham scan is a systematic procedure for removing nonconvex vertices from a polygon
- The polygon is traversed counterclockwise and a sequence  $H$  of vertices is maintained

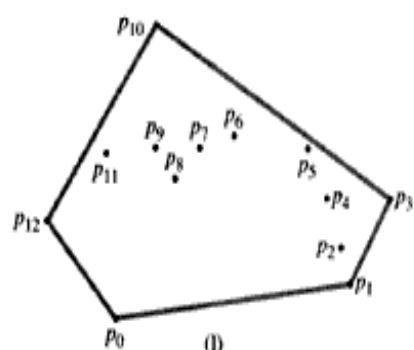
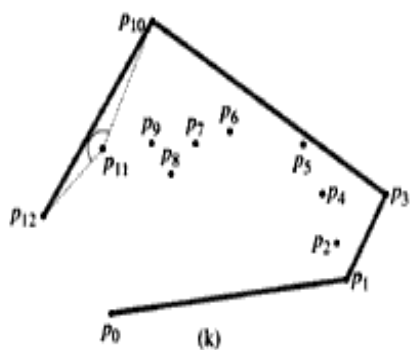
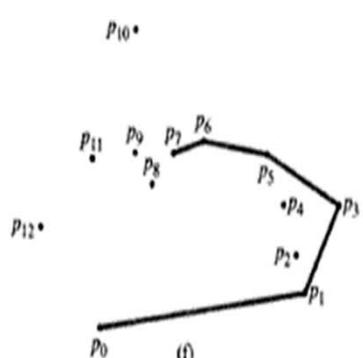
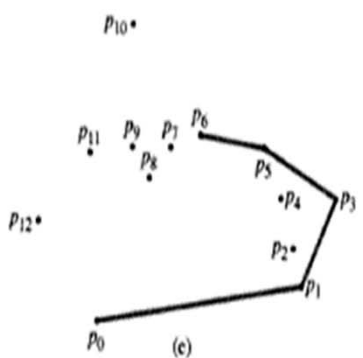
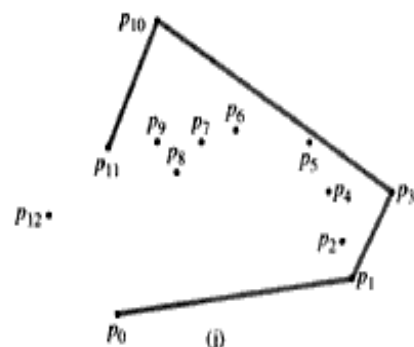
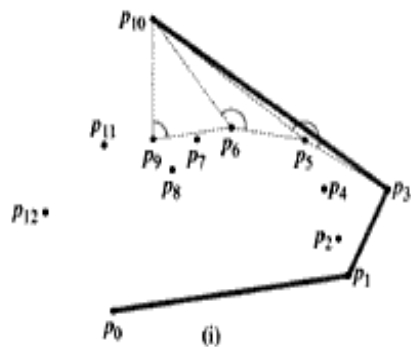
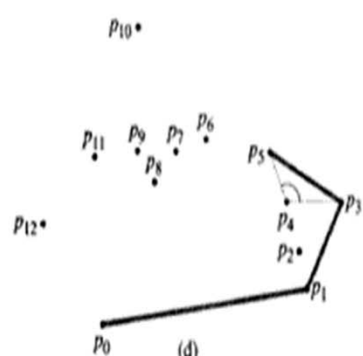
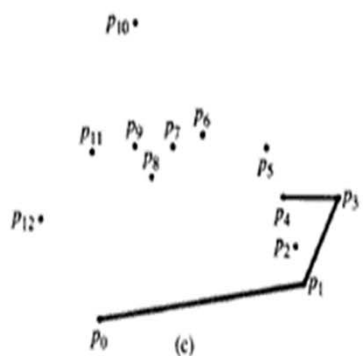
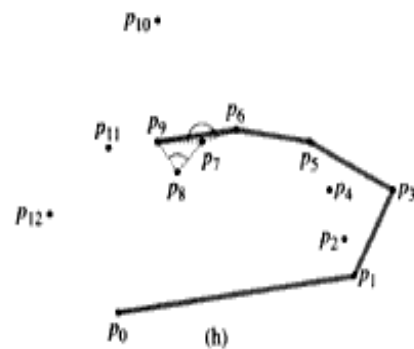
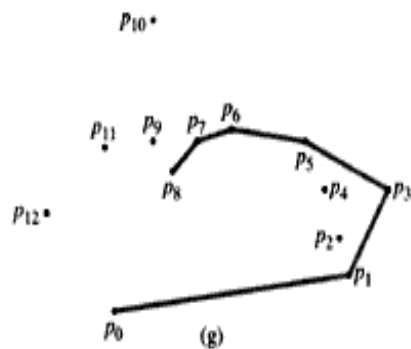
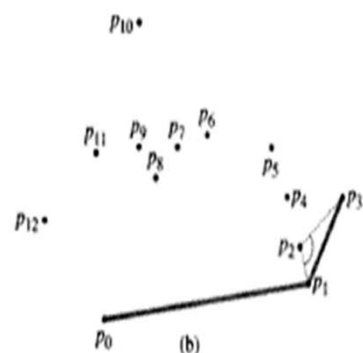
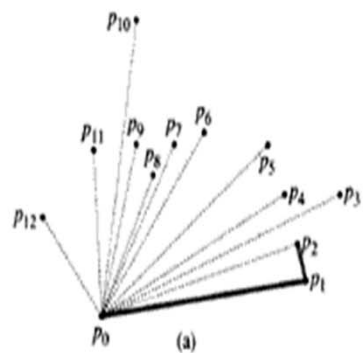
```
for each vertex  $r$  of the polygon
  Let  $q$  and  $p$  be the last and second last
  vertex of  $H$ 
  while  $\text{orientation}(p, q, r) = \text{CW or COLL}$ 
    remove  $q$  from  $H$ 
     $q \leftarrow p$ 
     $p \leftarrow$  vertex preceding  $p$  in  $H$ 
  Add  $r$  to the end of  $H$ 
```







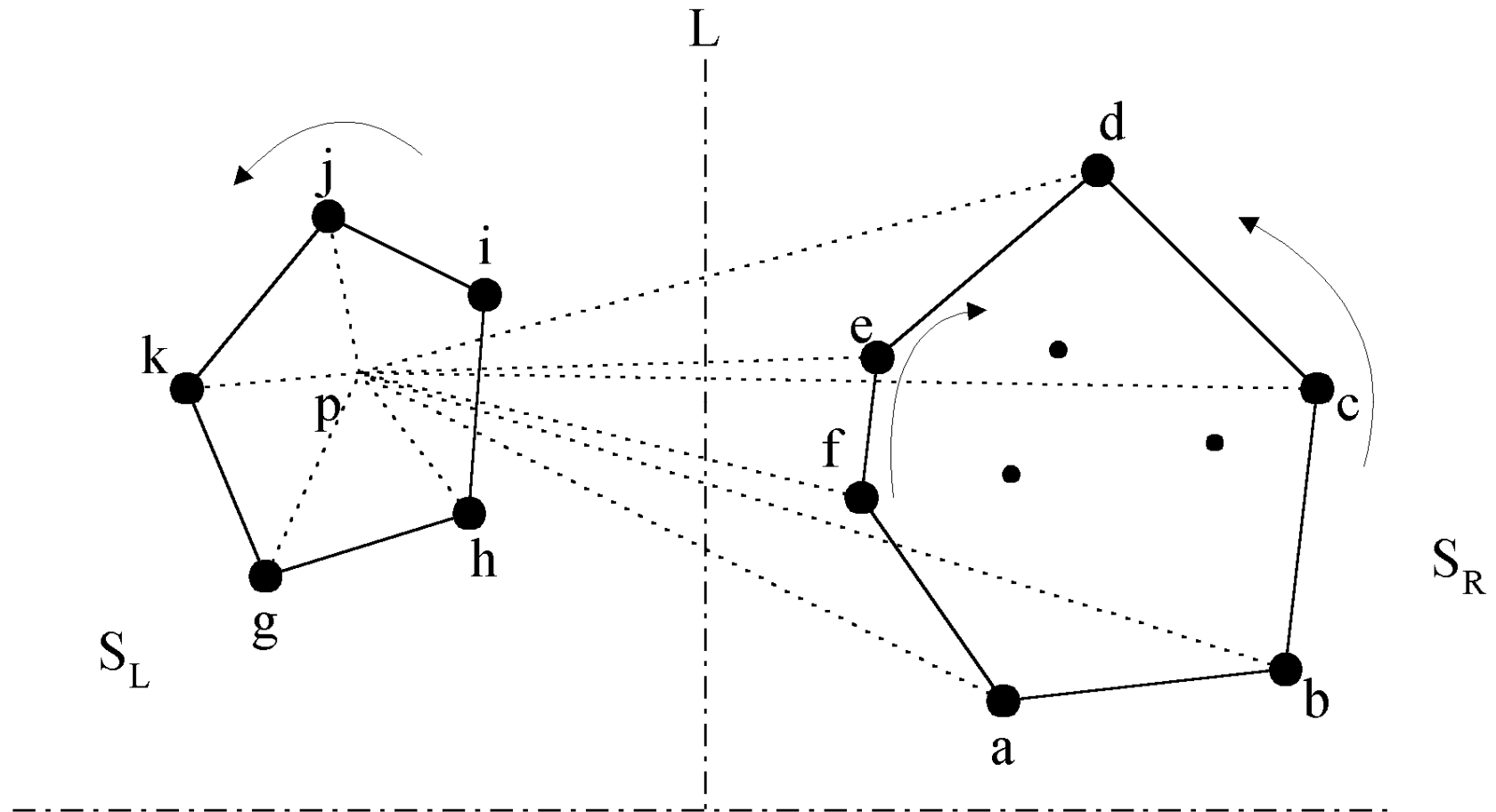




# Analysis

- Computing the convex hull of a set of points takes  $O(n \log n)$  time
  - Finding the anchor point takes  $O(n)$  time
  - Sorting the points counterclockwise around the anchor point takes  $O(n \log n)$  time
    - Use the orientation comparator and any sorting algorithm that runs in  $O(n \log n)$  time (e.g., heap-sort or merge-sort)
  - The Graham scan takes  $O(n)$  time
    - Each point is inserted once in sequence  $H$
    - Each vertex is removed at most once from sequence  $H$

# Divide-and-Conquer strategy

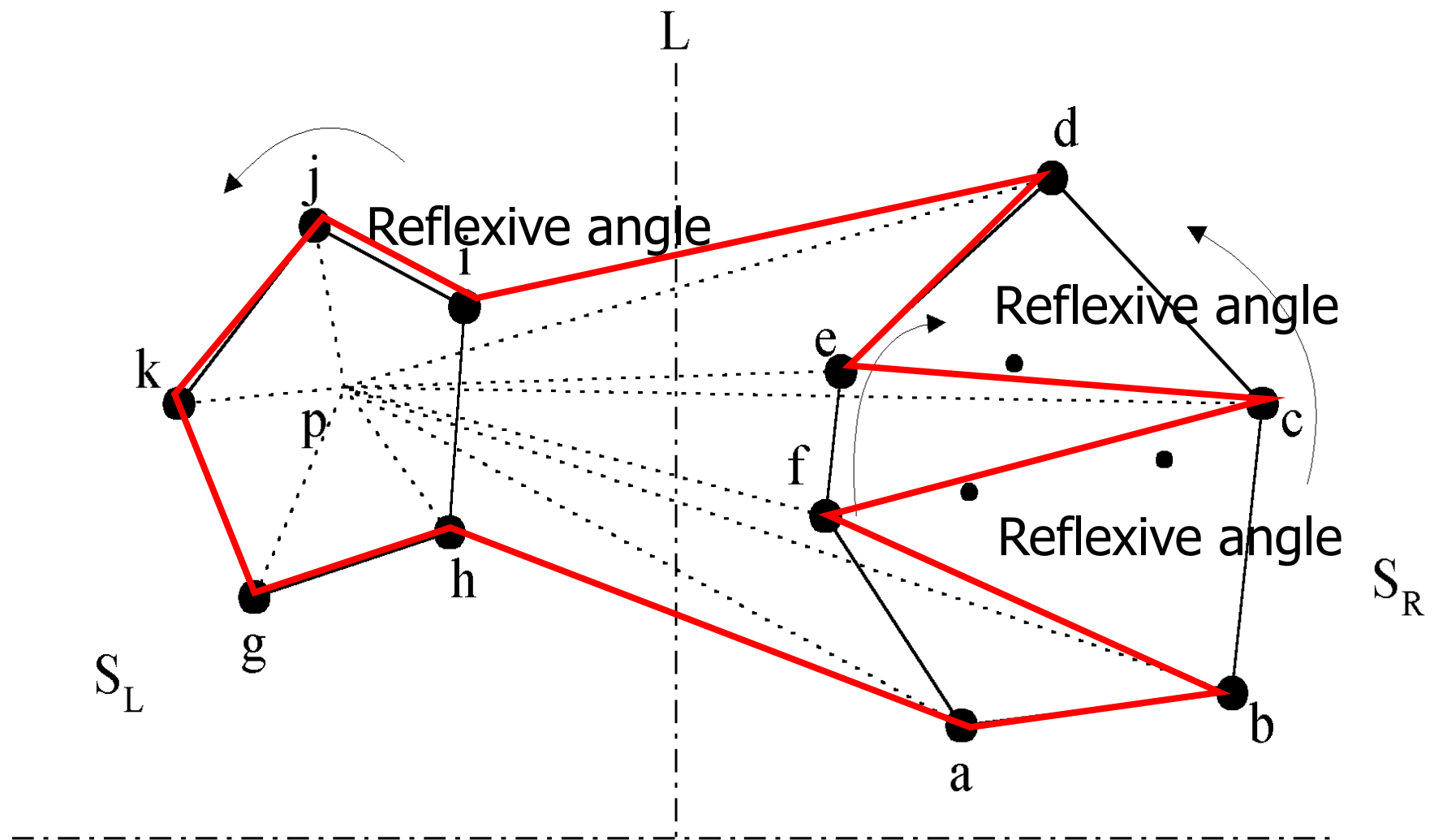


# Convex Hull problem

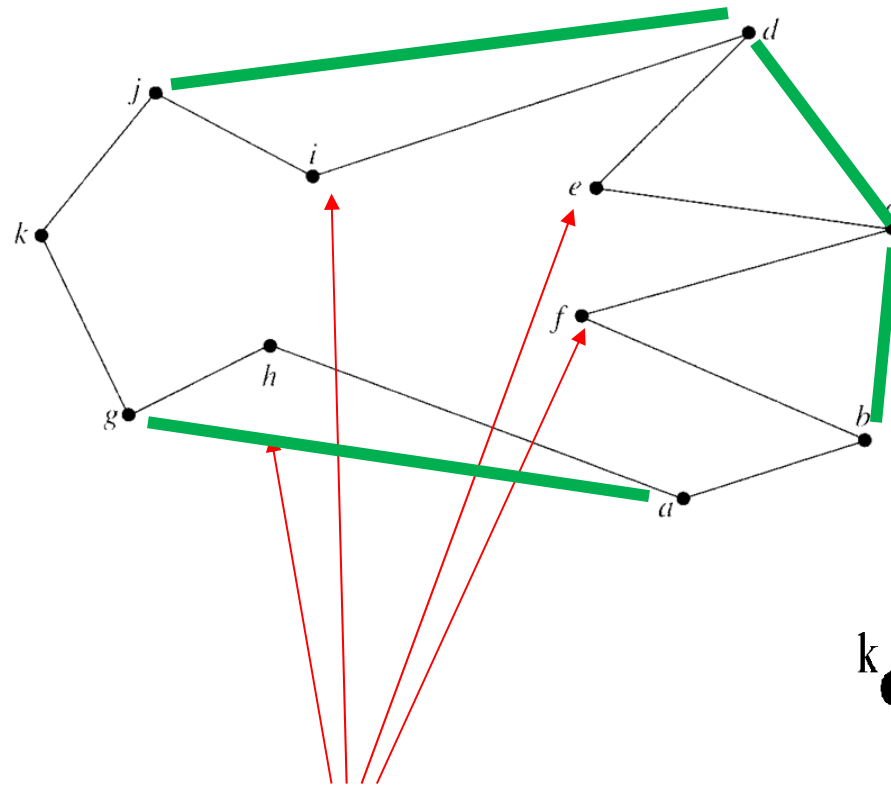
- To find a convex hull, we may use the divide-and-conquer.
- The set of planar points is **divided** into two subsets  $S_L$  and  $S_R$  by a line perpendicular to the x-axis.
- Convex hulls for  $S_L$  and  $S_R$  are now constructed and they are denoted as **Hull( $S_L$ )**, **Hull( $S_R$ )** respectively.
- To **combine** Hull( $S_L$ ) and Hull( $S_R$ ) into one convex hull use the **Graham scan**.

# Graham scan

- An **interior point** of **Hull( $S_L$ )** is selected.
- Consider the point as the origin.
- Then each other point forms a **polar angle** with interior point.
- All of the points are now sorted with respect to these **polar angle**.
- The Graham scan examines the points one by one and eliminates the points which cause **reflexive angles**.

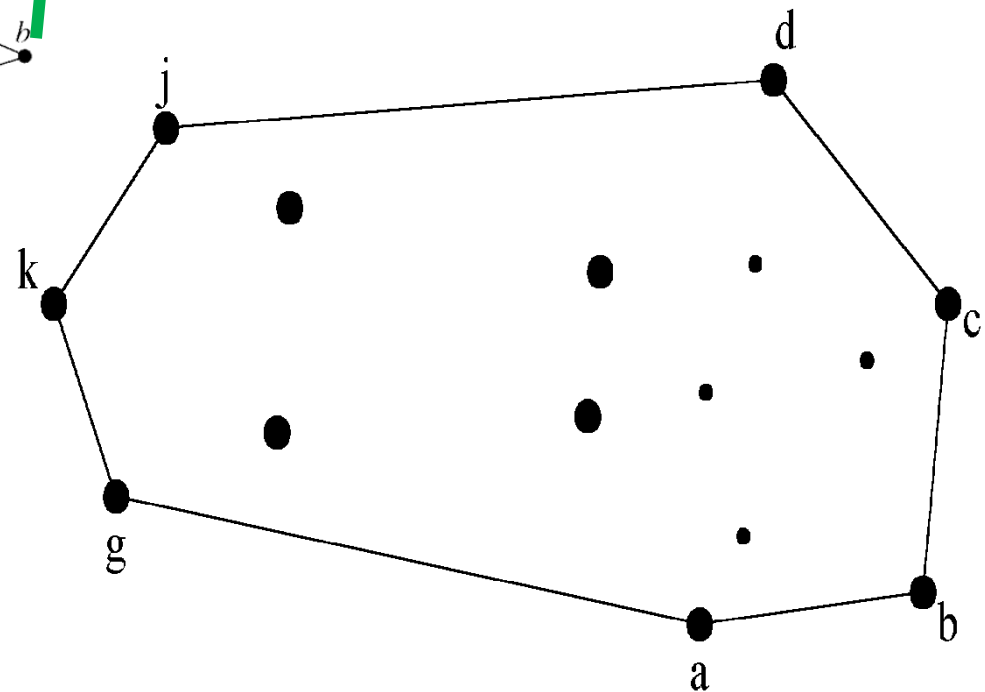


- e.g. points e, h, f and i need to be deleted.



reflexive angles

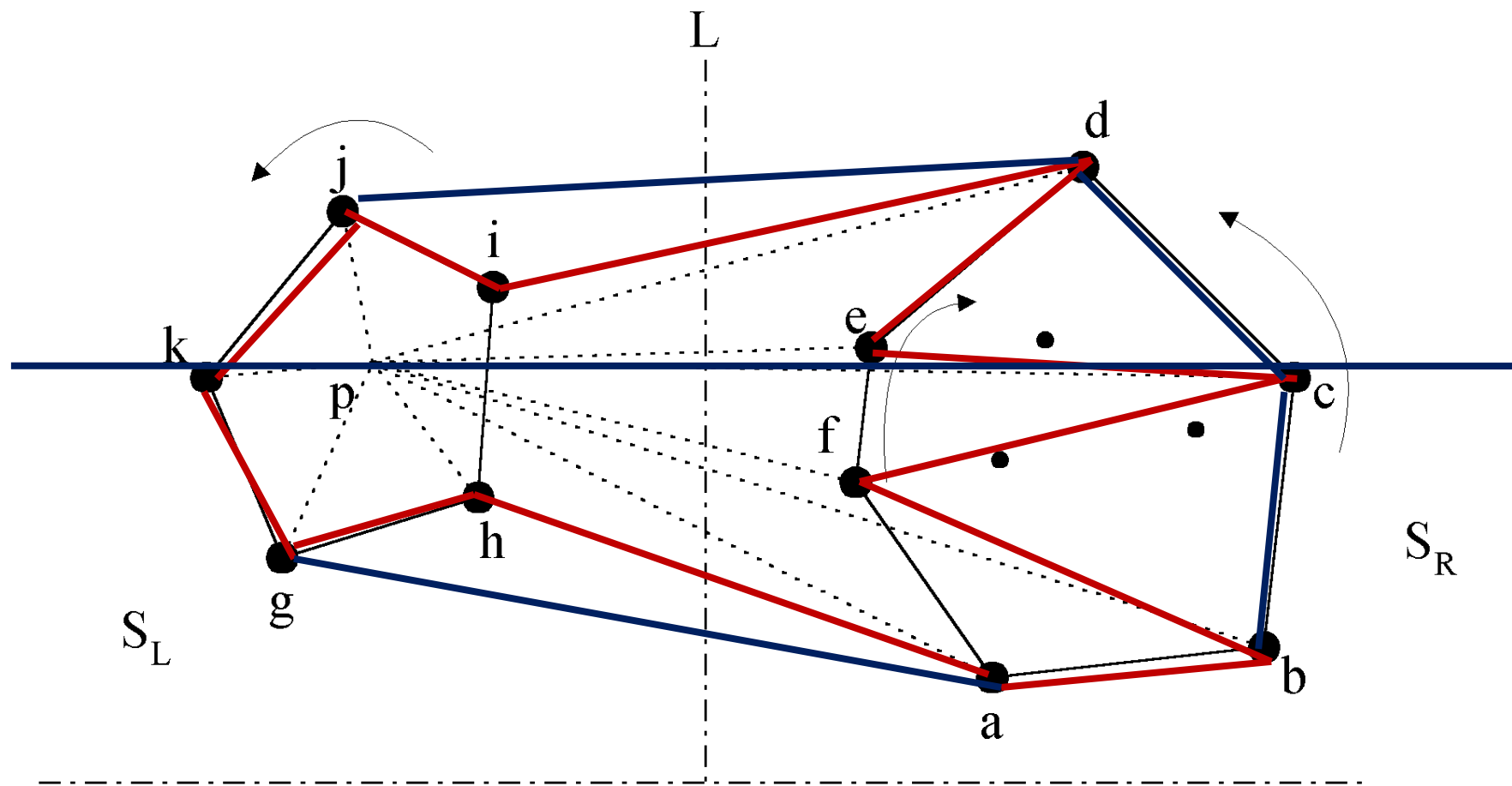
Final result:





- The merging procedure:
  1. Select an interior point  $p$ .
  2. There are 3 sequences of points which have increasing polar angles with respect to  $p$ .
    - (1)  $g, h, i, j, k$
    - (2)  $a, b, c, d$
    - (3)  $f, e$
  3. Merge these 3 sequences into 1 sequence:  
 $g, h, a, b, f, c, e, d, i, j, k$ .
  4. Apply Graham scan to examine the points one by one and eliminate the points which cause reflexive angles.

- The divide-and-conquer strategy to solve the problem:



# Divide-and-conquer for convex hull

- Input : A set  $S$  of planar points

- Output : A convex hull for  $S$

Step 1: If  $S$  contains no more than five points, use exhaustive searching to find the convex hull and return.

Step 2: Find a median line perpendicular to the  $X$ -axis which divides  $S$  into  $S_L$  and  $S_R$  ;  $S_L$  lies to the left of  $S_R$  .

Step 3: Recursively construct convex hulls for  $S_L$  and  $S_R$ .  
Denote these convex hulls by  $Hull(S_L)$  and  $Hull(S_R)$  respectively.

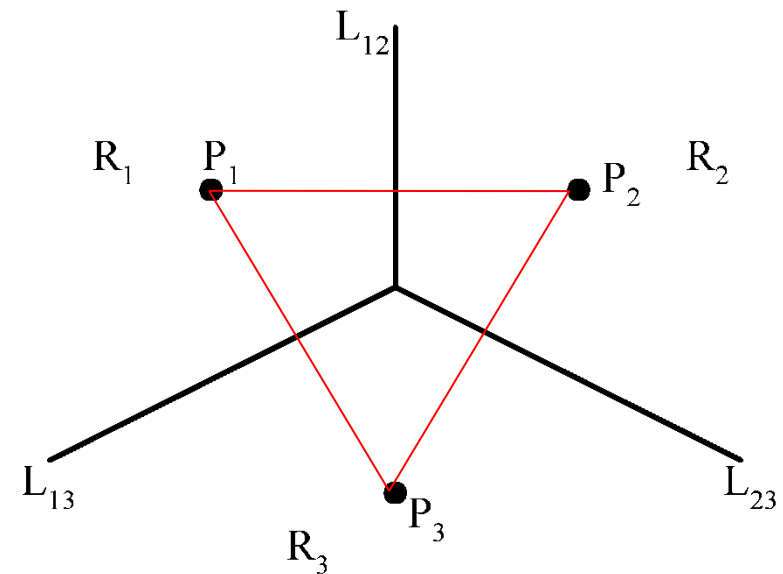
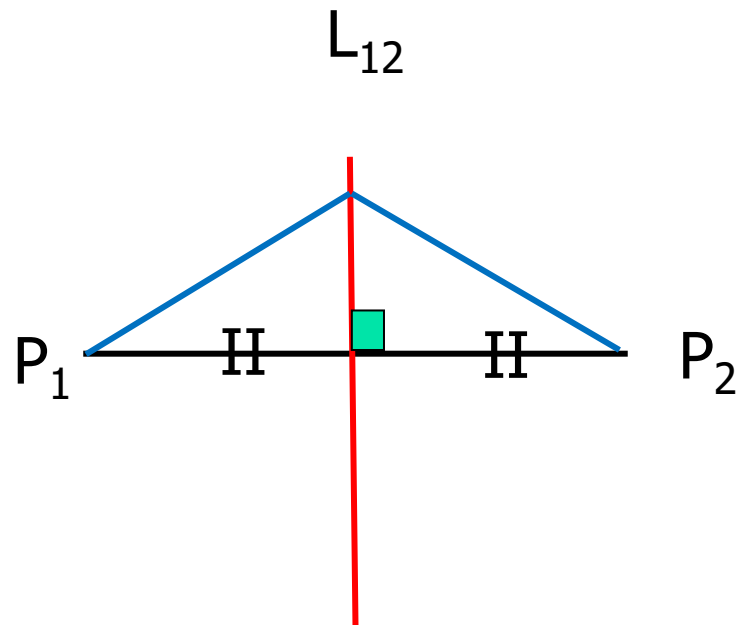
Step 4: Apply the merging procedure to merge  $Hull(S_L)$  and  $Hull(S_R)$  together to form a convex hull.

- Time complexity:  $T(n) = 2T(n/2) + O(n) = O(n \log n)$

# The Voronoi diagram problem

# The Voronoi diagram problem

The Voronoi diagram for two & three points

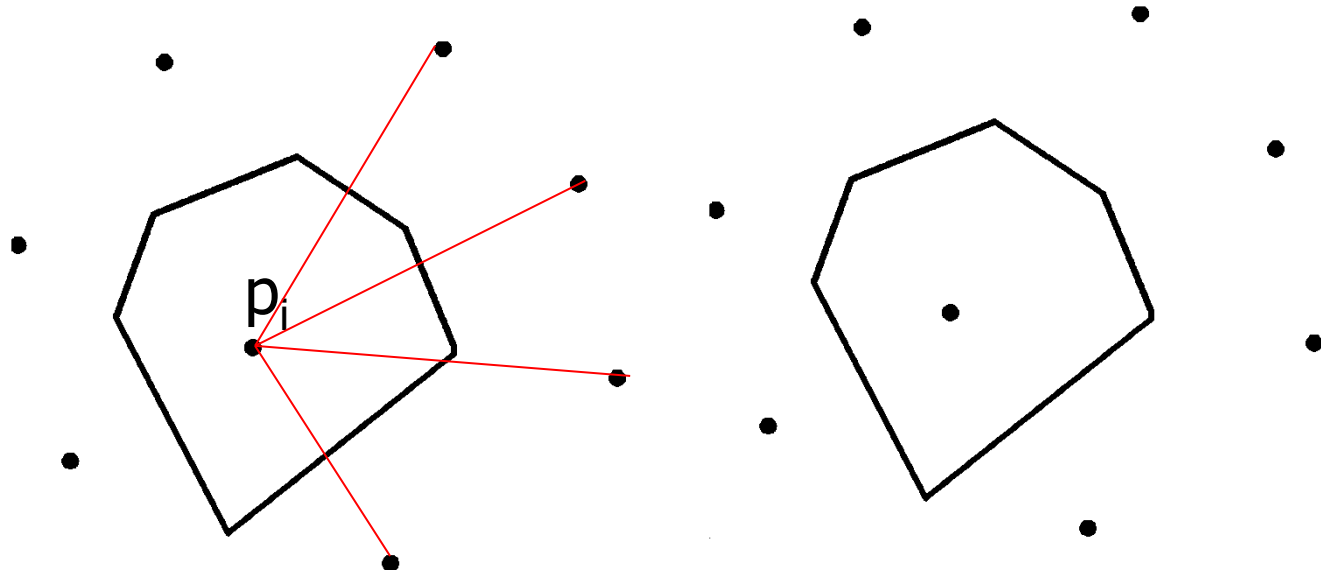


Each  $L_{ij}$  is the perpendicular bisector of the line.

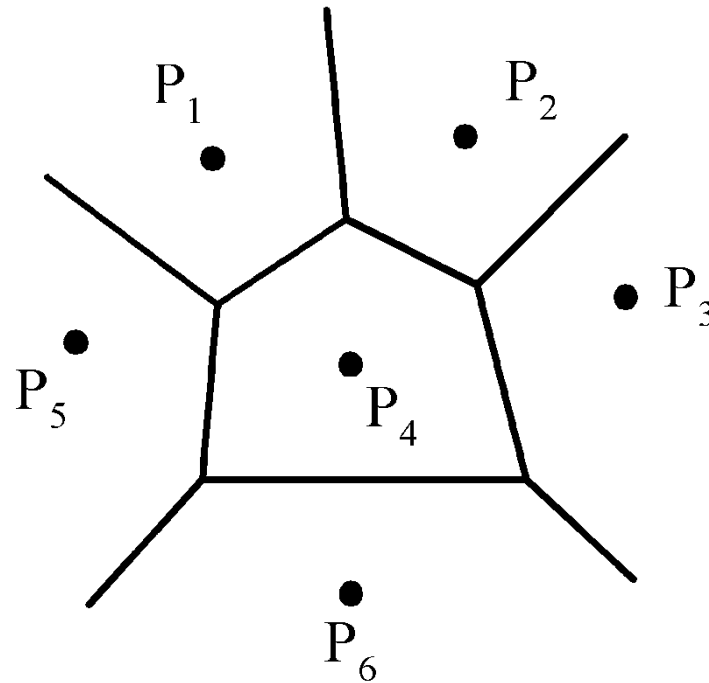
# Definition of Voronoi diagrams

- **Def** : Given two points  $P_i, P_j \in S$ , let  $H(P_i, P_j)$  denote the half plane containing  $P_i$ . The Voronoi polygon associated with  $P_i$  is defined as

$$\bigcap_{i \neq j} H(P_i, P_j)$$

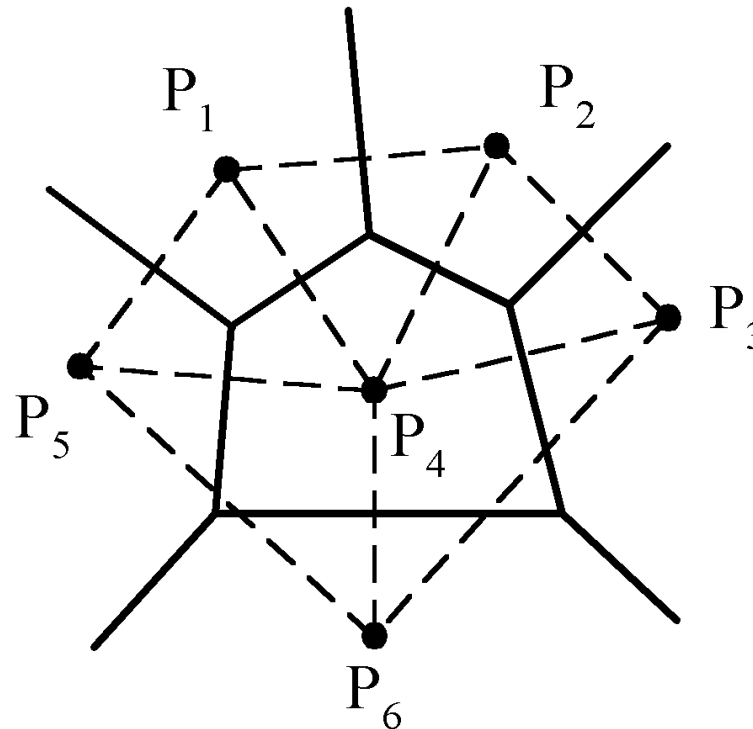


- Given a set of  $n$  points, the Voronoi diagram consists of all the Voronoi polygons of these points.



- The vertices of the Voronoi diagram are called Voronoi points and its segments are called Voronoi edges.

# Delaunay triangulation



- The straight line dual of a Voronoi diagram is called the *Delaunay triangulation*, in honor of a famous French mathematician.
- There is a line segment connecting  $P_i$  and  $P_j$  in a Delaunay triangulation if and only if the Voronoi polygons of  $P_i$  and  $P_j$  share the same edge.

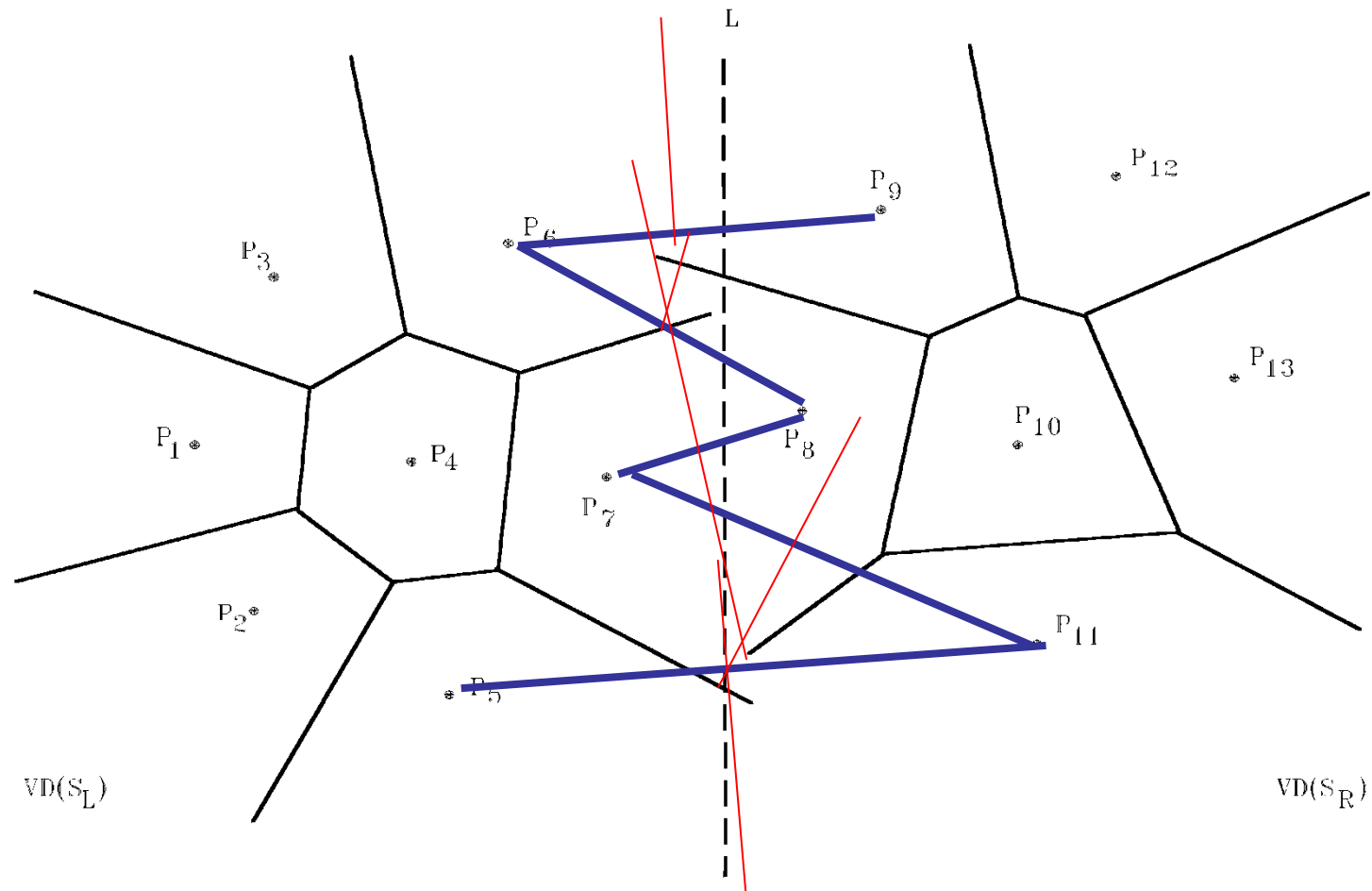


# Application of Voronoi Diagram

- Voronoi diagrams are very useful for many purposes:
  - We can solve the so called **all closest pairs problem** by extracting information from the Voronoi diagram.
  - A **minimal spanning tree** can also be found from the Voronoi diagram.

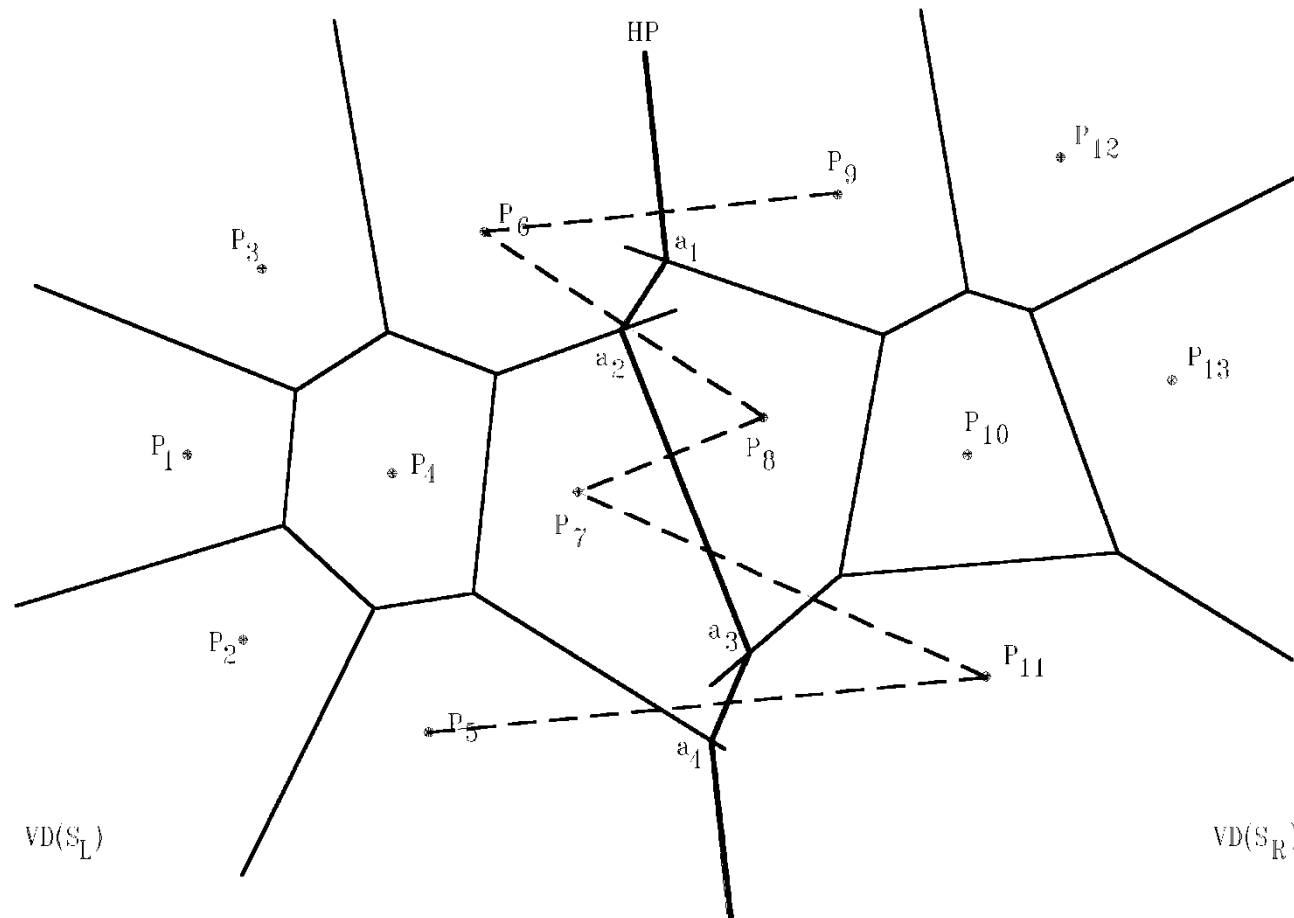
# Example for constructing Voronoi diagrams

- Divide the points into two parts.



# Merging two Voronoi diagrams

- Merging along the **piecewise linear hyperplane HP**

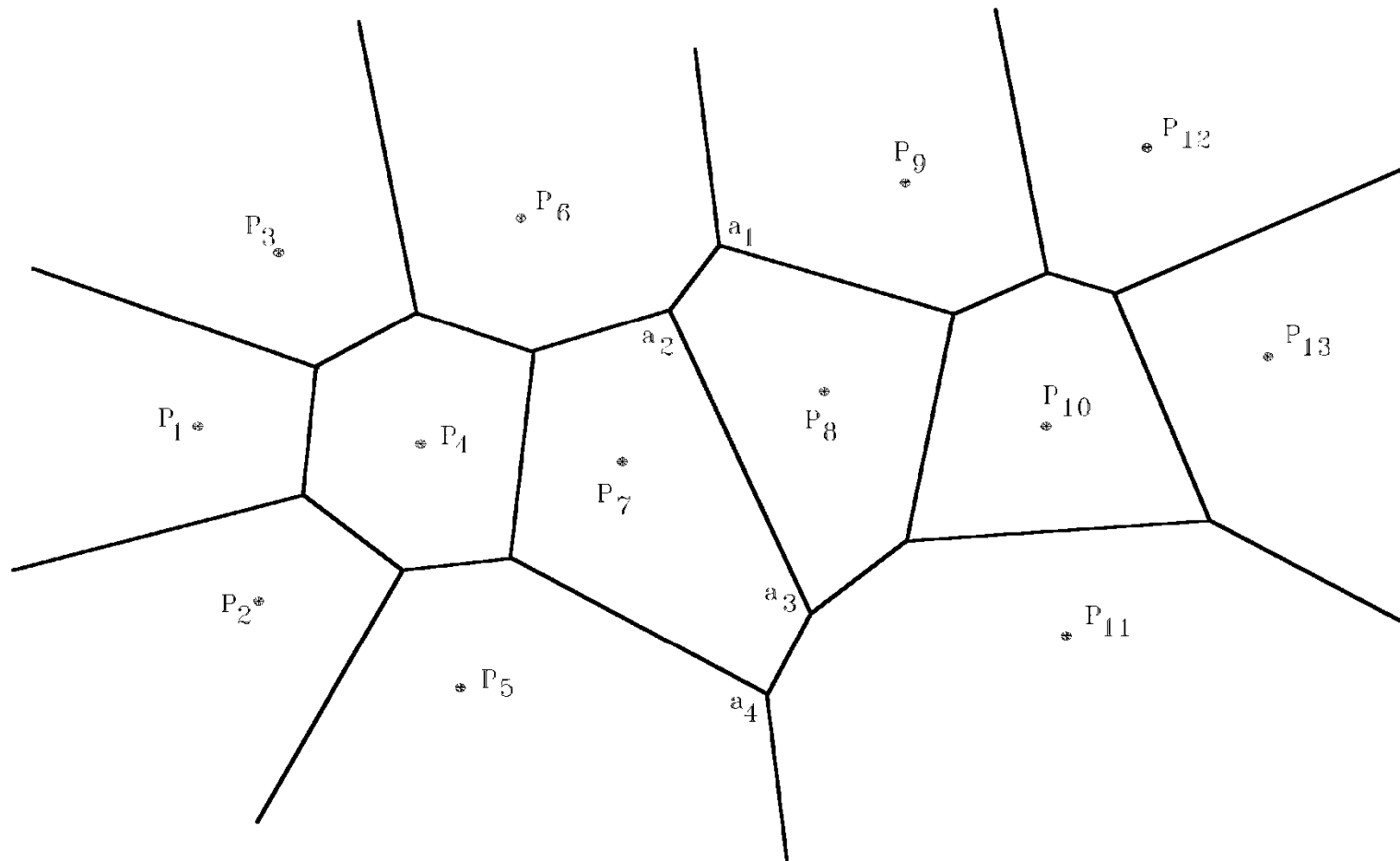


# Property of HP

- If a point  $P$  is within the left(right) side of HP, the nearest neighbor of  $P$  must be a point in  $S_L(S_R)$ .
- After discarding all of  $VD(S_L)$  to the right of HP and all of  $VD(S_R)$  to the left of HP, we obtain the resulting Voronoi diagram.

# The final Voronoi diagram

- After merging



# Divide-and-conquer for Voronoi diagram

- Input: A set  $S$  of  $n$  planar points.
- Output: The Voronoi diagram of  $S$ .

Step 1: If  $S$  contains only one point, return.

Step 2: Find a median line  $L$  perpendicular to the  $X$ -axis which divides  $S$  into  $S_L$  and  $S_R$  such that  $S_L$  ( $S_R$ ) lies to the left(right) of  $L$  and the sizes of  $S_L$  and  $S_R$  are equal.

Step 3: Construct Voronoi diagrams of  $S_L$  and  $S_R$  recursively.  
Denote these Voronoi diagrams by  $VD(S_L)$  and  $VD(S_R)$ .

Step 4: Construct a dividing piece-wise linear hyperplane HP which is the locus of points simultaneously closest to a point in  $S_L$  and a point in  $S_R$ .

Discard all segments of  $VD(S_L)$  which lie to the right of HP and all segments of  $VD(S_R)$  that lie to the left of HP.  
The resulting graph is the Voronoi diagram of  $S$ .

# Merging Two Voronoi Diagrams into One Voronoi Diagram

- Input: (a)  $S_L$  and  $S_R$  where  $S_L$  and  $S_R$  are divided by a perpendicular line  $L$ .

(b)  $VD(S_L)$  and  $VD(S_R)$ .

- Output:  $VD(S)$  where  $S = S_L \cap S_R$

Step 1: Find the convex hulls of  $S_L$  and  $S_R$ , denoted as  $Hull(S_L)$  and  $Hull(S_R)$ , respectively. (A special algorithm for finding a convex hull in this case will be given later.)



**Step 2:** Find segments  $\overline{P_a P_b}$  and  $\overline{P_c P_d}$  which join  $\text{HULL}(S_L)$  and  $\text{HULL}(S_R)$  into a convex hull ( $P_a$  and  $P_c$  belong to  $S_L$  and  $P_b$  and  $P_d$  belong to  $S_R$ ) Assume that  $\overline{P_a P_b}$  lies above  $\overline{P_c P_d}$ . Let  $x = a$ ,  $y = b$ ,  $\text{SG} = \overline{P_x P_y}$  and  $\text{HP} = \emptyset$ .

**Step 3:** Find the perpendicular bisector of SG. Denote it by BS. Let  $\text{HP} = \text{HP} \cup \{\text{BS}\}$ . If  $\text{SG} = \overline{P_c P_d}$ , go to Step 5; otherwise, go to Step 4.

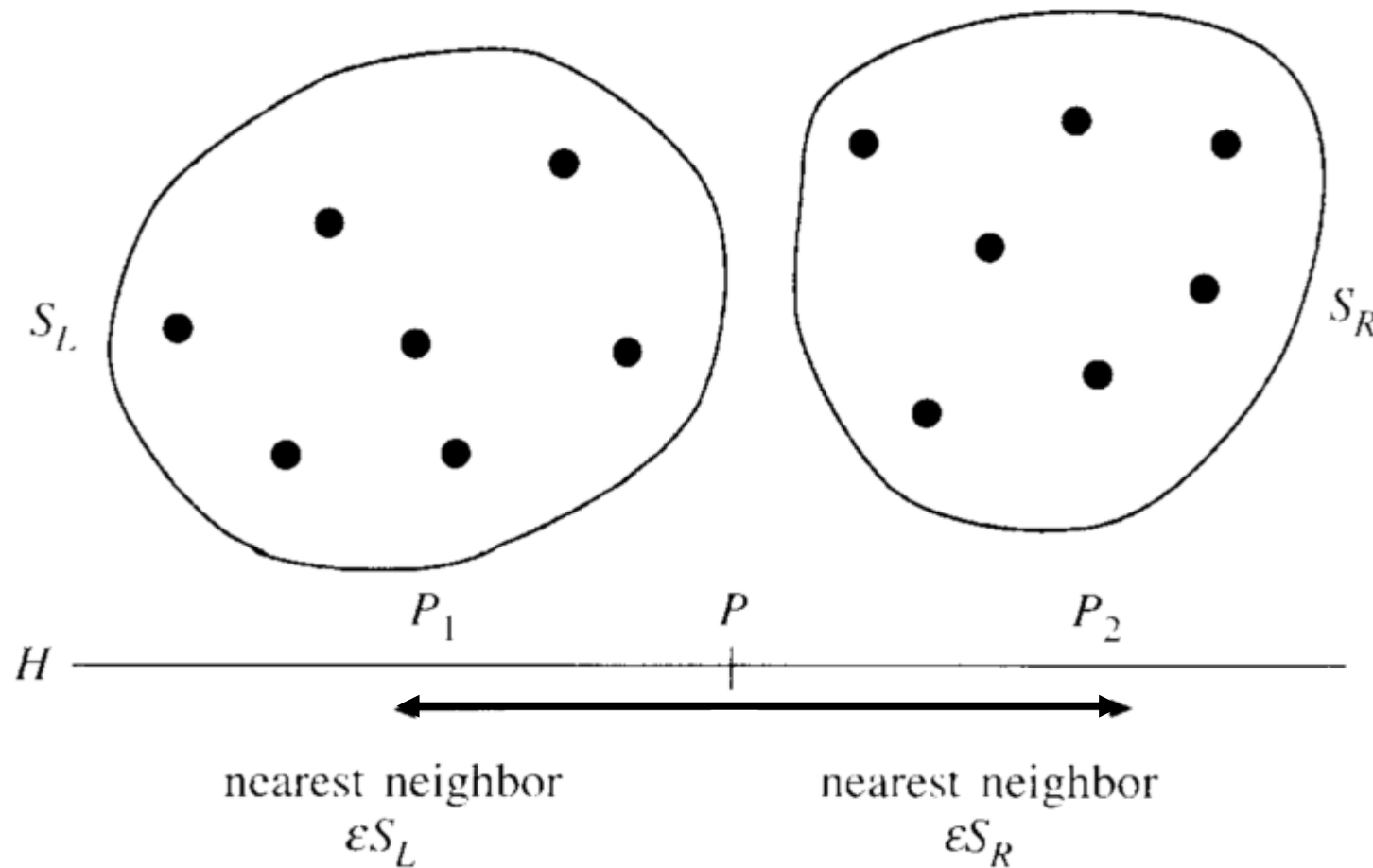
**Step 4:** The ray from  $VD(S_L)$  and  $VD(S_R)$  which BS first intersects with must be a perpendicular bisector of either  $\overline{P_x P_z}$  or  $\overline{P_y P_z}$  for some  $z$ . If this ray is the perpendicular bisector of  $\overline{P_y P_z}$ , then let  $SG = \overline{P_x P_z}$ ; otherwise, let  $SG = \overline{P_z P_y}$ . Go to Step 3.

**Step 5:** Discard the edges of  $VD(S_L)$  which extend to the right of HP and discard the edges of  $VD(S_R)$  which extend to the left of HP. The resulting graph is the Voronoi diagram of  $S = S_L \cup S_R$ .

# Properties of Voronoi Diagrams

- **Def :** Given a point  $P$  and a set  $S$  of points, the **distance between  $P$  and  $S$**  is the distance between  $P$  and  $P_i$  which is the nearest neighbor of  $P$  in  $S$ .
- The HP obtained from the above algorithm is the locus of points which keep equal distances to  $S_L$  and  $S_R$  .
- The HP is monotonic in  $y$ .

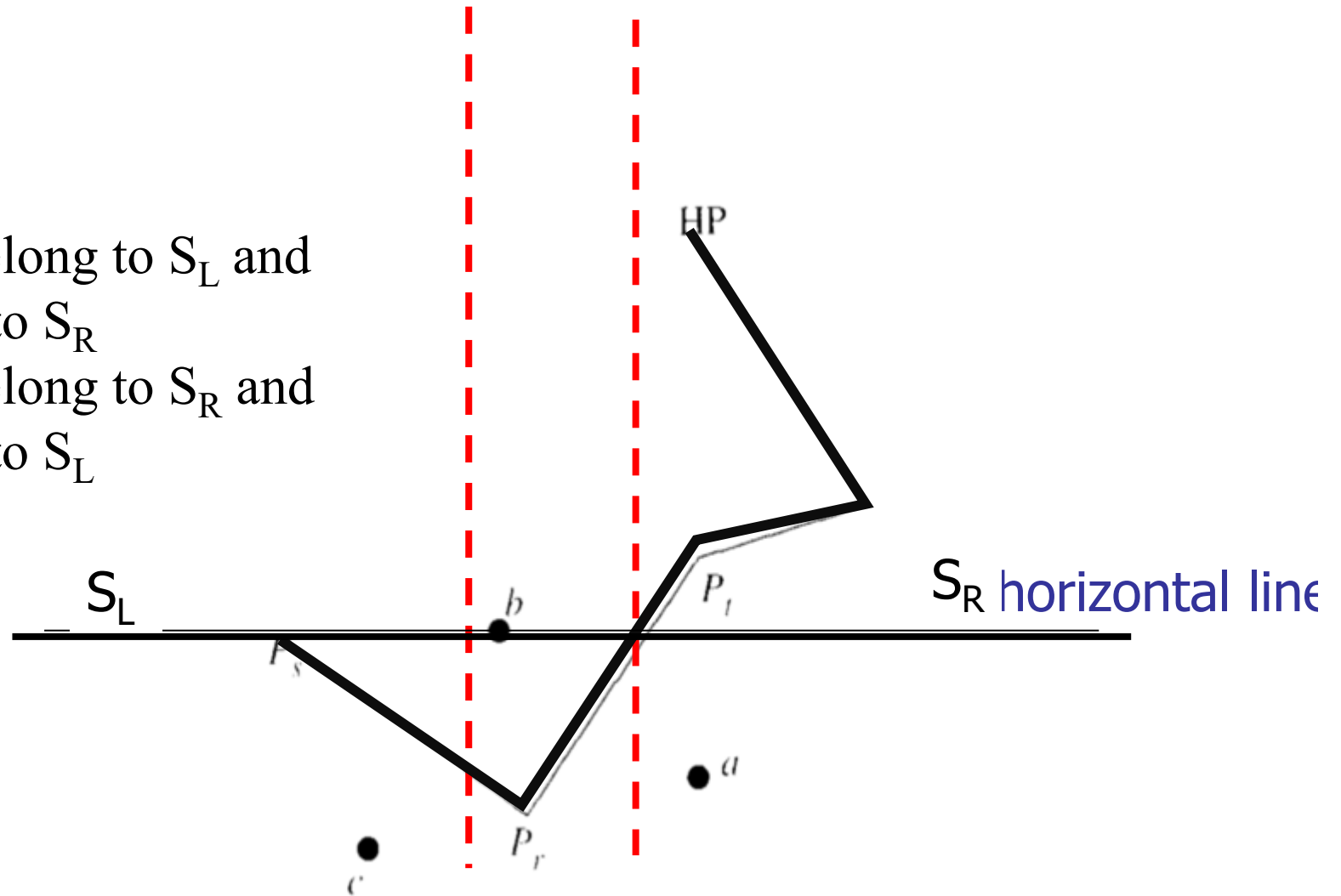
# The relationship between a horizontal line $H$ and $S_L$ and $S_R$ .



Each horizontal line  $H$  intersects with  $HP$  at one and only on point.

The HP is monotonic in  $y$ .

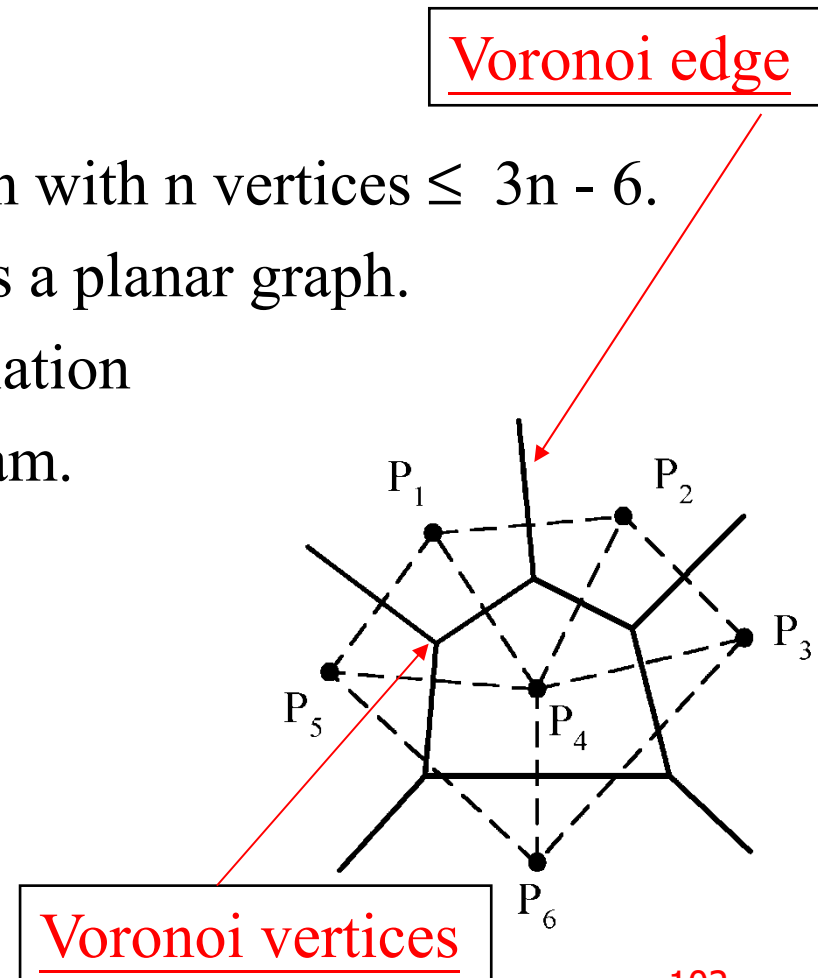
- (1)  $a, c$  belong to  $S_L$  and  $b$  belongs to  $S_R$
- (2)  $a, c$  belong to  $S_R$  and  $b$  belongs to  $S_L$



# # of Voronoi edges

- # of edges of a Voronoi diagram  $\leq 3n - 6$ , where  $n$  is # of points.
- Reasoning:
  - i. # of edges of a planar graph with  $n$  vertices  $\leq 3n - 6$ .
  - ii. A Delaunay triangulation is a planar graph.
  - iii. Edges in Delaunay triangulation  
 $\xleftrightarrow{1-1}$  edges in Voronoi diagram.

**Corollary:** If  $G$  is a connected planar simple graph with  $E$  edges and  $V$  vertices where  $V \geq 3$ , then  $E \leq 3V - 6$ .



# # of Voronoi vertices

- # of Voronoi vertices  $\leq 2n - 4$  (upper bound).

- Reasoning:

- i. Let  $F$ ,  $E$  and  $V$  denote # of face(region), edges and vertices in a planar graph.

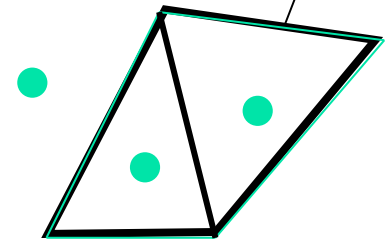
**Euler's relation:**  $F = E - V + 2$ .

- ii. In a Delaunay triangulation,

$$V = n, E \leq 3n - 6$$

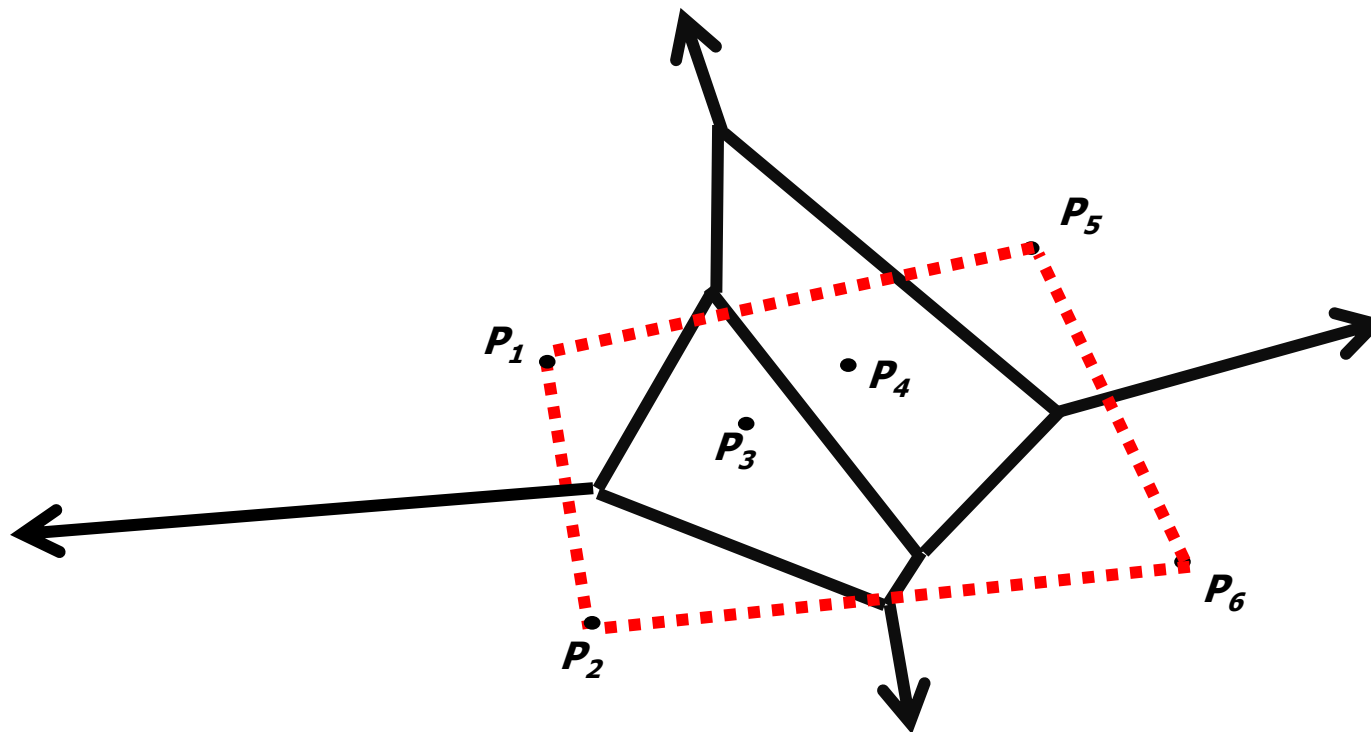
$$\Rightarrow F = E - V + 2 \leq 3n - 6 - n + 2 = 2n - 4.$$

Voronoi vertices



# Construct a convex hull from a Voronoi diagram

- After a Voronoi diagram is constructed, a convex hull can be found in  $O(n)$  time.
- Connecting the points associated with the **infinite rays**.





# Construct Convex Hull from Voronoi diagram

Step 1: Find an infinite ray by examining all Voronoi edges.  $O(n)$

Step 2: Let  $P_i$  be the point to the left of the infinite ray.  $P_i$  is a convex hull vertex. Examine the **Voronoi polygon** of  $P_i$  to find the next infinite ray.

Step 3: Repeat Step 2 until we return to the Starting ray.

# Time complexity

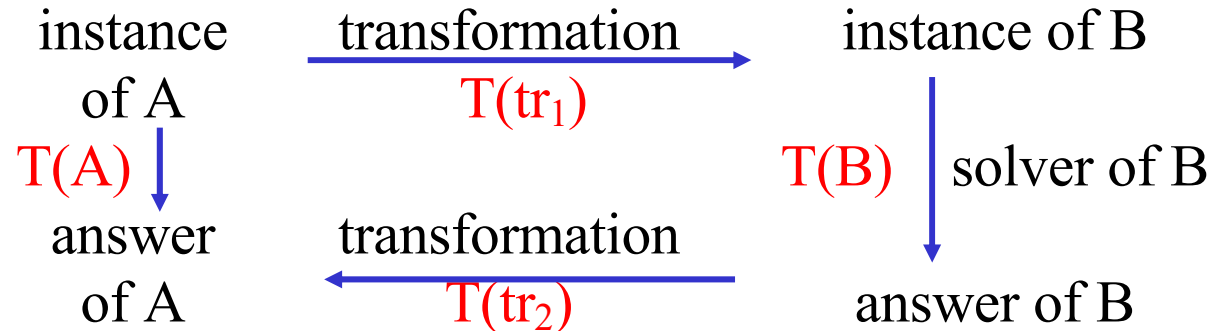
- Time complexity for merging 2 Voronoi diagrams:  
Total:  $O(n)$ 
  - Step 1:  $O(n)$
  - Step 2:  $O(n)$
  - Step 3 ~ Step 5:  $O(n)$   
(at most  $3n - 6$  edges in  $VD(S_L)$  and  $VD(S_R)$  and at most  $n$  segments in HP)
- Time complexity for constructing a Voronoi diagram:  $O(n \log n)$   
because  $T(n) = 2T(n/2) + O(n) = O(n \log n)$

# Finding lower bound by problem transformation

Problem A reduces to problem B ( $A \propto B$ )

iff A can be solved by using any algorithm which solves B.

If  $A \propto B$ , B is more difficult.



Note:  $T(tr_1) + T(tr_2) < T(B)$

$$T(A) \leq T(tr_1) + T(tr_2) + T(B) \sim O(T(B))$$

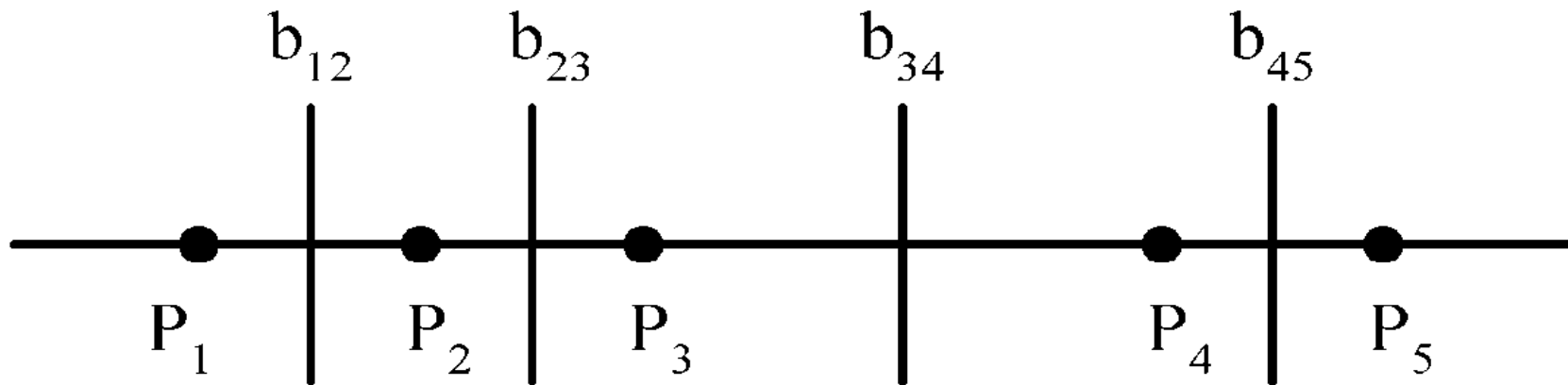
# Lower bound of the Voronoi diagram

- Let us consider a set of points on a **straight line**.
- The Voronoi diagram of such a set of points consists of a set of bisecting lines.
- **After these lines have been constructed, a linear scanning of these Voronoi edges will accomplish the function of sorting.**
- In other words, *the Voronoi diagram problem can not be easier than the sorting problem.*
- A lower bound of the Voronoi diagram problem is therefore  $\Omega(n \log n)$  and the algorithm is consequently optimal.

# Lower bound

- The lower bound of the Voronoi diagram problem is  $\Omega(n \log n)$ .

sorting  $\propto$  Voronoi diagram problem



The Voronoi diagram for a set of points on a straight line

# Applications of Voronoi diagrams

# Applications of Voronoi diagrams

- The Euclidean nearest neighbor searching problem.
- The Euclidean all nearest neighbor problem.

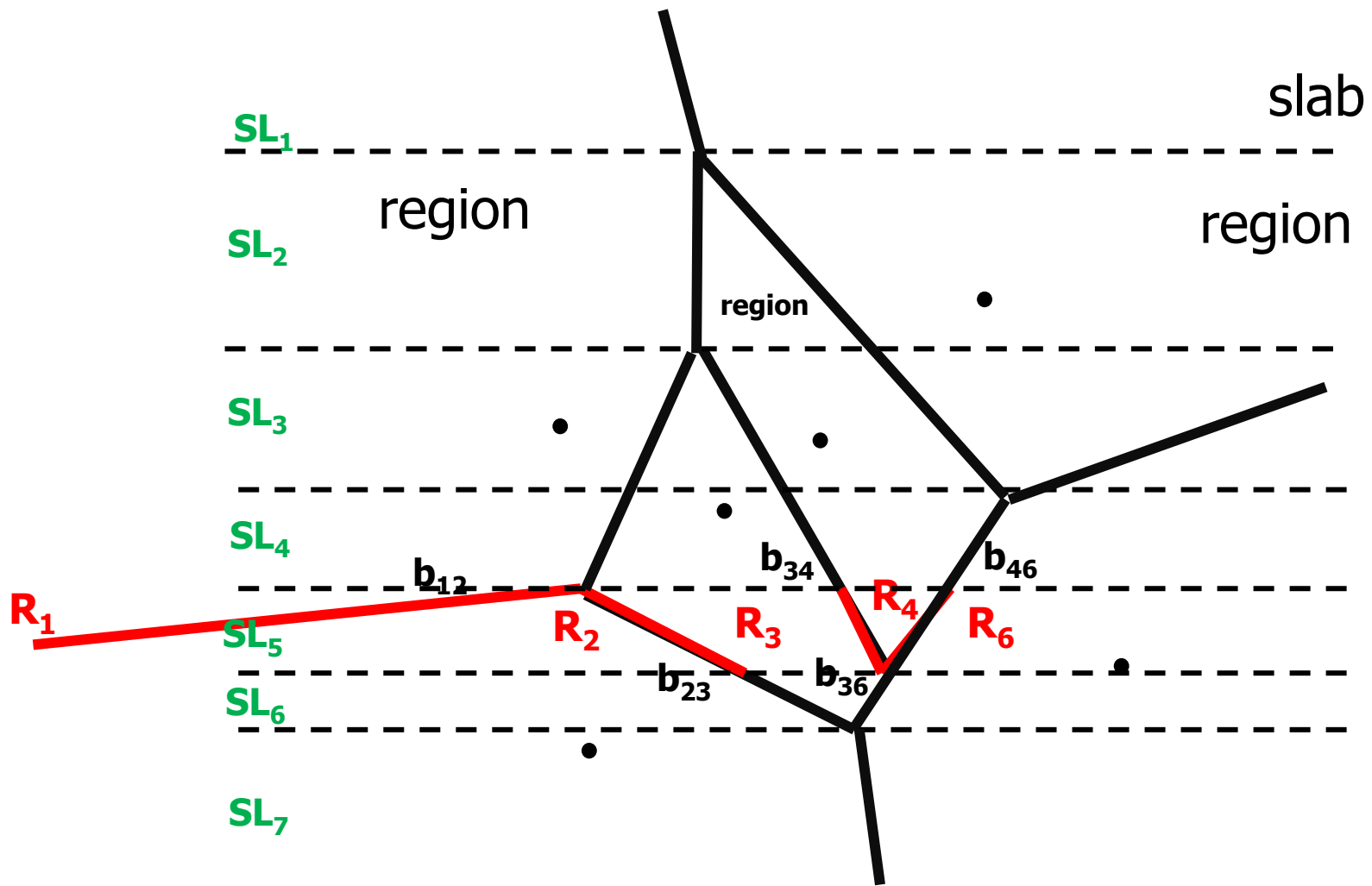
# The Euclidean nearest neighbor searching problem.

- The Euclidean nearest neighbor searching problem is defined as follows: We are given a set of  $n$  planar points:  $P_1, P_2, \dots, P_n$ , and a testing point  $P$ . Our problem is to find **a nearest neighbor of  $P$**  among  $P_i$ 's and the distance used is the Euclidean distance.
- A straightforward method is to conduct an exhaustive search. This algorithm would be an  **$O(n)$**  algorithm.
- Using the Voronoi diagram, we can reduce the searching time to  **$O(\log n)$**  with preprocessing time  **$O(n \log n)$** .



- Note that the Voronoi diagram divides the entire plane into regions  $R_1, R_2, \dots, R_n$ . Within each region  $R_i$ , there is a point  $P_i$ .
- If a testing point falls within region  $R_i$ , then its nearest neighbor, among all points, is  $P_i$ .
- Therefore, we may avoid an exhaustive search by simply transforming the problem into a region location problem.
- That is, if we can determine which region  $R_i$  a testing point is located, we can determine a nearest neighbor of this testing point.

- A Voronoi diagram is a planar graph.
- Our first step is to sort these Voronoi vertices according to their y-values.
- The Voronoi vertices are labeled  $V_1, V_2, \dots, V_6$  according to their decreasing y-values. For each Voronoi vertex, a horizontal line is drawn passing this vertex.
- These horizontal lines divide the entire space into slabs.



# Euclidean nearest neighbor searching algorithm

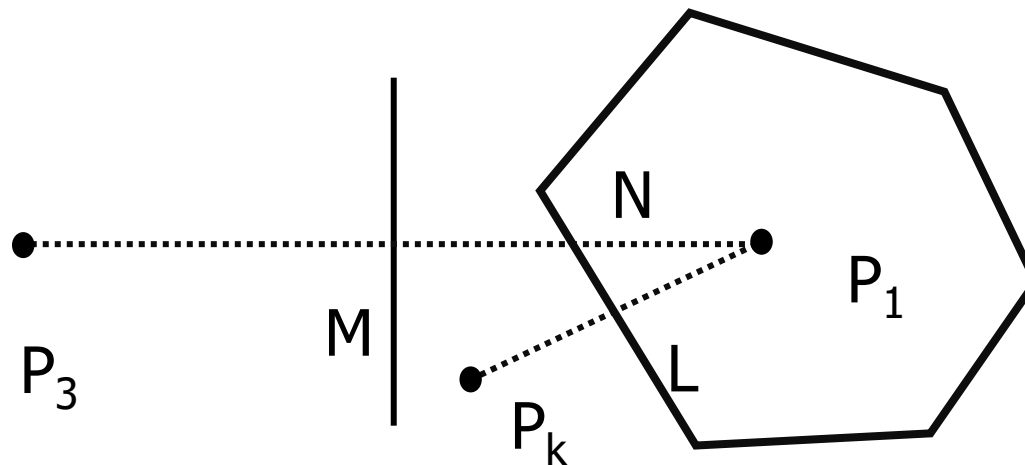
- Conduct a binary search to **determine which slab** this testing point is located. Since there are at most  $O(n)$  Voronoi vertices, this can be done in  $O(\log n)$  time.
- Within each slab, conduct a binary search to **determine which region** this point is located in. Since there are at most  $O(n)$  Voronoi edges, this can be done in  $O(\log n)$  time.
- The total searching time is  $O(\log n)$ .
- It is easy to see that the **preprocessing time** is  $O(n \log n)$ , essentially the time needed to construct the Voronoi diagram.

# The Euclidean all nearest neighbor problem.

- We are given a set of  $n$  planar points  $P_1, P_2, \dots, P_n$ .
- The Euclidean **closest pair problem** is to find a nearest neighbor of every  $P_i$ .
- Properties:
  - *If  $P_j$  is a nearest neighbor of  $P_i$ , then  $P_i$  and  $P_j$  share the same Voronoi edge.*
  - *Moreover, the **midpoint of** segment  $P_iP_j$  is located exactly on this commonly shared Voronoi edge.*

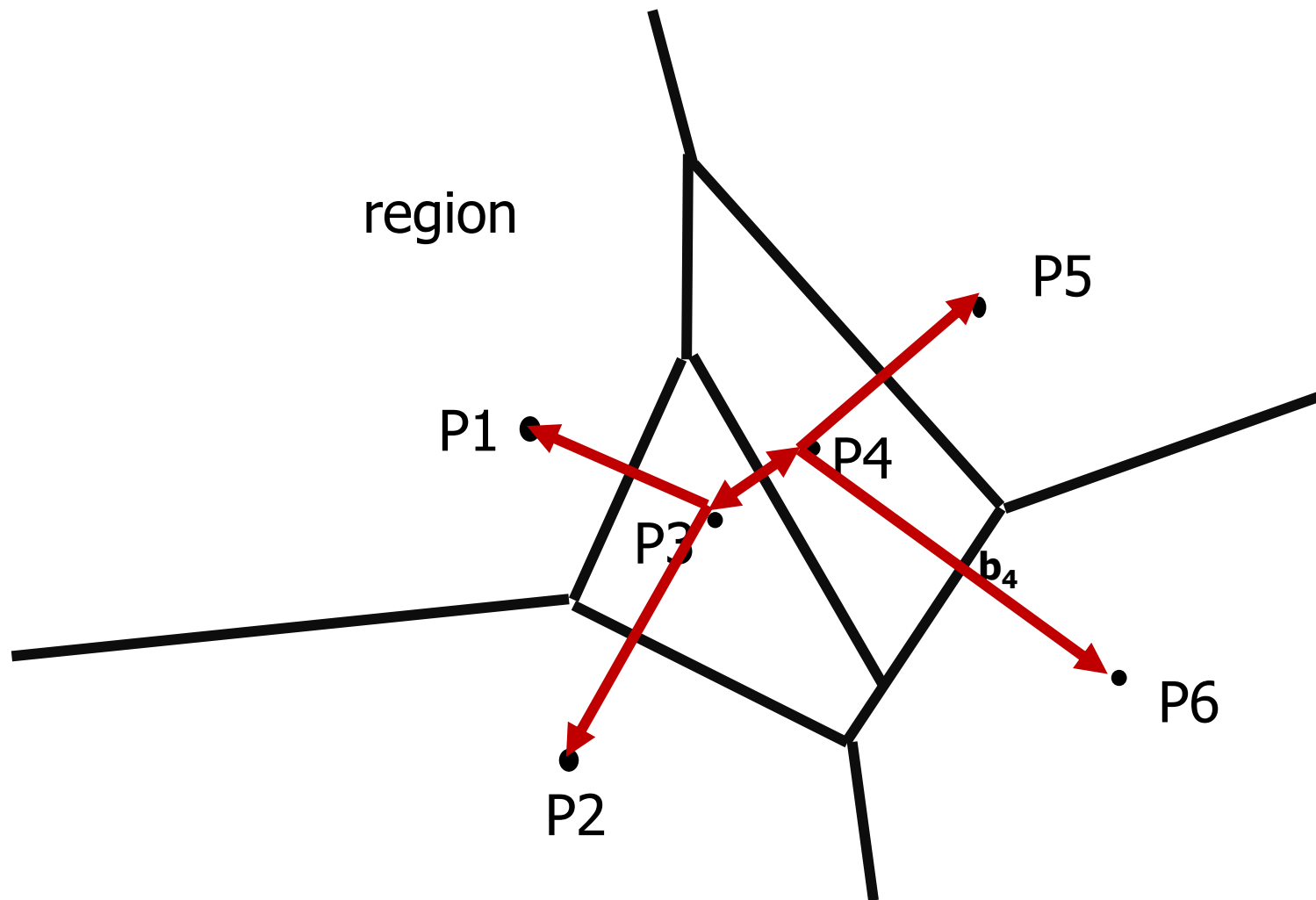
# Proof

- We shall show this **property by contradiction**.
- Suppose that  $P_i$  and  $P_j$  do not share the same Voronoi edge.  
By the definition of Voronoi polygons, the perpendicular bisector of  $P_i P_j$  must be outside of the Voronoi polygon associated with  $P_i$ .
- Let  $P_i$  intersect the bisector at  $M$  and some Voronoi edge at  $N$ .



# Euclidean all nearest neighbor problem

- Given the above property, the Euclidean all nearest neighbor problem can be solved by examining every Voronoi edge of each Voronoi polygon.
- Since each Voronoi edge is shared by **exactly two Voronoi polygons**, no Voronoi edge is examined more than twice.
- That is, this Euclidean all nearest neighbor problem can be solved in linear time after the Voronoi diagram is constructed.
- Thus this problem can be solved in  **$O(n \log n)$  time**.





# Matrix multiplication

# Matrix multiplication

- Let  $A$ ,  $B$  and  $C$  be  $n \times n$  matrices

$$C = AB$$

$$C(i, j) = \sum_{1 \leq k \leq n} A(i, k)B(k, j)$$

- The straightforward method to perform a matrix multiplication requires  $O(n^3)$  time.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$$

$n$ 個乘法,  $n-1$ 個加法,  
產生了一個entry, 共有  $n^2$ 個  
entries

# Divide-and-conquer approach

- $C = AB$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

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

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

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

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

- Time complexity:

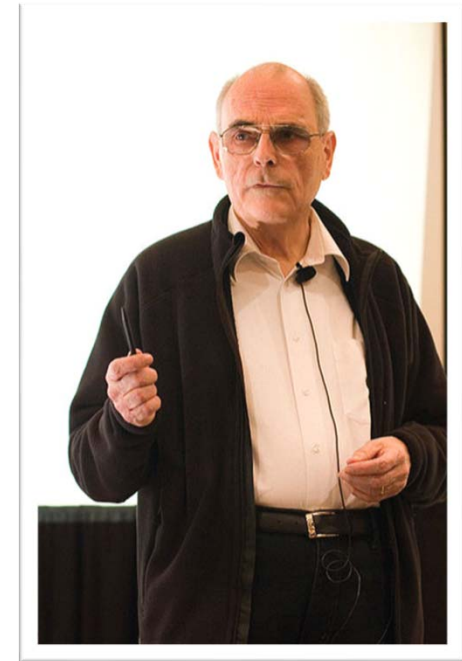
$$T(n) = \begin{cases} b & , n \leq 2 \\ 8T(n/2) + cn^2 & , n > 2 \end{cases}$$

$$T(n) = \Theta(1) + 8T\left(\frac{n}{2}\right) + \Theta(n^2) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

(# of additions :  $n^2$ ) We get  $T(n) = O(n^3)$

# Strassen's matrix multiplication

- $P = (A_{11} + A_{22})(B_{11} + B_{22})$   
 $Q = (A_{21} + A_{22})B_{11}$   
 $R = A_{11}(B_{12} - B_{22})$   
 $S = A_{22}(B_{21} - B_{11})$   
 $T = (A_{11} + A_{12})B_{22}$   
 $U = (A_{21} - A_{11})(B_{11} + B_{12})$   
 $V = (A_{12} - A_{22})(B_{21} + B_{22}).$
- $C_{11} = P + S - T + V$   
 $C_{12} = R + T$   
 $C_{21} = Q + S$   
 $C_{22} = P + R - Q + U$



德國數學家  
Volker  
Strassen  
攝於2009年

# Time complexity

- 7 multiplications and 18 additions or subtractions
- Time complexity:

$$T(n) = \begin{cases} b & , n \leq 2 \\ 7T(n/2) + an^2 & , n > 2 \end{cases}$$

$$\begin{aligned} T(n) &= an^2 + 7T(n/2) \\ &= an^2 + 7(a(n/2)^2 + 7T(n/4)) \\ &= an^2 + (7/4)an^2 + 7^2T(n/4) \\ &= \dots \\ &\vdots \\ &= an^2(1 + 7/4 + (7/4)^2 + \dots + (7/4)^{k-1} + 7^k T(1)) \\ &\leq cn^2(7/4)^{\log_2 n} + 7^{\log_2 n}, \quad c \text{ is a constant} \\ &= cn^{\log_2 4 + \log_2 7 - \log_2 4} + n^{\log_2 7} \\ &= O(n^{\log_2 7}) \\ &\cong O(n^{2.81}) \end{aligned}$$