Chapter 4

The Divide-and-Conquer Strategy

Outlines

4-1 The 2-Dimensional Maxima Finding Problem

4-2 The Closest Pair Problem

4-3 The Convex Hull Problem

4-4 The Voronoi Diagrams Constructed by the Divide-and-Conquer Strategy

4-5 Applications of the Voronoi Diagrams

4-6 Matrices Multiplication

Introduction

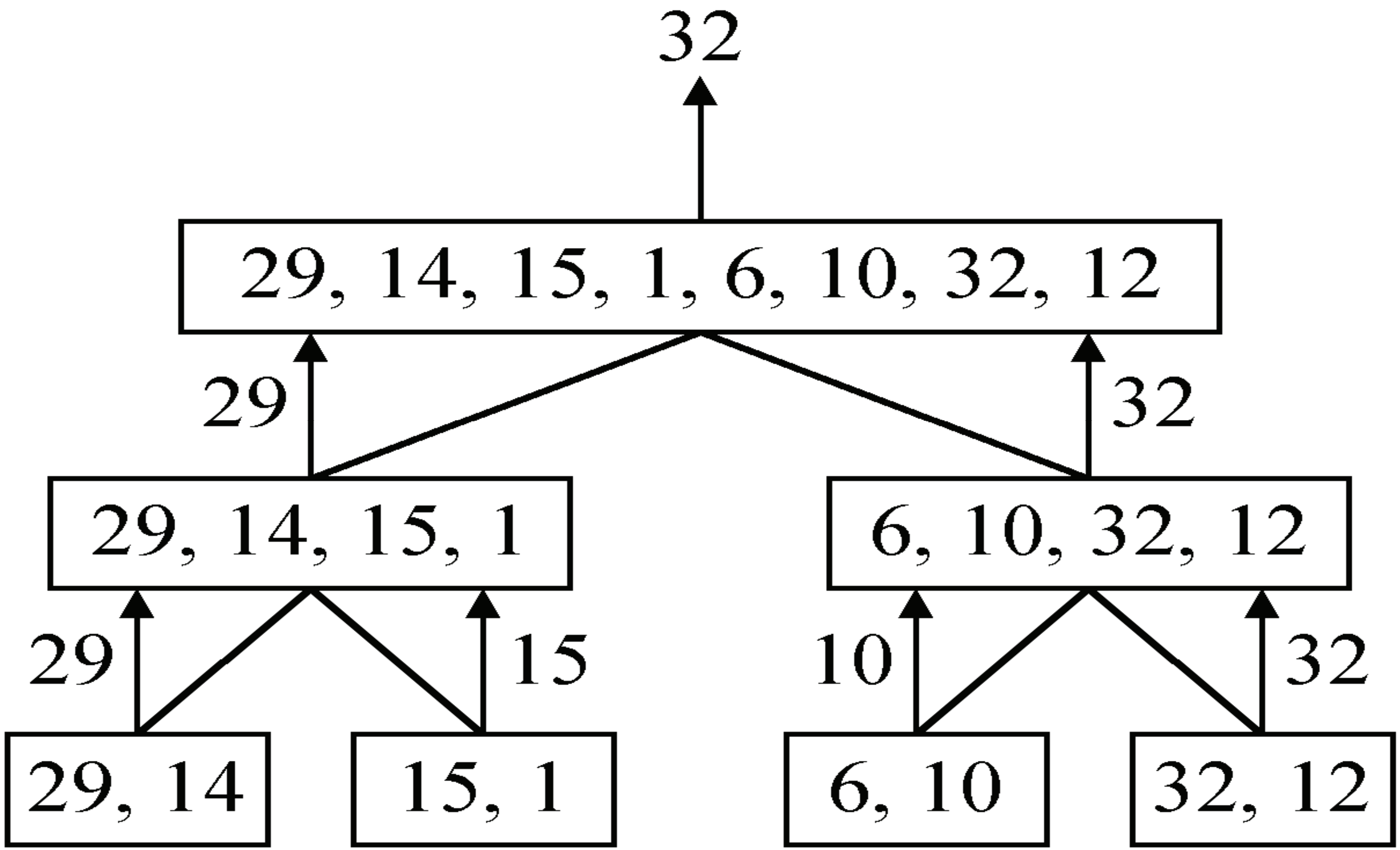
- divide-and-conquer strategy
  - first divides a problem into two smaller sub-problems and each sub-problem is identical to its original problem, except its input size is smaller.
  - Both sub-problems are then solved and the sub-solutions are finally merged into the final solution.
- These two sub-problems 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

- finding the maximum of a set S of n numbers



Time Complexity

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

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

- 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.

Time complexity

- Time complexity:

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

- Calculation of T(n):

Assume $n = 2^k$,

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

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.

Time complexity of the general algorithm

- Time complexity:

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

where S(n) : time for splitting
M(n) : time for merging
b : a constant
c : a constant

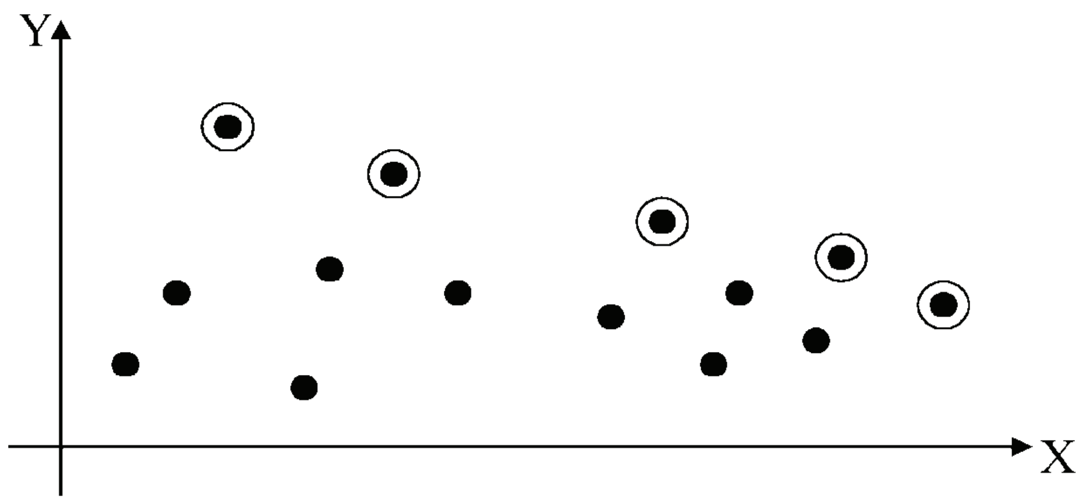
- e.g. Binary search
- e.g. quick sort
- e.g. merge sort

4.1 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.

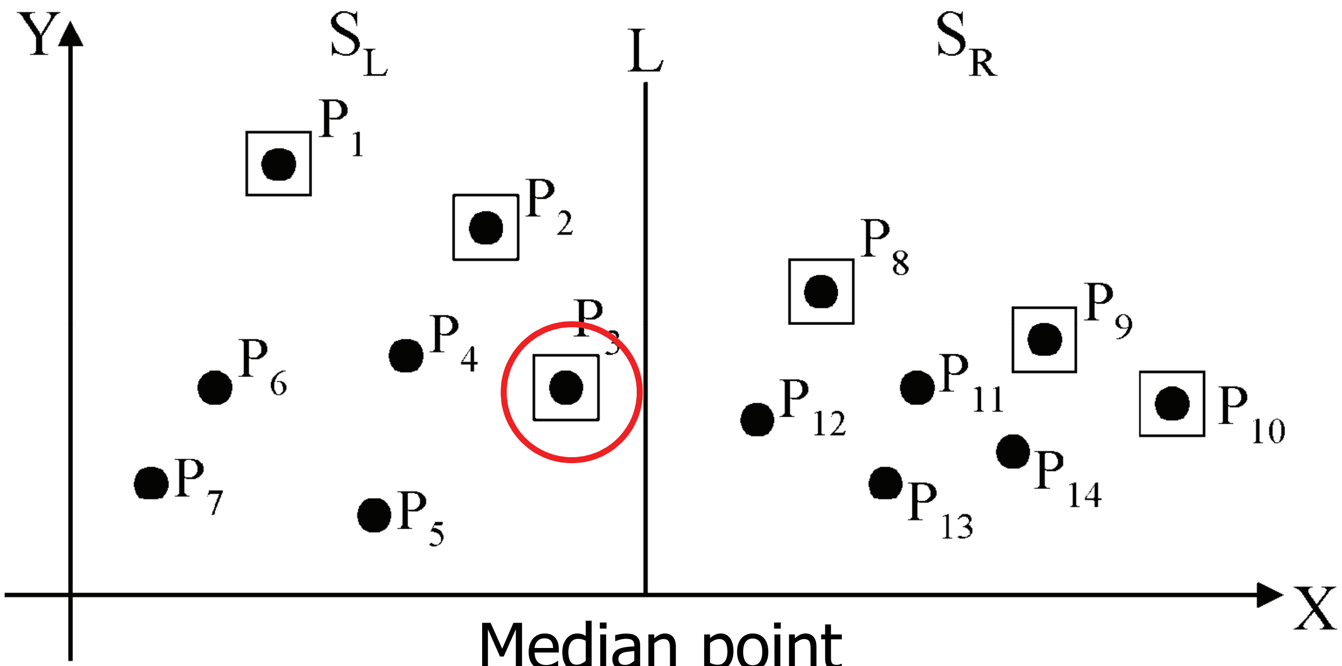
Time complexity:

$O(n^2)$



Divide-and-conquer for maxima finding

Perpendicular line



Median point

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

  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. If this is done, the merging complexity is $O(n)$ and the total number of time steps needed is O(nlogn) + $T(n)$

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(nlogn). Thus the total time-complexity of using the divide-and-conquer strategy to find maximal points with presorting is O(nlogn).

Recurrence

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

is a *recurrence.*

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

Recurrence (Examples)

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

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

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

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

Iteration Method

- Expand the recurrence
- Work some algebra to express as a summation
- Evaluate the summation

Iteration Method (Example)

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

Iteration Method (Example)

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

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

$$\begin{aligned}
&= c + c + T(n\text{-}2) = 2c + T(n\text{-}2) \\
&= 2c + c + T(n\text{-}3) = 3c + T(n\text{-}3) \\
&= \ldots \\
&= kc + T(n\text{-}k) \qquad \text{Set } k = n \\
&= nc + T(0) = nc
\end{aligned}$$

$$\boxed{T(n) = \Theta(n)}$$

Iteration Method (Example)

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

Iteration Method (Example)

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

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

$$\begin{aligned}
&= n + n\text{-}1 + T(n\text{-}2) \\
&= n + n\text{-}1 + n\text{-}2 + T(n\text{-}3) \\
&= \dots \\
&= n + n\text{-}1 + n\text{-}2 + \dots + n\text{-}(k\text{-}1) + T(n\text{-}k) \\
&= n + n\text{-}1 + n\text{-}2 + \dots + 1 + T(0) \\
&= \frac{n(n+1)}{2}
\end{aligned}$$

$$\boxed{T(n) = \Theta(n^2)}$$

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:

$$\underbrace{T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + cn & n > 1 \end{cases}}_{\text{Becomes A Recursion Tree}}$$

Merge Sort (Recursion Tree)

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

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

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

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

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

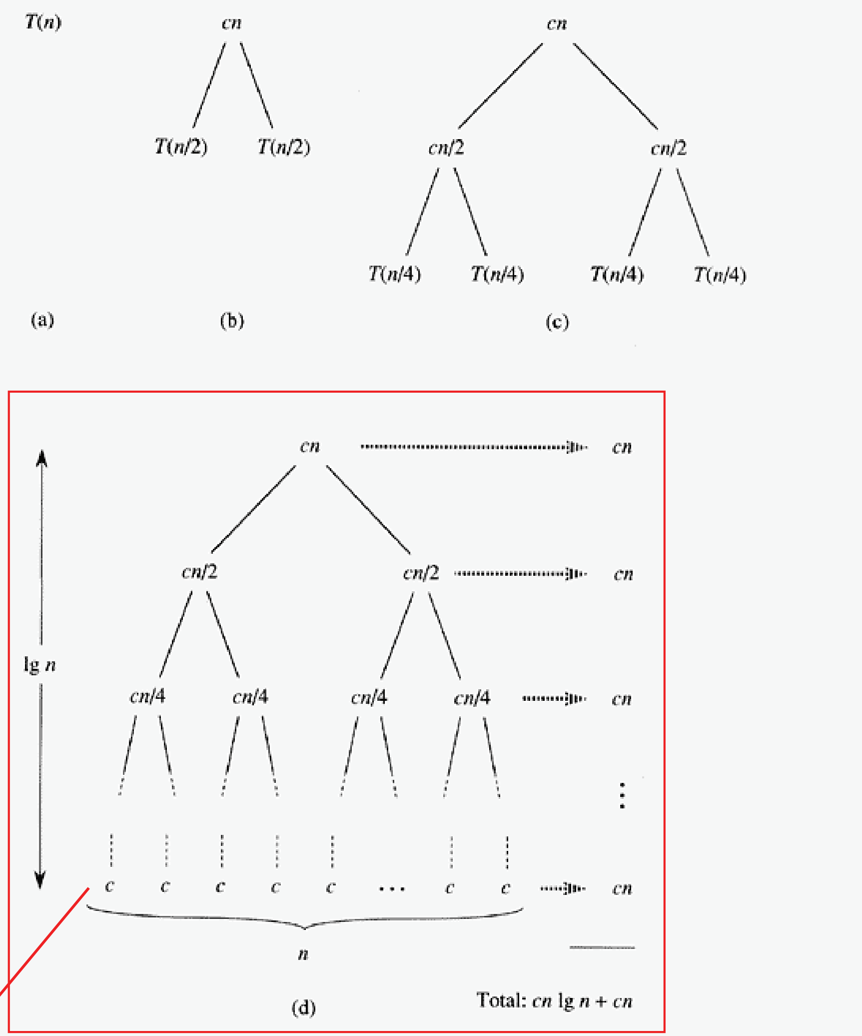


Figure 2.5 The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

4

Merge Sort (Recursion Tree)

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

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

$$\boxed{T(n) = \Theta(n \lg n)}$$

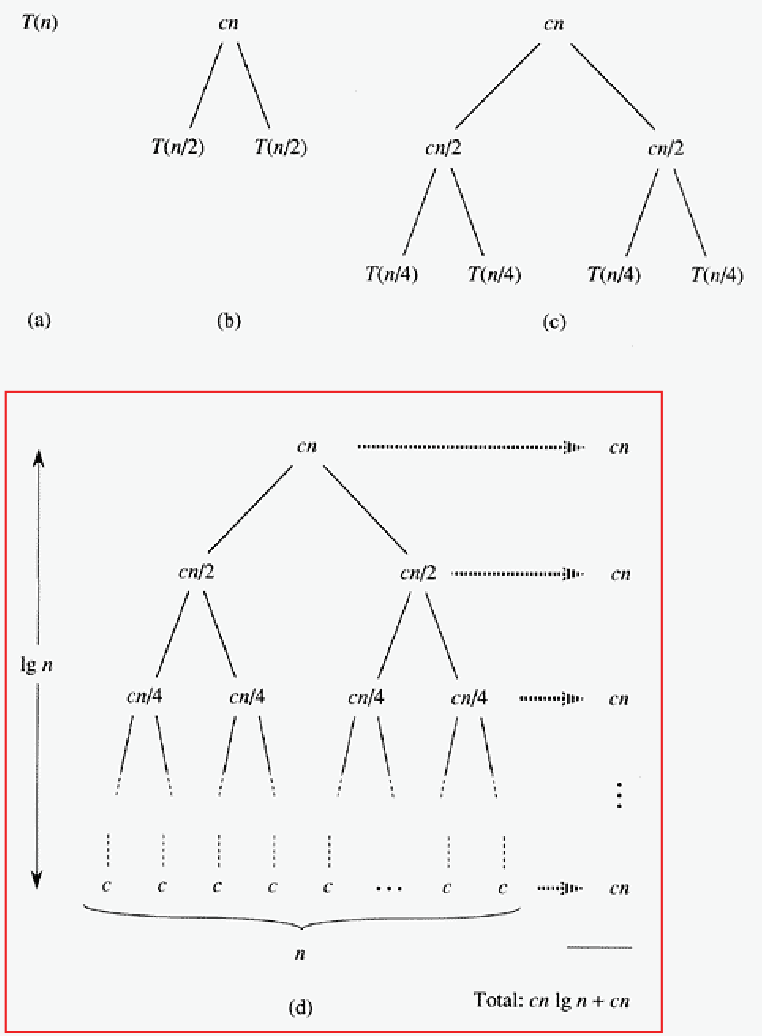


Figure 2.5 The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

Recursion-Tree Method (Example)

$$T(n) = 3T(n/4) + cn^2$$

Recursion-Tree Method (Example)

$$T(n) = 3T(n/4) + cn^2$$

$$= cn^2 + 3T(n/4)$$

$$= cn^2 + \left(\frac{3}{16}\right)cn^2 + 9T(n/16)$$

$$= cn^2 + \left(\frac{3}{16}\right)cn^2 + \left(\frac{3}{16}\right)^2 cn^2$$

$$+ \ldots + \left(\frac{3}{16}\right)^{i-1} cn^2 + 3^i T(n/4^i)$$

$$\frac{n}{4^i} = 1,\ i = \log_4 n$$

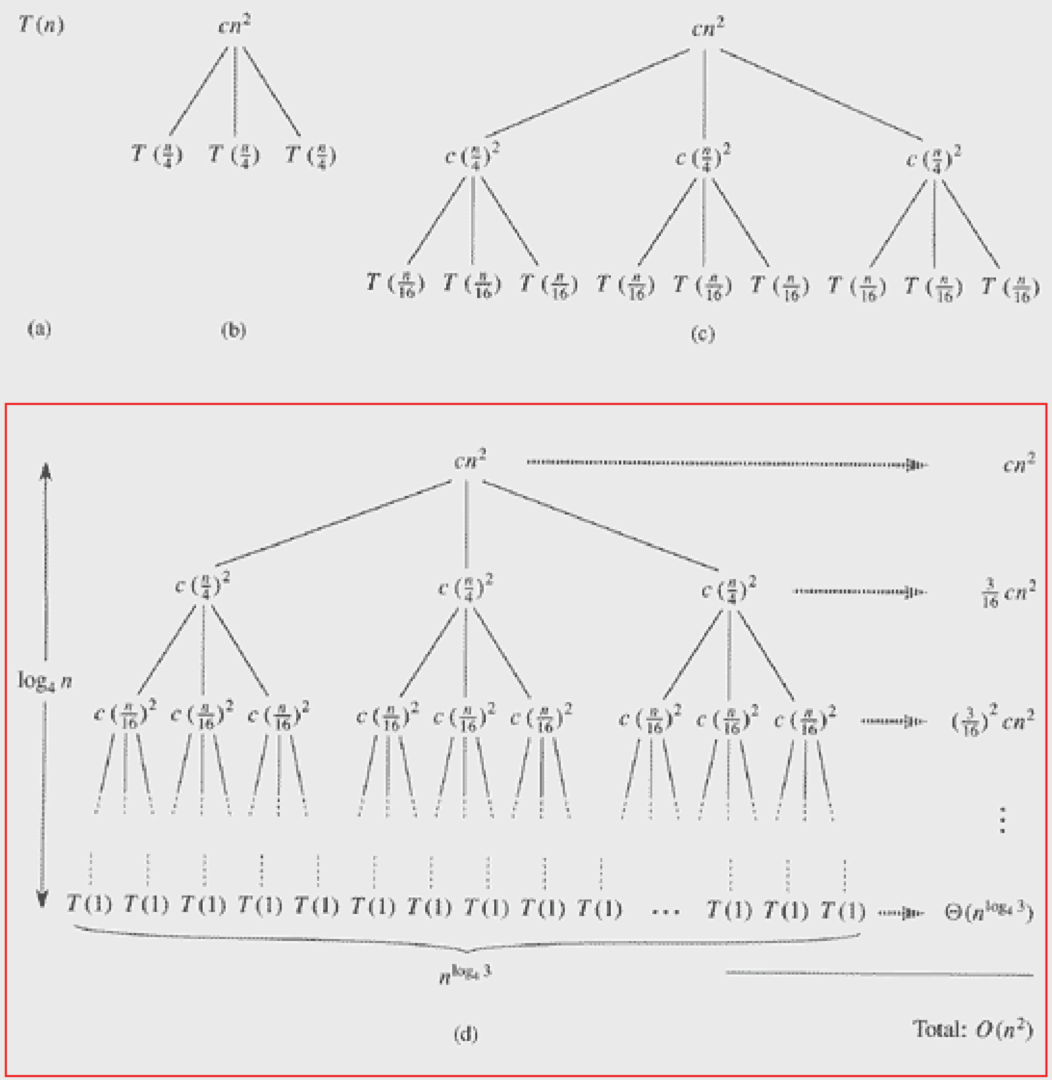


Figure 4.1 The construction of a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

Recursion-Tree Method (Example)

$$T(n) = 3T(n/4) + cn^2$$

$$= cn^2 + 3T(n/4)$$

$$= cn^2 + \left(\frac{3}{16}\right)cn^2 + 9T(n/16)$$

$$= cn^2 + \left(\frac{3}{16}\right)cn^2 + \left(\frac{3}{16}\right)^2 cn^2$$

$$+ \dots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + 3^{\log_4 n} T(1)$$

$$3^{\log_4 n} T(1) = n^{\log_4 3}\Theta(1)$$

$$= \Theta(n^{\log_4 3})$$

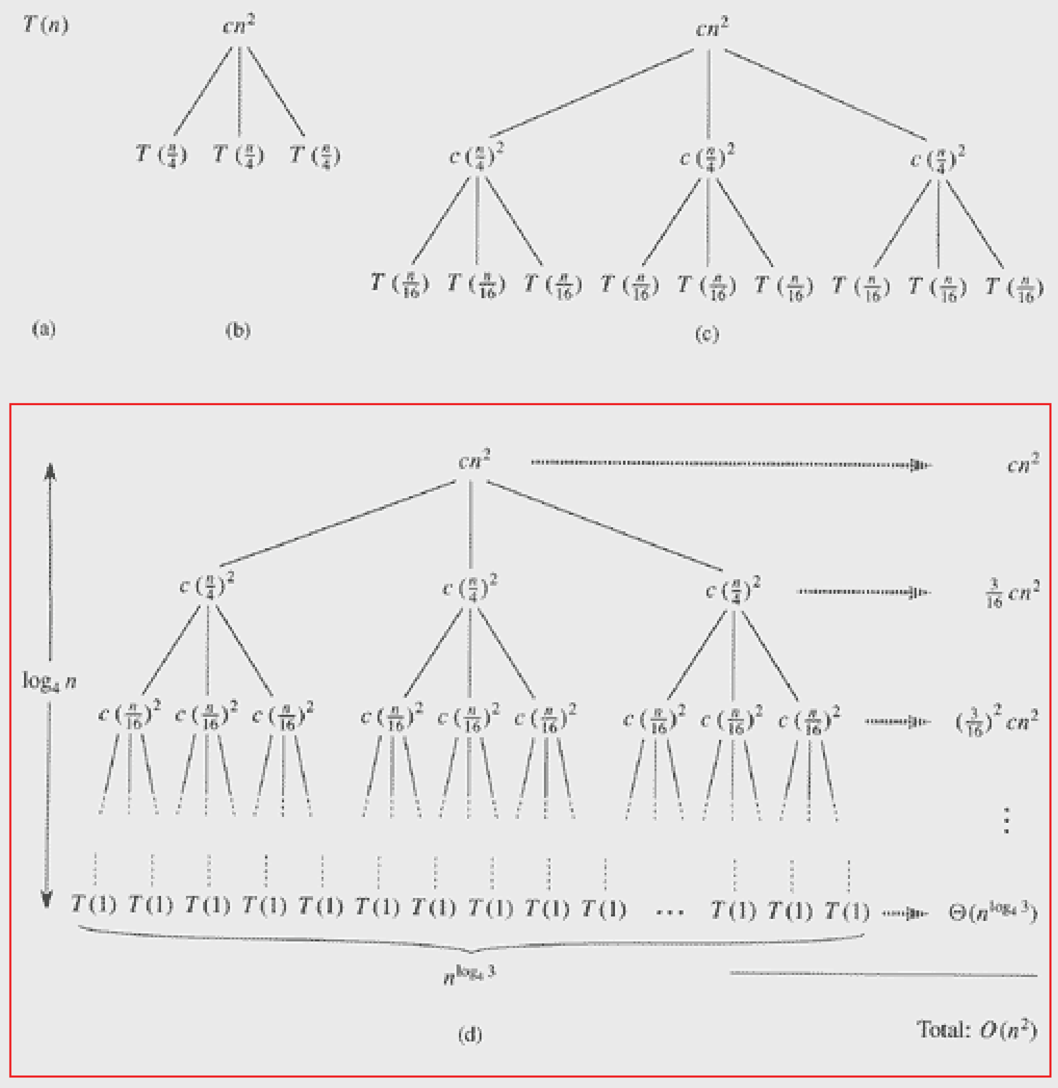


Figure 4.1 The construction of a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

4

Recursion-Tree Method (Example)

$$T(n) = 3T(n/4) + cn^2$$

$$= cn^2 + 3T(n/4)$$

$$= cn^2 + \left(\frac{3}{16}\right)cn^2 + 9T(n/16)$$

$$= cn^2 + \left(\frac{3}{16}\right)cn^2 + \left(\frac{3}{16}\right)^2 cn^2$$

$$+ \ldots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + 3^{\log_4 n} T(1)$$

$$= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

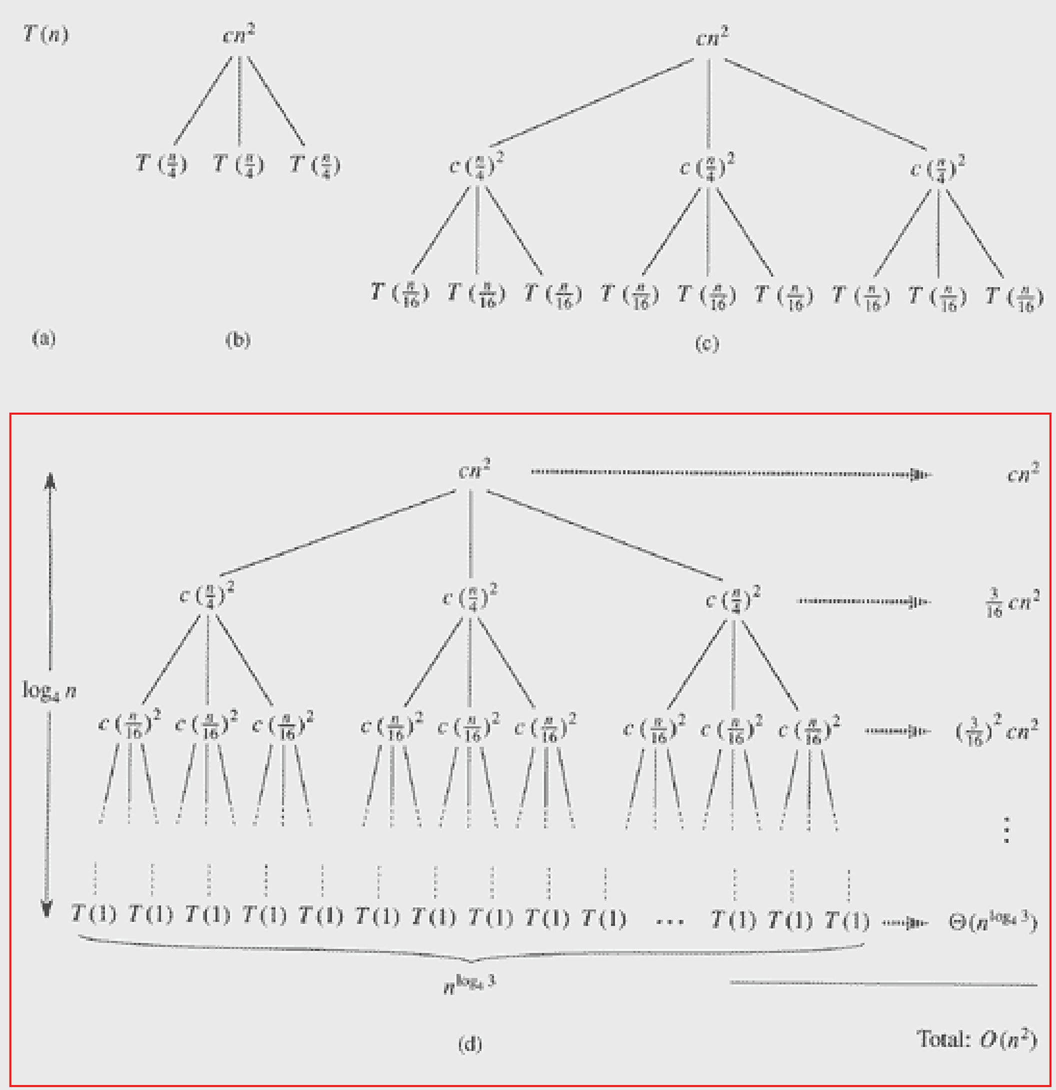


Figure 4.1 The construction of a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

4

Recursion-Tree Method (Example)

$$T(n) = 3T(n/4) + cn^2$$

$$= cn^2 + 3T(n/4)$$

$$= cn^2 + \left(\frac{3}{16}\right)cn^2 + 9T(n/16)$$

$$= cn^2 + \left(\frac{3}{16}\right)cn^2 + \left(\frac{3}{16}\right)^2 cn^2$$

$$+ \ldots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + 3^{\log_4 n} T(1)$$

$$= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

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

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

Recursion-Tree Method (Example)

$$T(n) = 3T(n/4) + cn^2$$

$$= cn^2 + 3T(n/4)$$

$$= cn^2 + \left(\frac{3}{16}\right)cn^2 + 9T(n/16)$$

$$= cn^2 + \left(\frac{3}{16}\right)cn^2 + \left(\frac{3}{16}\right)^2 cn^2$$

$$+ \ldots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + 3^{\log_4 n} T(1)$$

$$= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

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

$$< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

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

$$= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3})$$

$$\boxed{T(n) = O(n^2)}$$

Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.

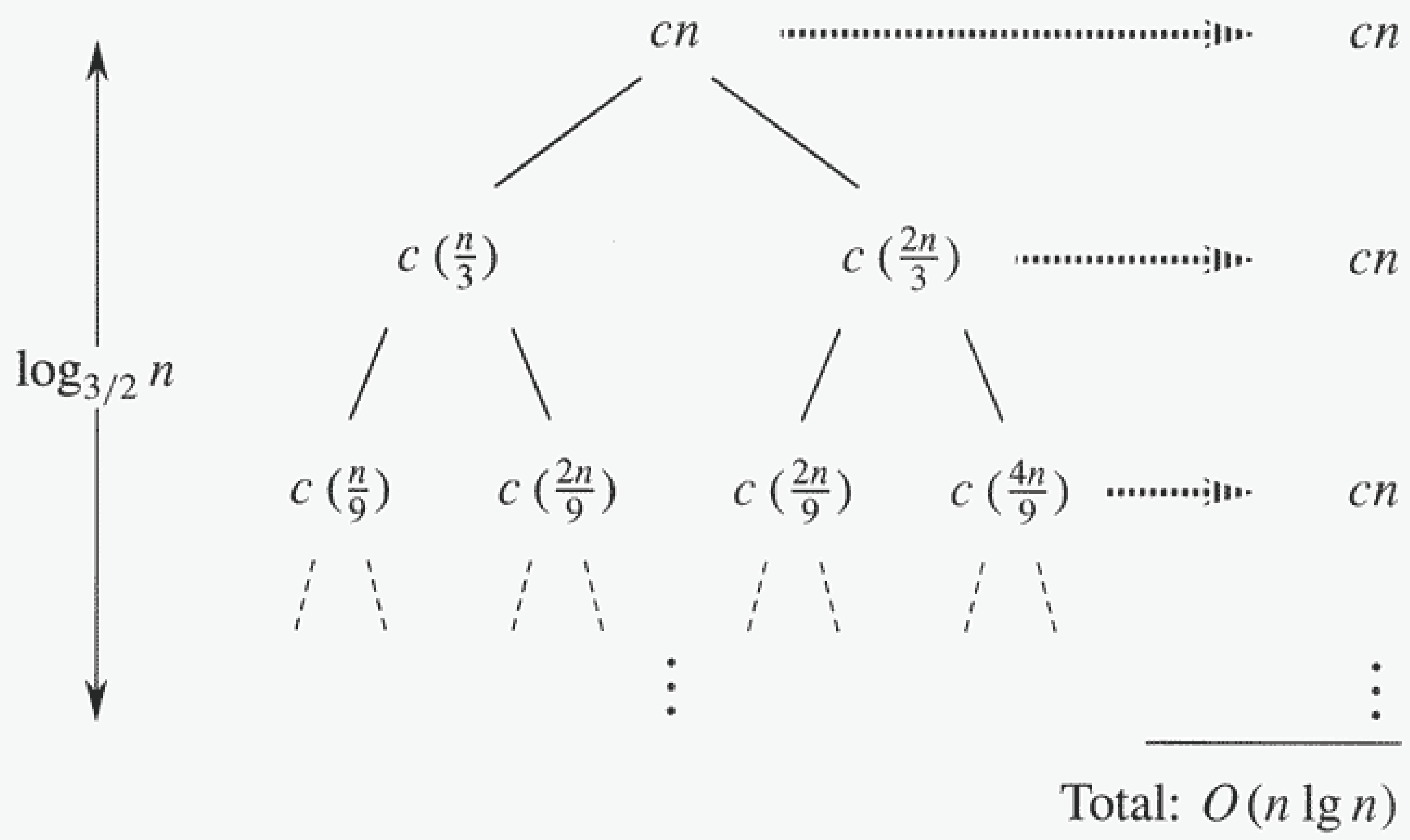


Figure 4.2 A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.

Algorithms Analysis

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

$$\boxed{\begin{array}{c} T(n) = aT(n/b) + f(n) \\ a \geq 1 \; and \; b > 1 \end{array}}$$

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

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

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^{ε}, $\varepsilon > 0$

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

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

Using the Master Method

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

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

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

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

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

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

$$\boxed{T(n) = \Theta(n^2)}$$

Using the Master Method

$$\boxed{\begin{aligned} T(n) &= aT(n/b) + f(n) \\ a \geq 1 &\; and \; b > 1 \end{aligned}}$$

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

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

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

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

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

$$\boxed{T(n) = \Theta(\lg n)}$$

Using the Master Method

$$\boxed{\begin{array}{c} T(n) = aT(n/b) + f(n) \\ a \ge 1 \text{ and } b > 1 \end{array}}$$

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

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

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

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

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

Use Case 3: $\textit{If } f(n) = \Omega(n^{\log_b a+\varepsilon})$, *and if* $af(n/b) \le cf(n)$,

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

$$\boxed{T(n) = \Theta(n\lg n)}$$

Using the Master Method

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

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

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

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 $n^{\varepsilon}, \varepsilon > 0$

$\boxed{\textbf{Master Method doesn't Apply}}$

Extended Master Method

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

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

Back to the previous recurrence:

$$T(n) = 2T(n/2) + n \lg n$$
$$f(n) = \Theta(n^{\log_b a} \lg n), \; and \;\; k = 1$$

$$\boxed{T(n) = \Theta(n \lg^2 n)}$$

Proof of Master Theorem (Lemma 4.2)

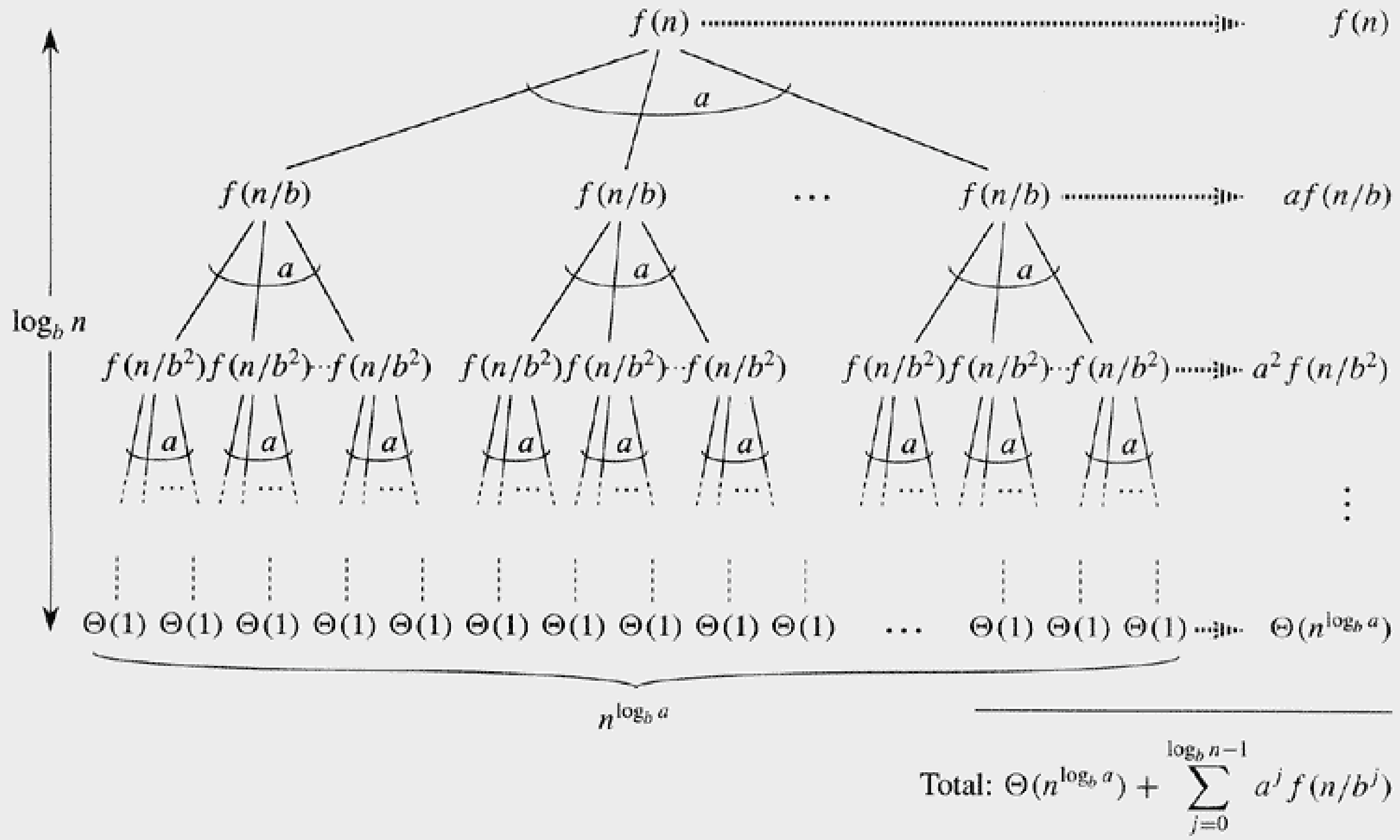


Figure 4.3 The recursion tree generated by $T(n) = aT(n/b) + f(n)$. The tree is a complete a-ary tree with $n^{\log_b a}$ leaves and height $\log_b n$. The cost of each level is shown at the right, and their sum is given in equation (4.6).

4.2 The closest pair problem

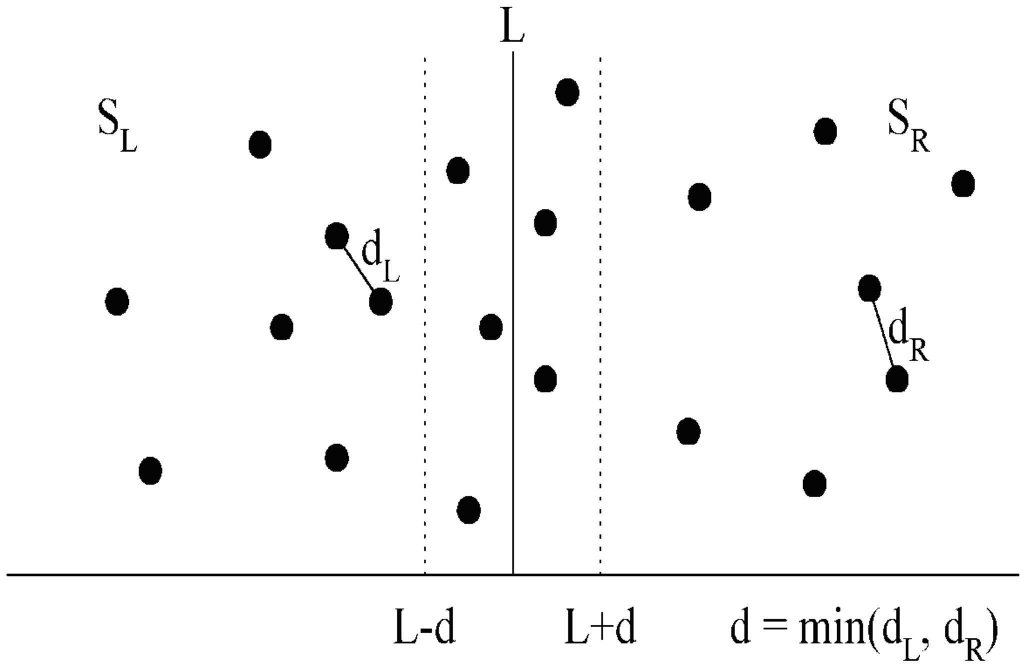
- Given a set S of n points, find a pair of points which are closest together.
- 1-D version :

  Solved by sorting

  Time complexity :

  O(n log n)

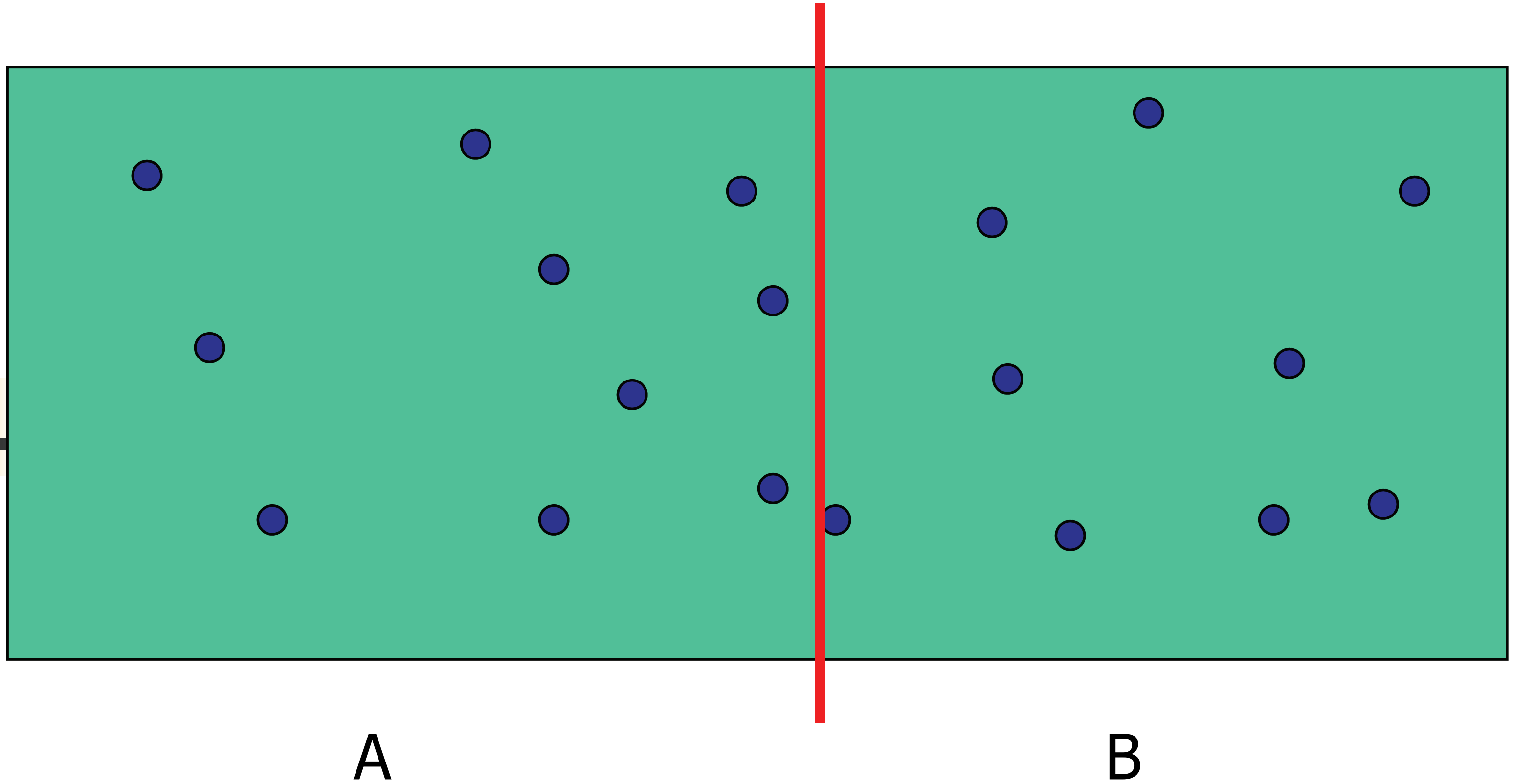
- 2-D version



Divide-And-Conquer Solution

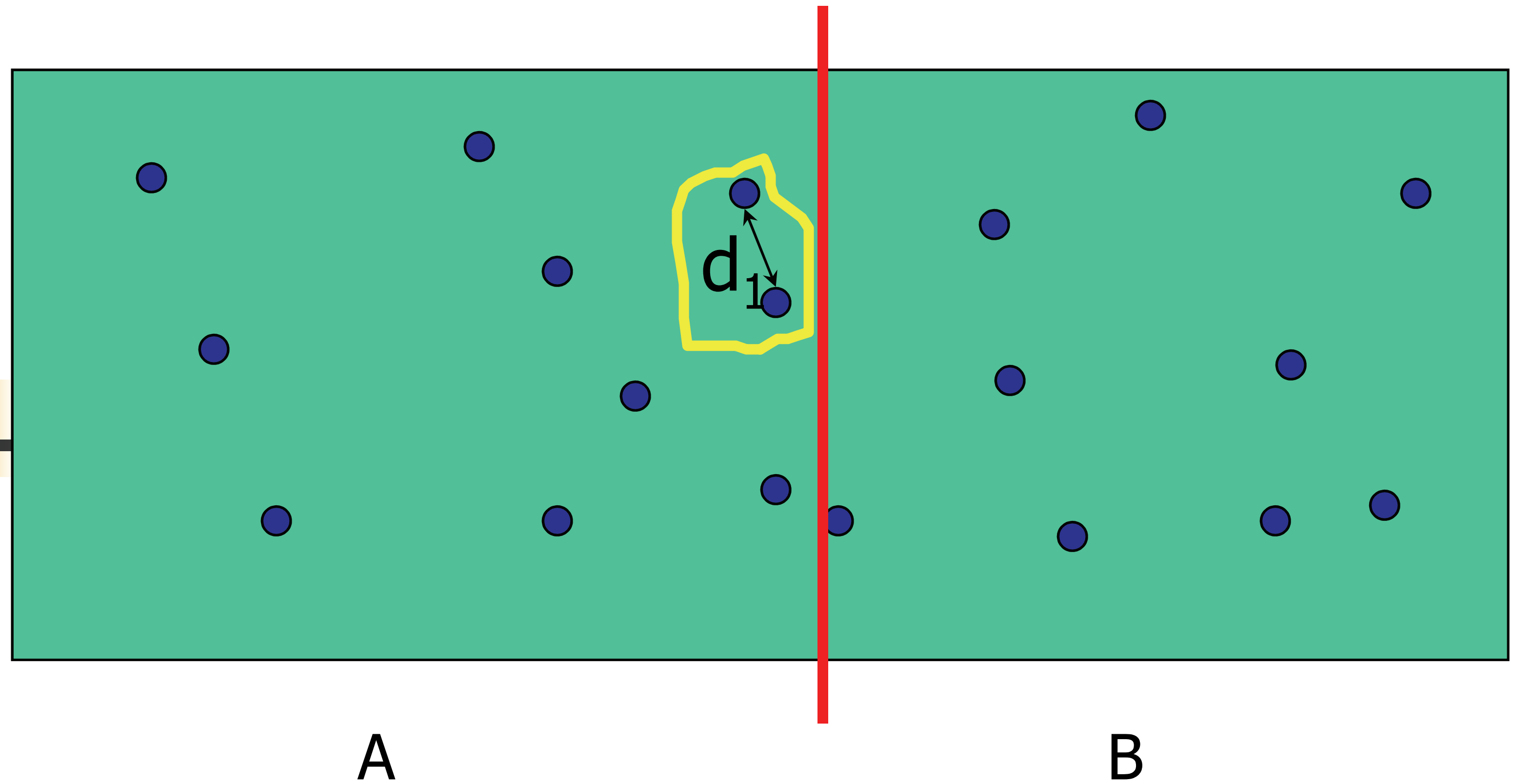
- When n is small, use simple solution.
- When n is large
  - Divide the point set into two roughly equal parts A and B.
  - Determine the closest pair of points in A.
  - Determine the closest pair of points in B.
  - Determine the closest pair of points such that one point is in A and the other in B.
  - From the three closest pairs computed, select the one with least distance.

Example



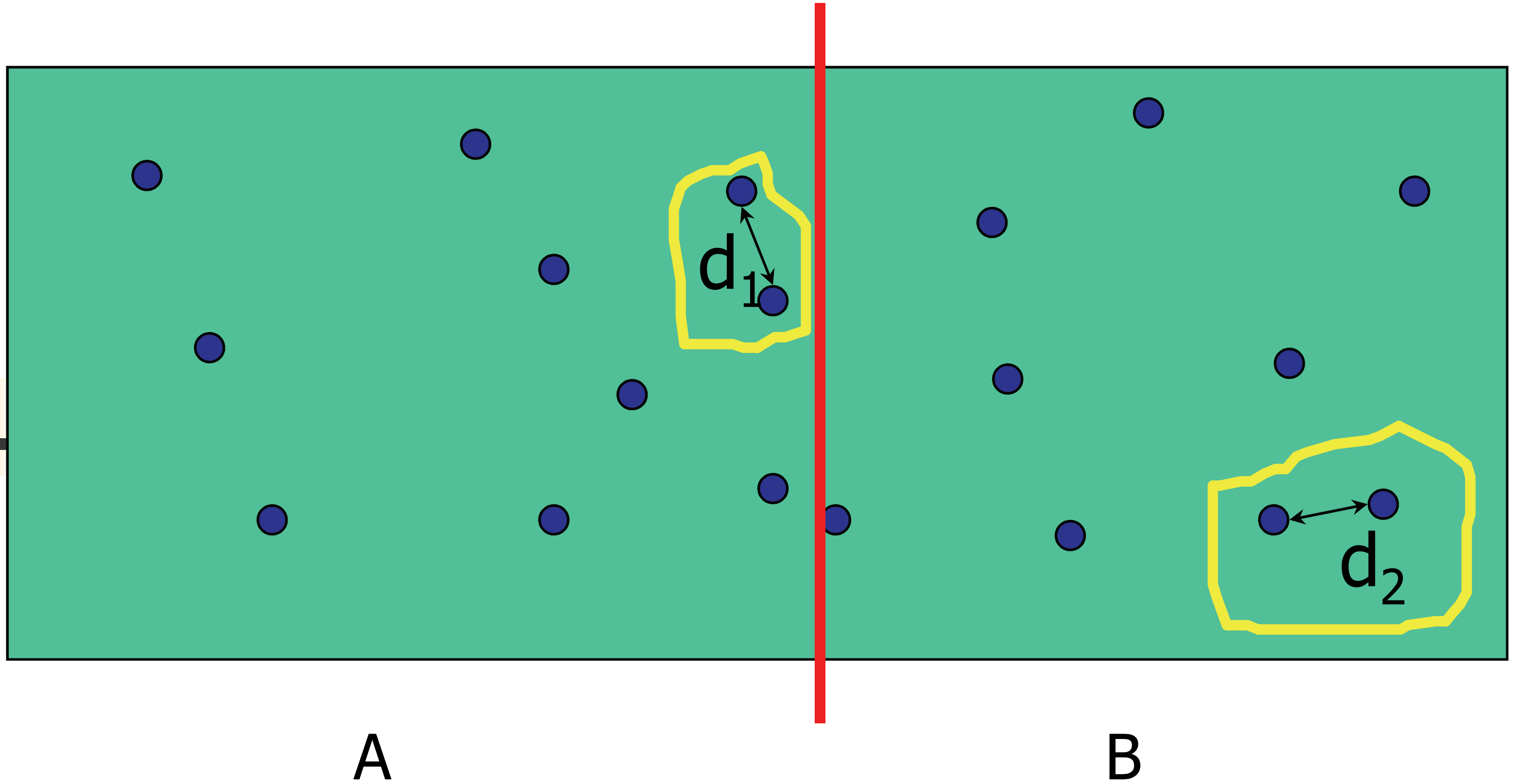
- Divide so that points in A have x-coordinate <= that of points in B.

Example



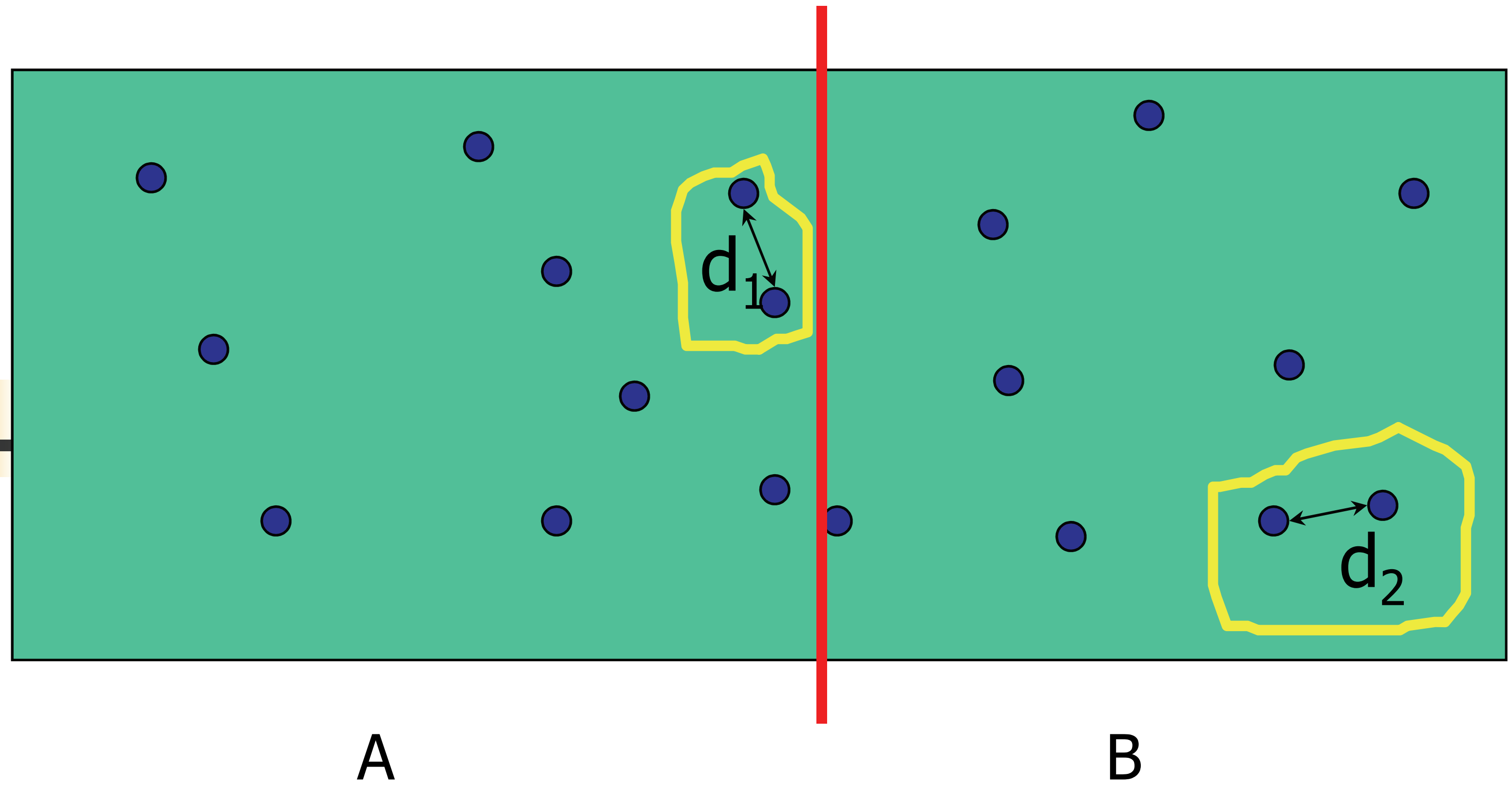
- Find closest pair in A.
- Let d_1 be the distance between the points in this pair.

Example



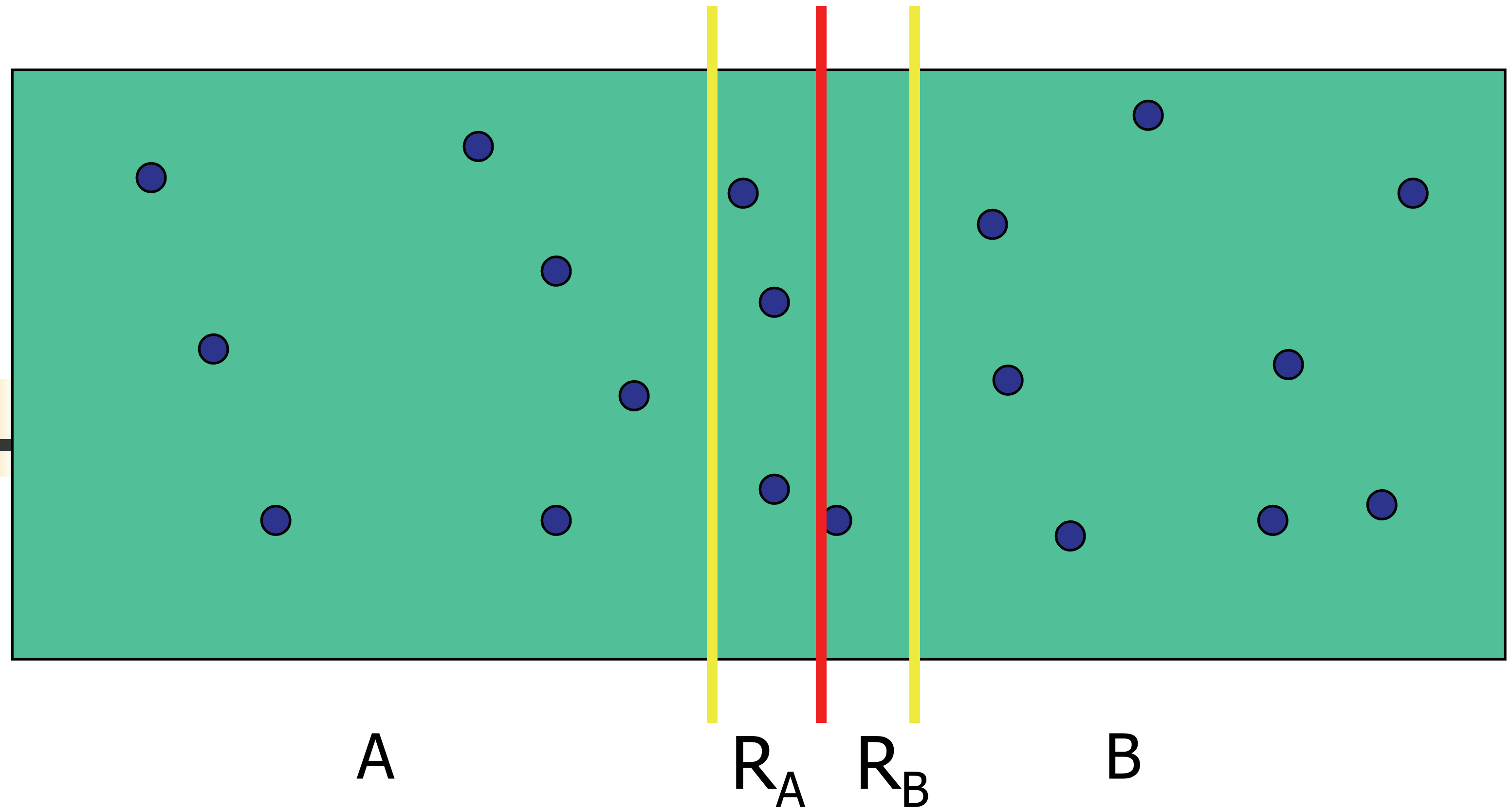
- Find closest pair in B.
- Let d_2 be the distance between the points in this pair.

Example



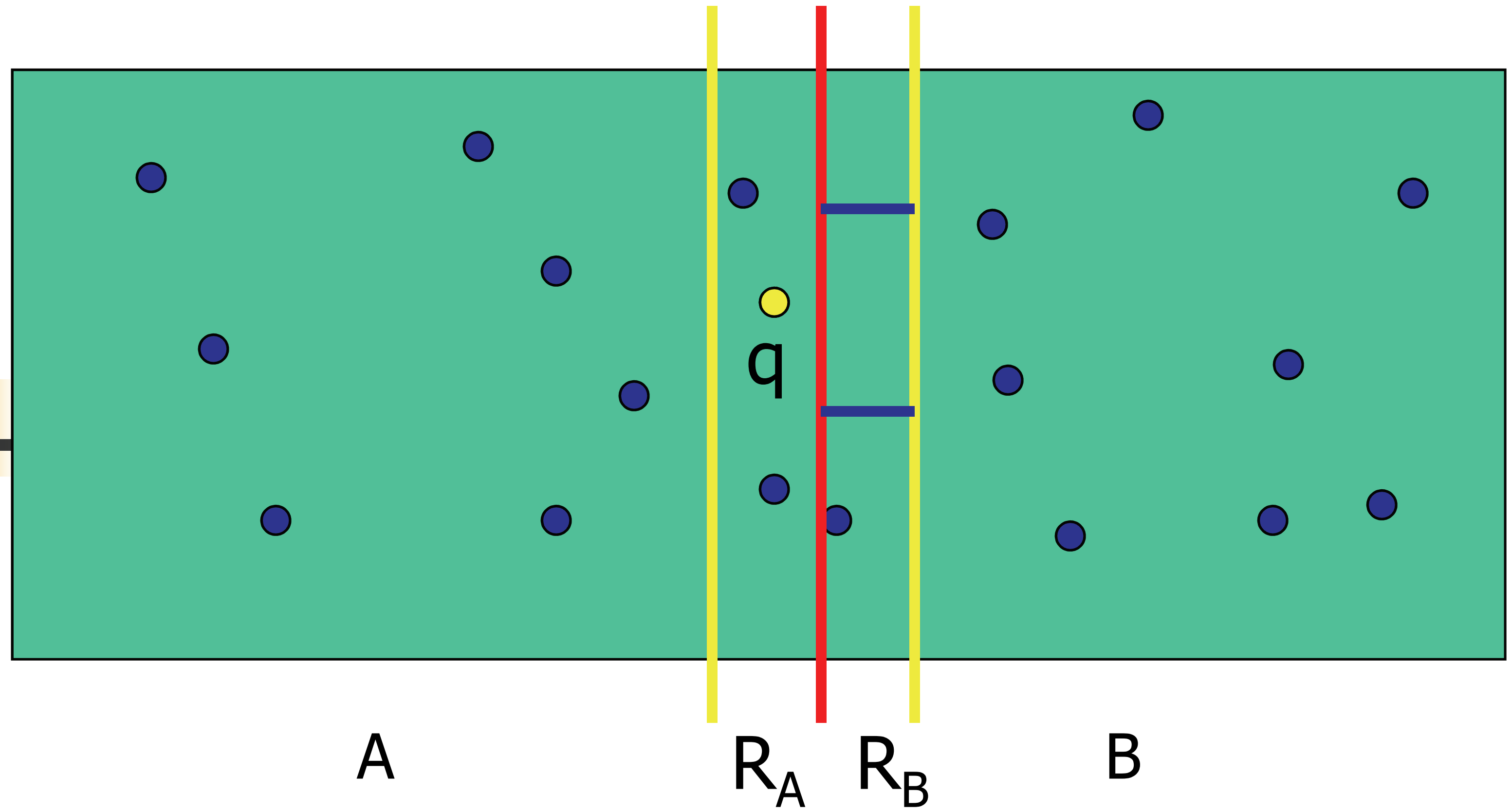
- Let $d = \min\{d_1, d_2\}$.
- Is there a pair with one point in A, the other in B and distance $< d$?

Example



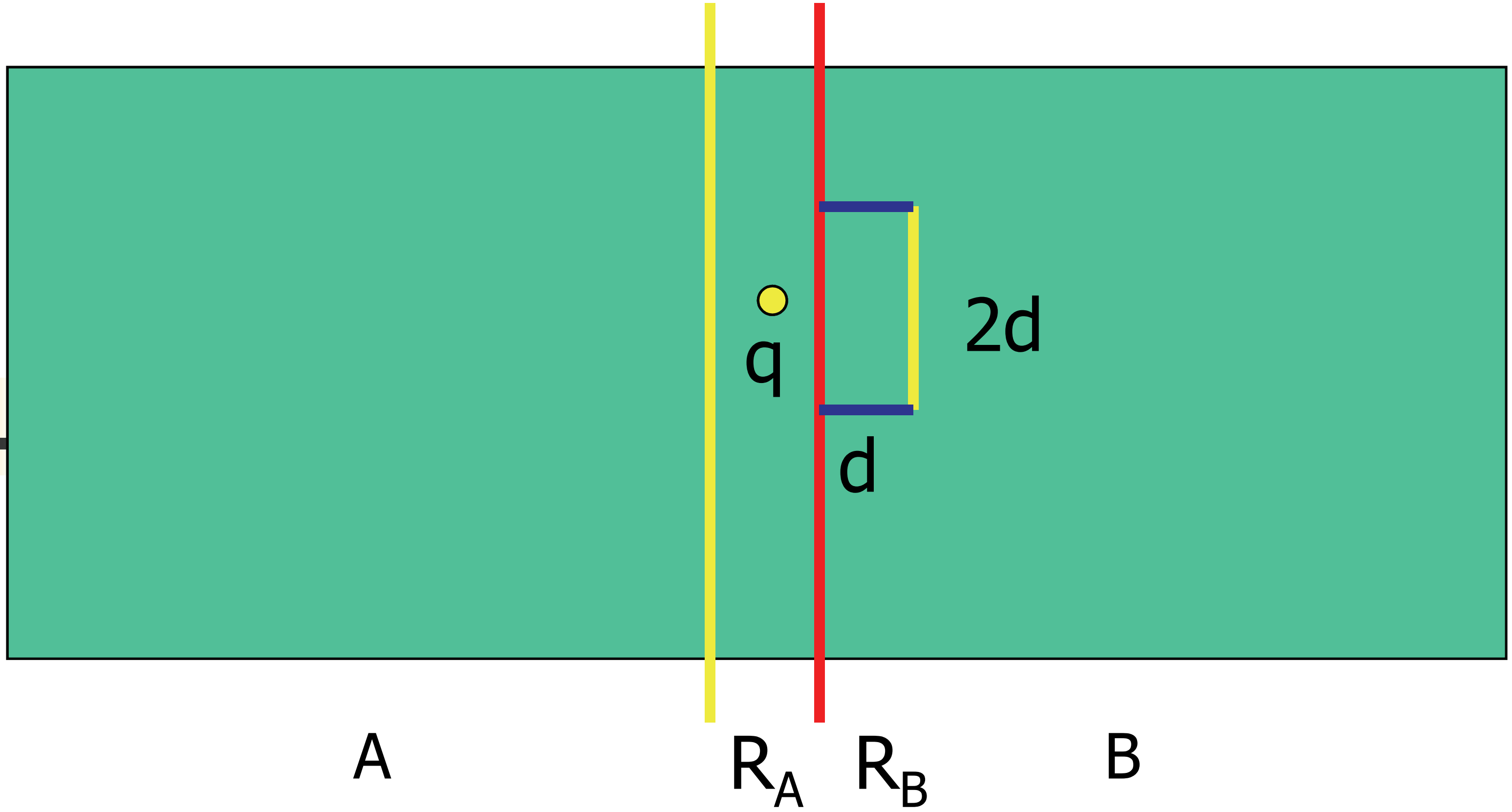
- Candidates lie within d of the dividing line.
- Call these regions R_A and R_B, respectively.

Example



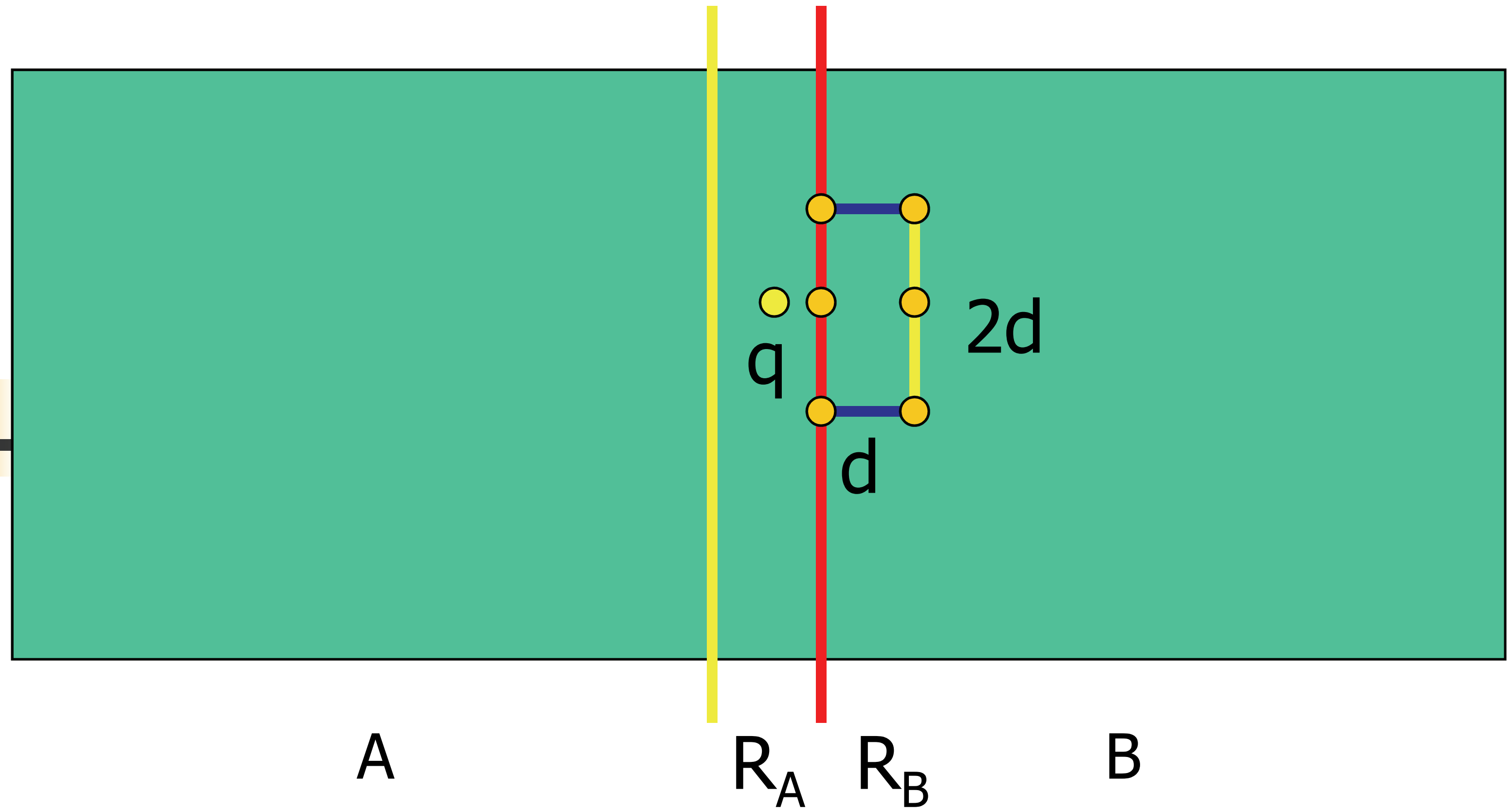
- Let q be a point in R_A.
- q need be paired only with those points in R_B that are within d of q.y.

Example



- Points that are to be paired with q are in a d x 2d rectangle of R_B (comparing region of q).
- Points in this rectangle are at least d apart.

Example



- So the comparing region of q has at most 6 points.
- So number of pairs to check is $<= 6|R_A| = O(n)$.

D&C

- We first partition the set S into S_L and S_R such that every point in S_L lies to the left of every point in 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.

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 $\mathbf{n^2/4}$ 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 $0(n)$ time. (how?)**

Merge Step

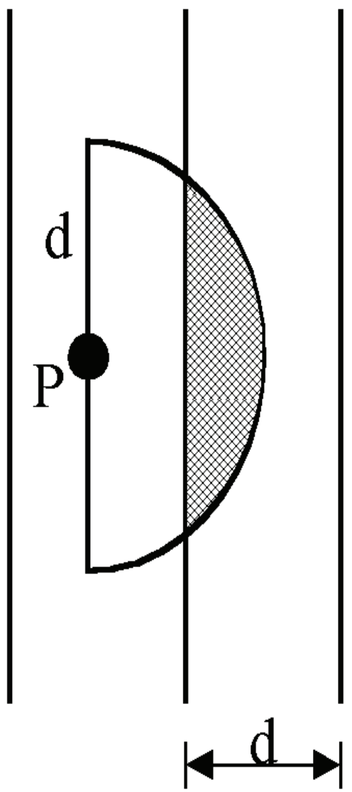
- If a point P in S_L and a point Q in S_R constitute a closest pair, the distance between P and Q must be less than d. Hence we do not have to consider a point too far away from P. Consider Figure 4-5.
- We only have to examine the shaded area in Figure 4-5. If P is exactly on line L, this shaded area will be the largest.
- Even in this case, this shaded area will be contained in the $d \times 2d$ rectangle A as shown in Figure 5-6.
- Thus we only have to examine points within this rectangle A.

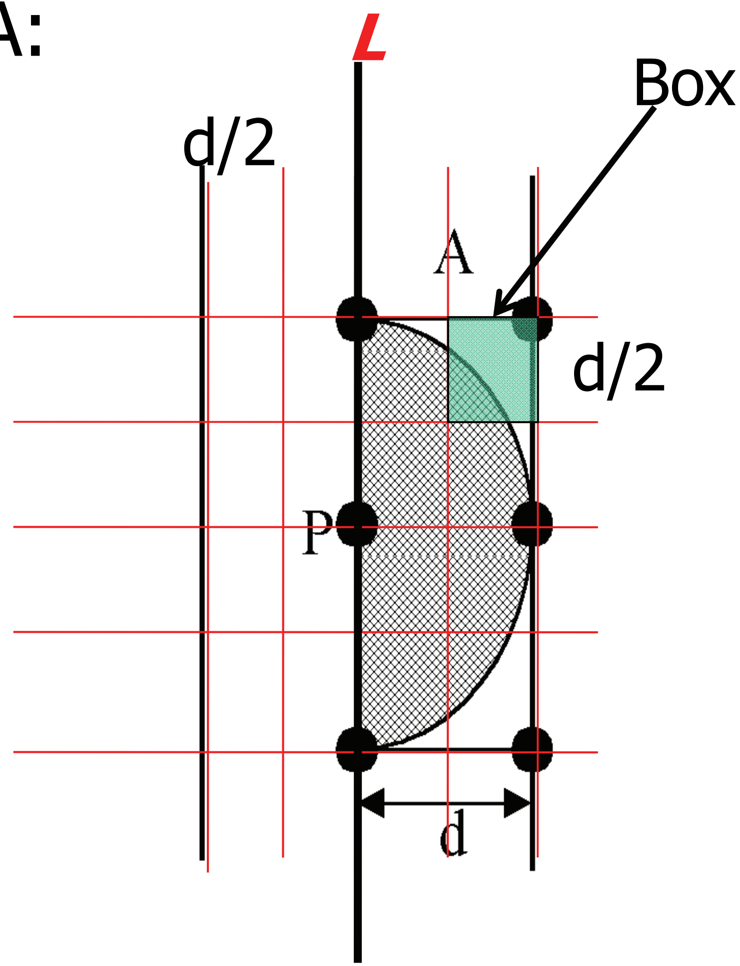
Merge Step

- 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 six such points as discussed above **(why?).**
- O(n) Step

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

- at most 6 points in area A:





One box contains one point.
If s,s′∈ S have the property that d(s,s′)<d, then s and s′ are within 15 positions of each other in the sorted list Sy.

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 points, return infinity(∞) 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 by 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)

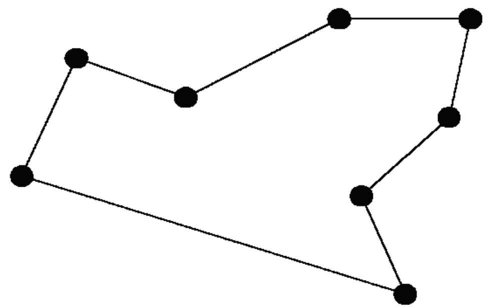
Steps 2~5:

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

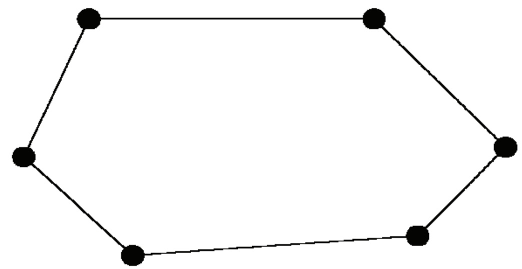
$\Rightarrow T(n) = O(n \log n)$

4.3 The convex hull problem

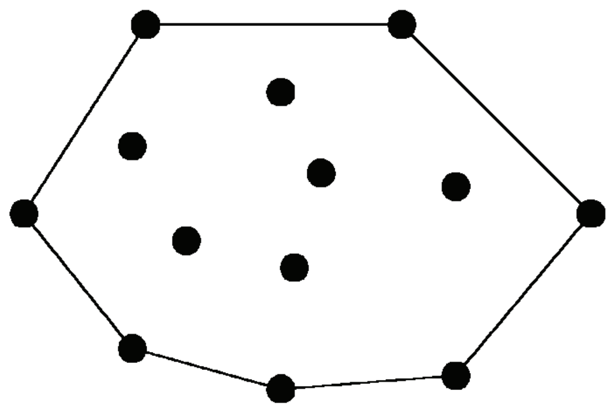
concave polygon:



convex polygon:

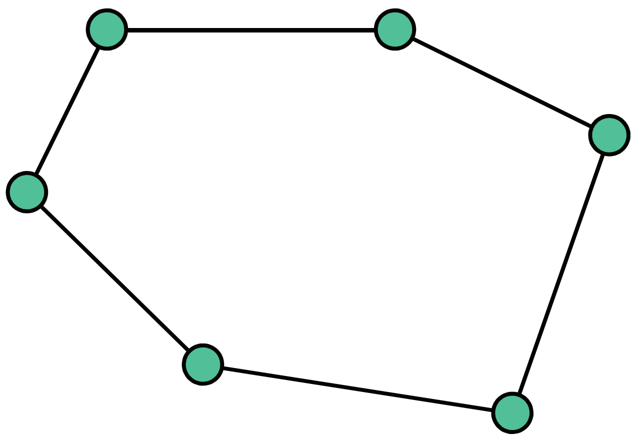


- The convex hull of a set of planar points is the smallest convex polygon containing all of the points.

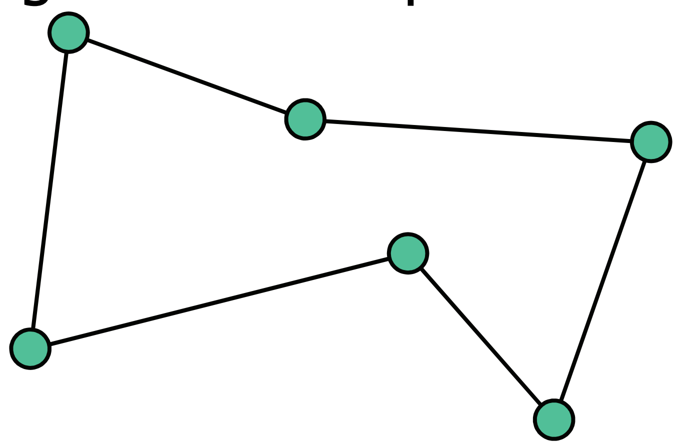


Convex Polygon

- A convex polygon is a nonintersecting polygon whose internal angles are all convex (i.e., less than π)
- 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
- Think of a rubber band snapping around the points



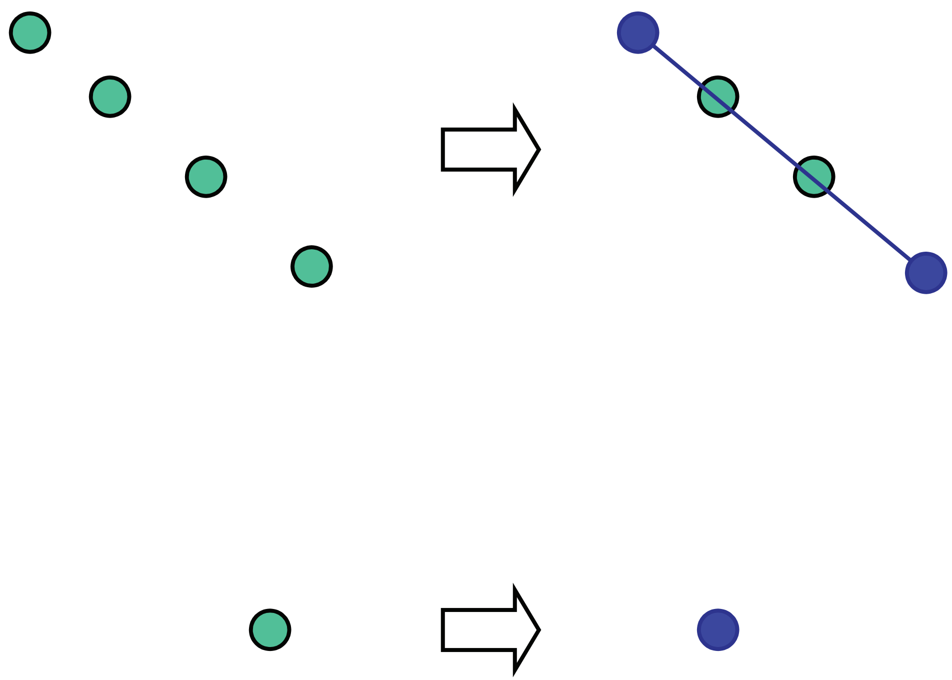
convex



nonconvex

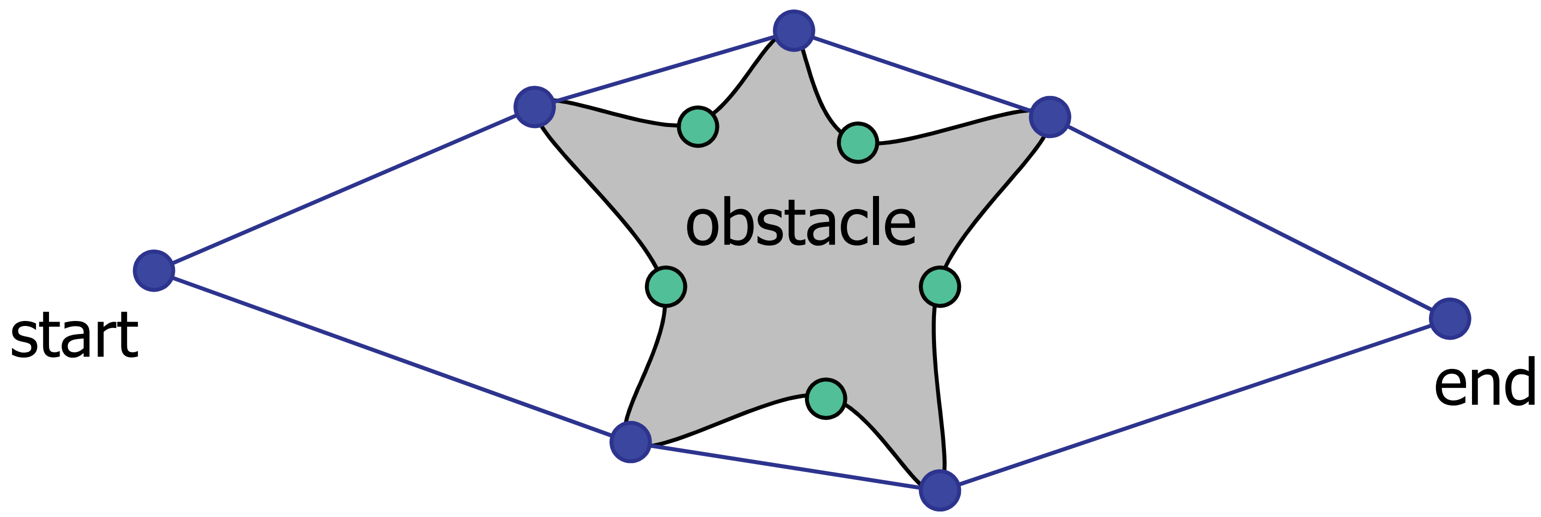
Special Cases

- 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



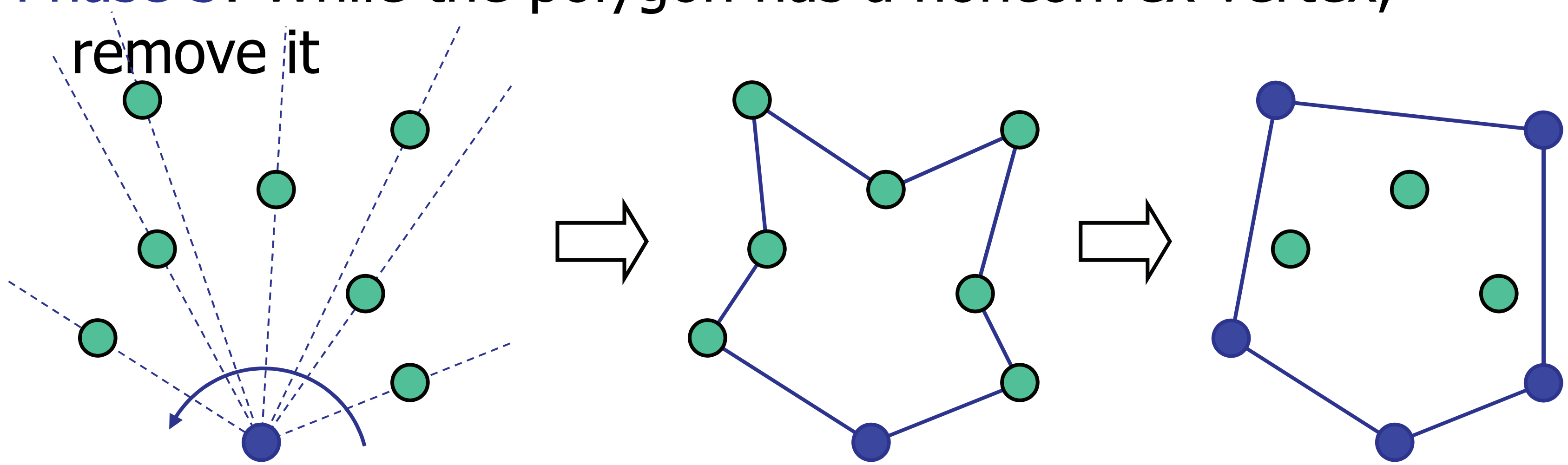
Applications

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



Computing the Convex Hull

- The following method computes the convex hull of a set of points
  - Phase 1: Find the lowest point (anchor point)
  - 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



- *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.*

we show a triplet (p_1, p_2, p_3) of points such that $x_1 < x_2 < x_3$. Clearly, this triplet makes a left turn if the slope of segment p_2p_3 is greater than the slope of segment p_1p_2. This is expressed by the following question:

$$\text{Is } \frac{y_3 - y_2}{x_3 - x_2} > \frac{y_2 - y_1}{x_2 - x_1} \ ?$$

By the expansion of $\Delta(p_1, p_2, p_3)$ shown in 12.1, we can verify that inequality 12.2 is equivalent to $\Delta(p_1, p_2, p_3) > 0$.

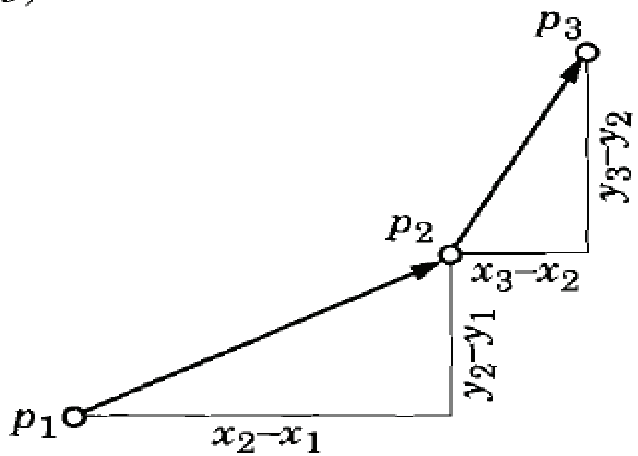
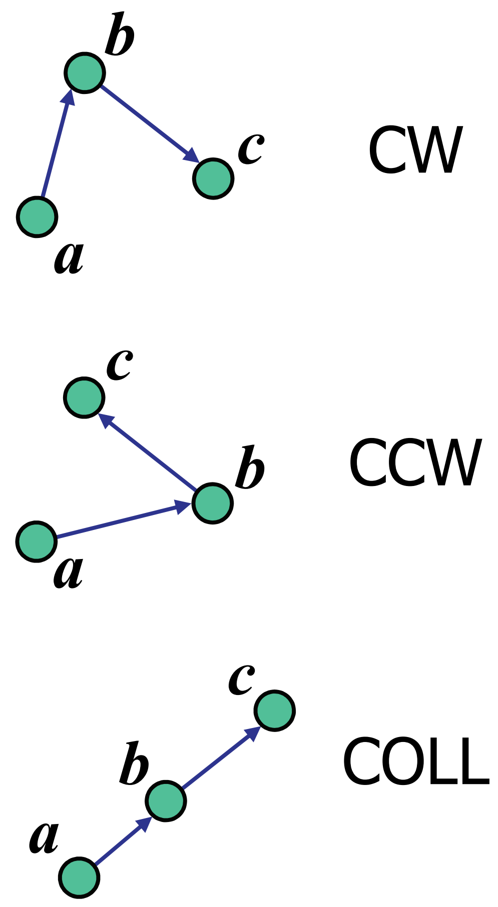


Figure An example of a left turn. The differences between the coordinates between p_1 and p_2 and the coordinates of p_2 and p_3 are also illustrated.

Orientation

- The orientation of three points in the plane is clockwise, counterclockwise, or collinear
- ***orientation***$(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$
  - 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 $\Delta(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$, whose absolute value is twice the area of the triangle with vertices $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$

$$\Delta(a,b,c) = \begin{vmatrix} x_a & y_a & 1 \\ x_b & y_b & 1 \\ x_c & y_c & 1 \end{vmatrix}$$



Theorem 12.11: *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.*

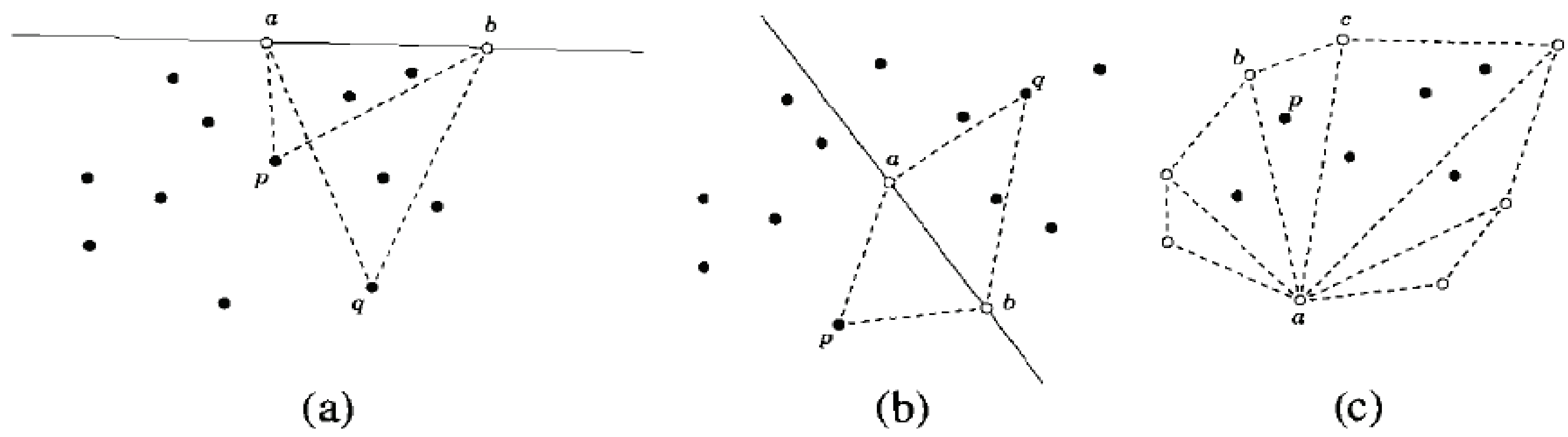
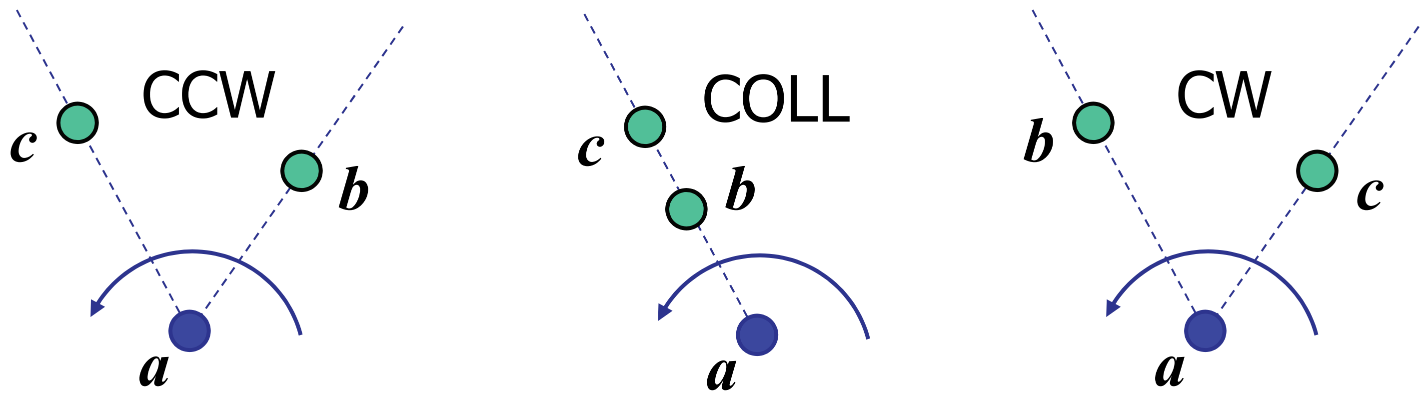


Figure 12.22: Illustration of the properties of the convex hull given in Theorem 12.11: (a) points a and b form an edge of the convex hull; (b) points a and b do not form an edge of the convex hull; (c) point p is not on the convex hull.

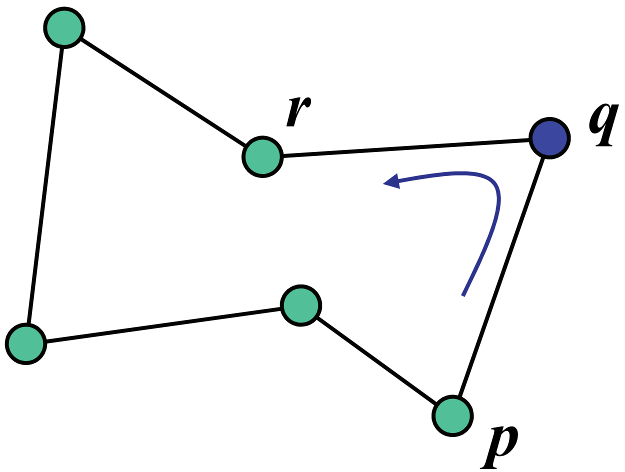
Sorting by Angle

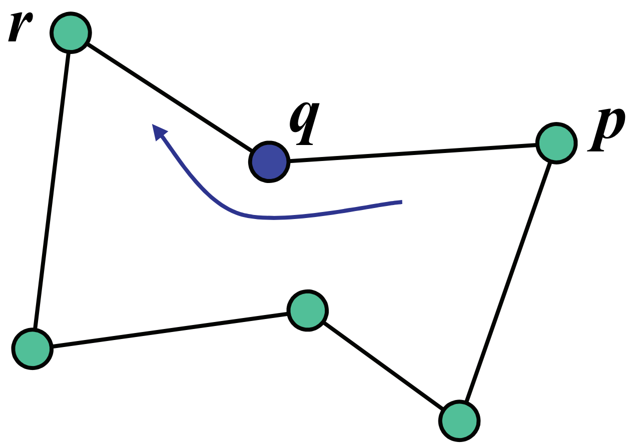
- Computing angles from coordinates is complex and leads to numerical inaccuracy
- We can sort a set of points by angle with respect to the anchor point a using a comparator based on the orientation function
  - $b < c \Leftrightarrow \textit{orientation}(a, b, c) = \text{CCW}$
  - $b = c \Leftrightarrow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



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 $\Leftrightarrow$ ***orientation***$(p, q, r) =$ CCW
  - q nonconvex $\Leftrightarrow$ ***orientation***$(p, q, r) =$ CW or COLL



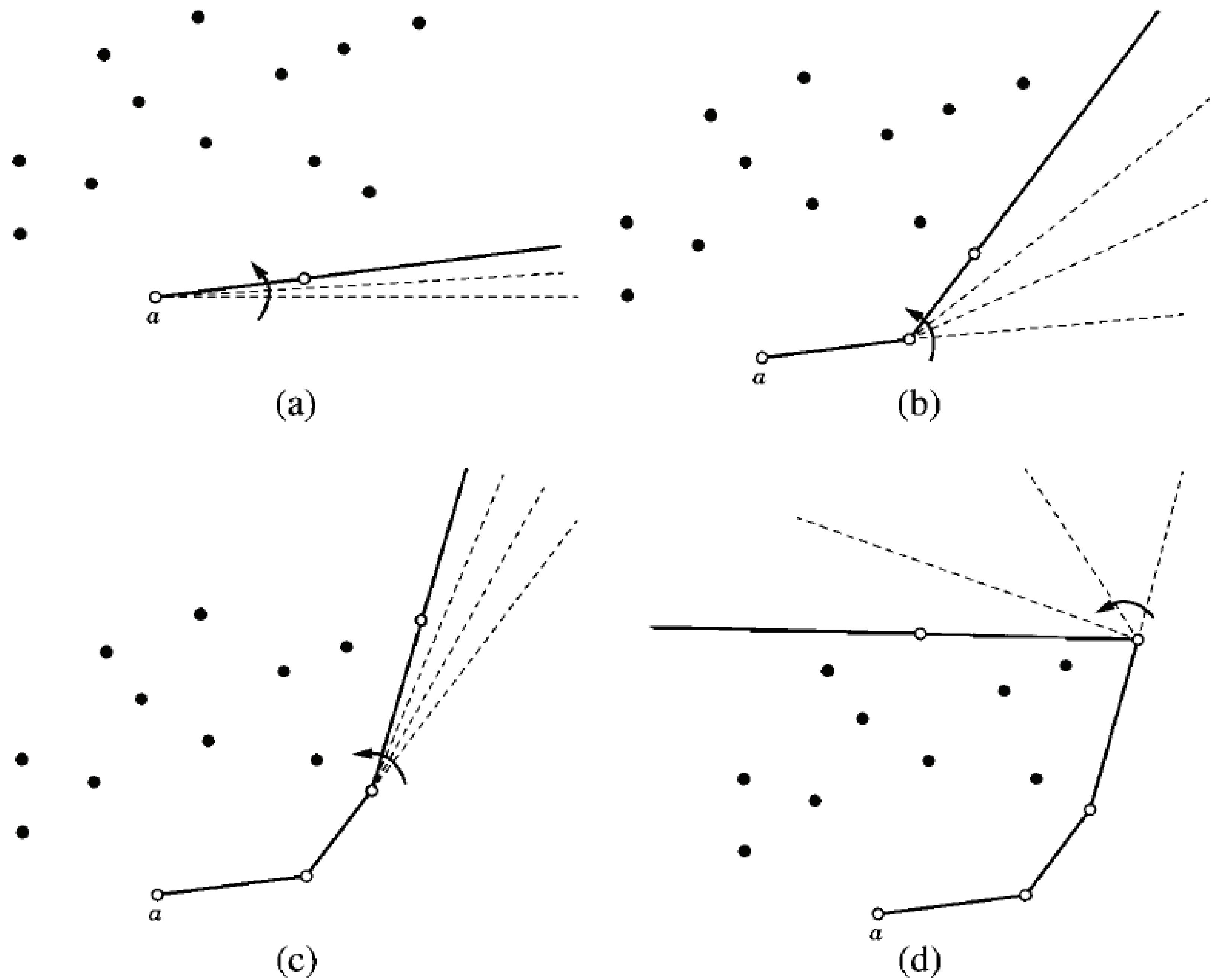


Gift Wrapping Algorithm

- 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.

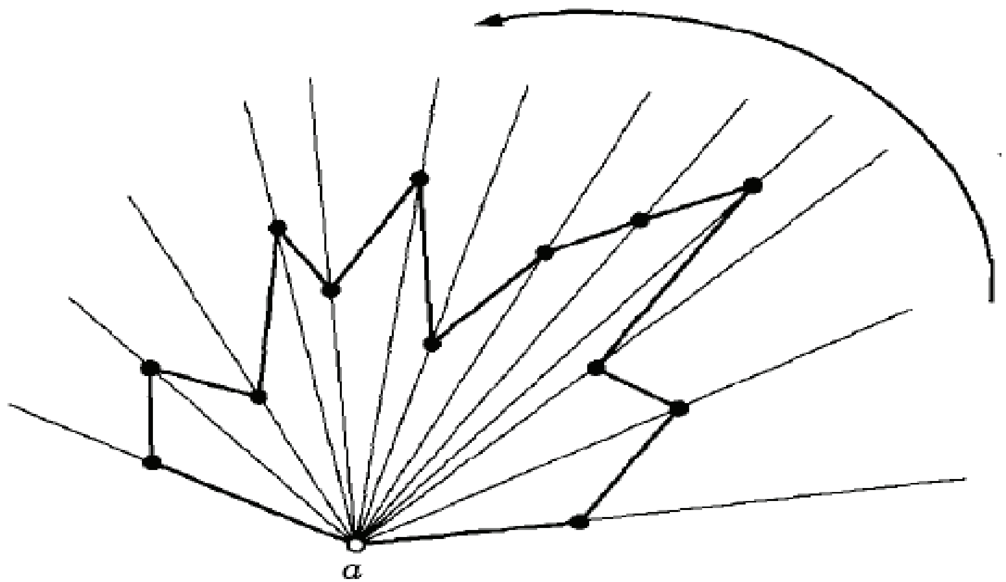
Gift Wrapping

- 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
(a)
a
(b)
a
(c)
a
(d)

Graham Scan Algorithm

1. We find a point a of P that is a vertex of H and call it the ***anchor point***. We can, for example, pick as our anchor point a the point in P with minimum y-coordinate (and minimum x-coordinate if there are ties).
2. We sort the remaining points of P (that is, $P - \{a\}$) using the radial comparator $C(a)$, and let S be the resulting sorted list of points. (See Figure 12.24.) In the list S, the points of P appear sorted counterclockwise "by angle" with respect to the anchor point a, although no explicit computation of angles is performed by the comparator.



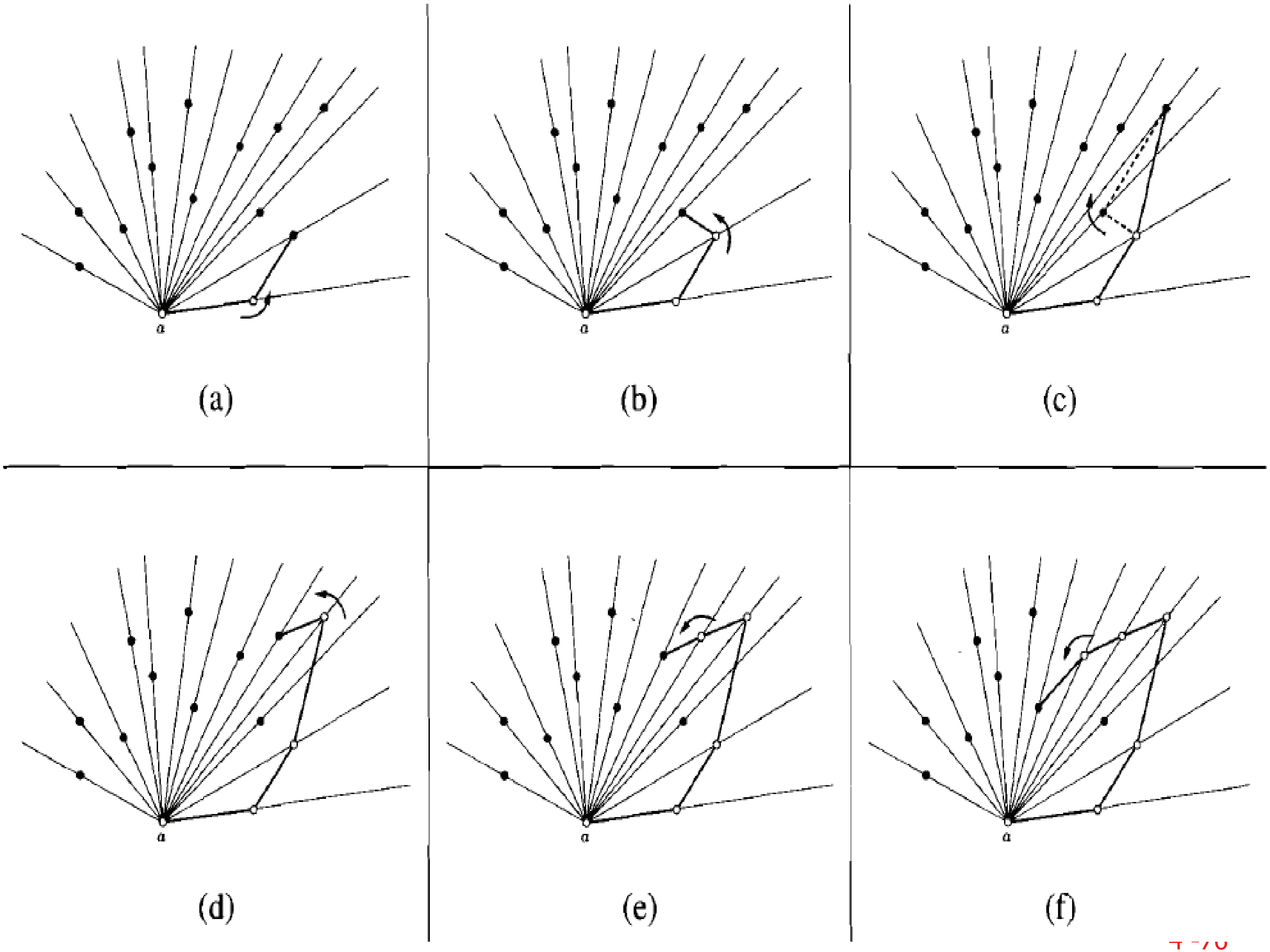
Graham Scan

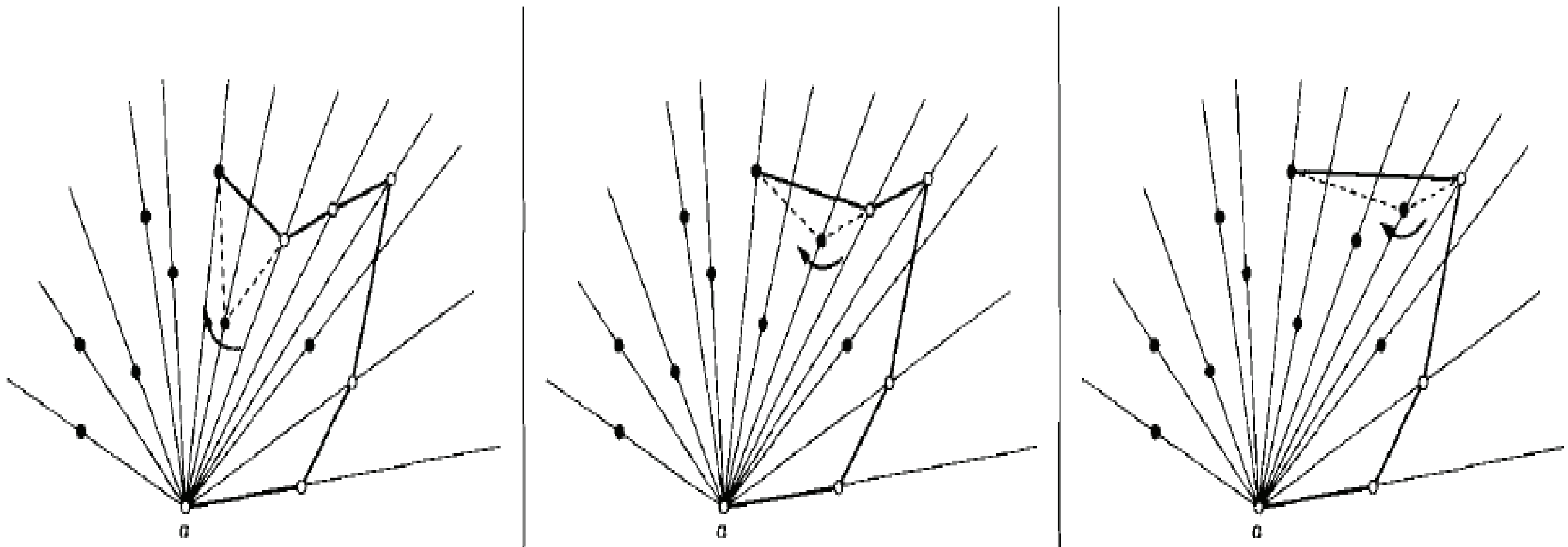
3. After adding the anchor point a at the first and last position of S, we ***scan*** through the points in S in (radial) order, maintaining at each step a list H storing a convex chain "surrounding" the points scanned so far. Each time we consider new point p, we perform the following test:

   (a) If p forms a left turn with the last two points in H, or if H contains fewer than two points, then add p to the end of H.

   (b) Otherwise, remove the last point in H and repeat the test for p.

We stop when we return to the anchor point a, at which point H stores the vertices of the convex hull of P in counterclockwise order.


a
(a)
(b)
(c)
(d)
(e)
(f)

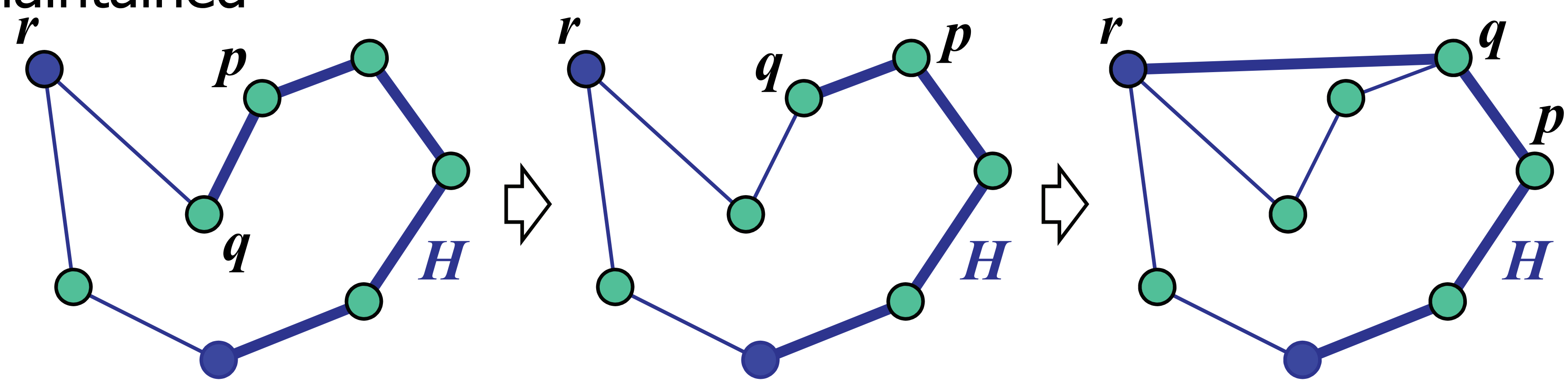


(g) (h) (i)

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

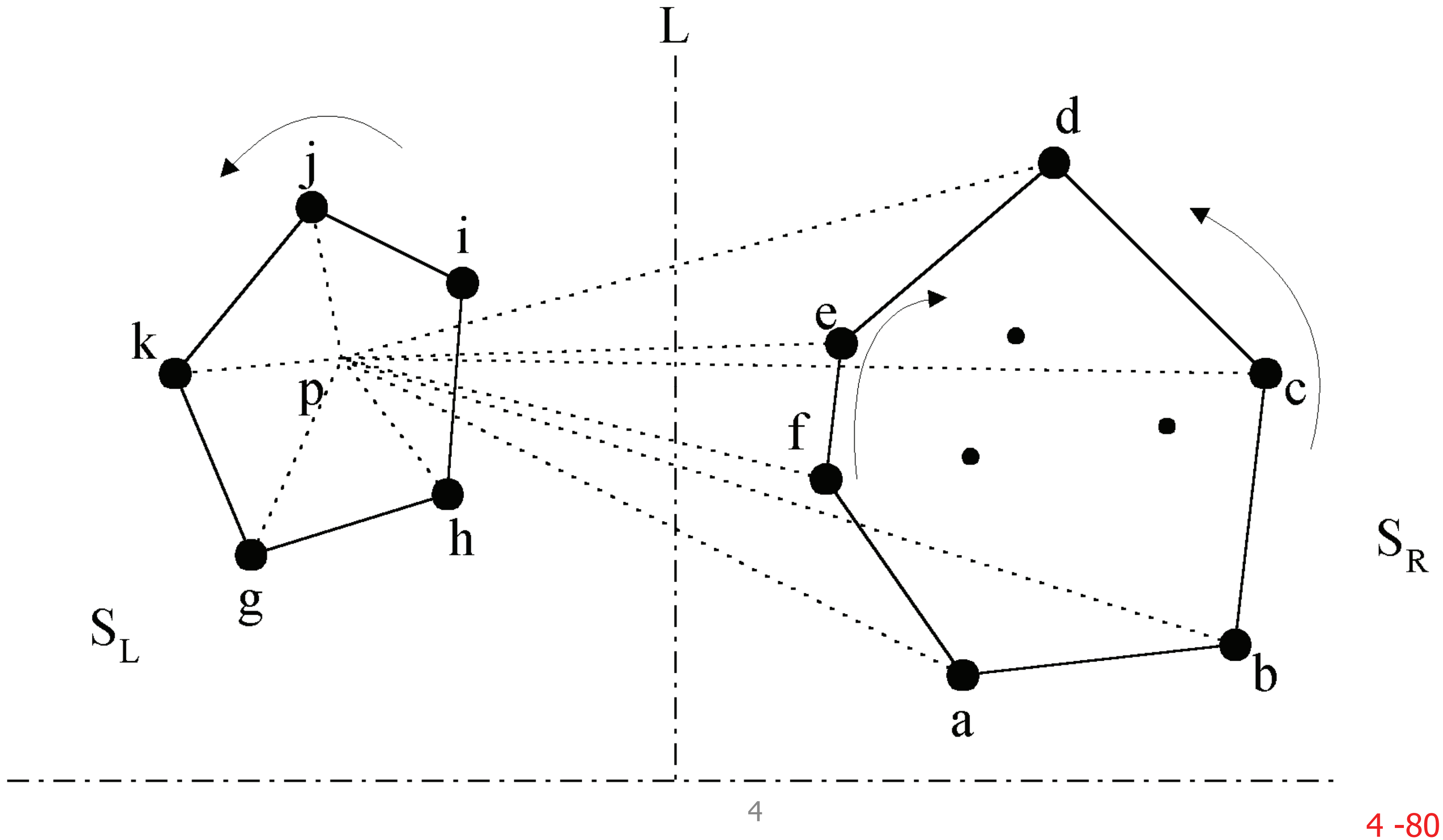
```
for each vertex r of the polygon
    Let q and p be the last and second last
        vertex of H
    while orientation(p, q, r) = CW or COLL
        remove q from H
        q ← p
        p ← vertex preceding p in H
    Add r to the end of H
```



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

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

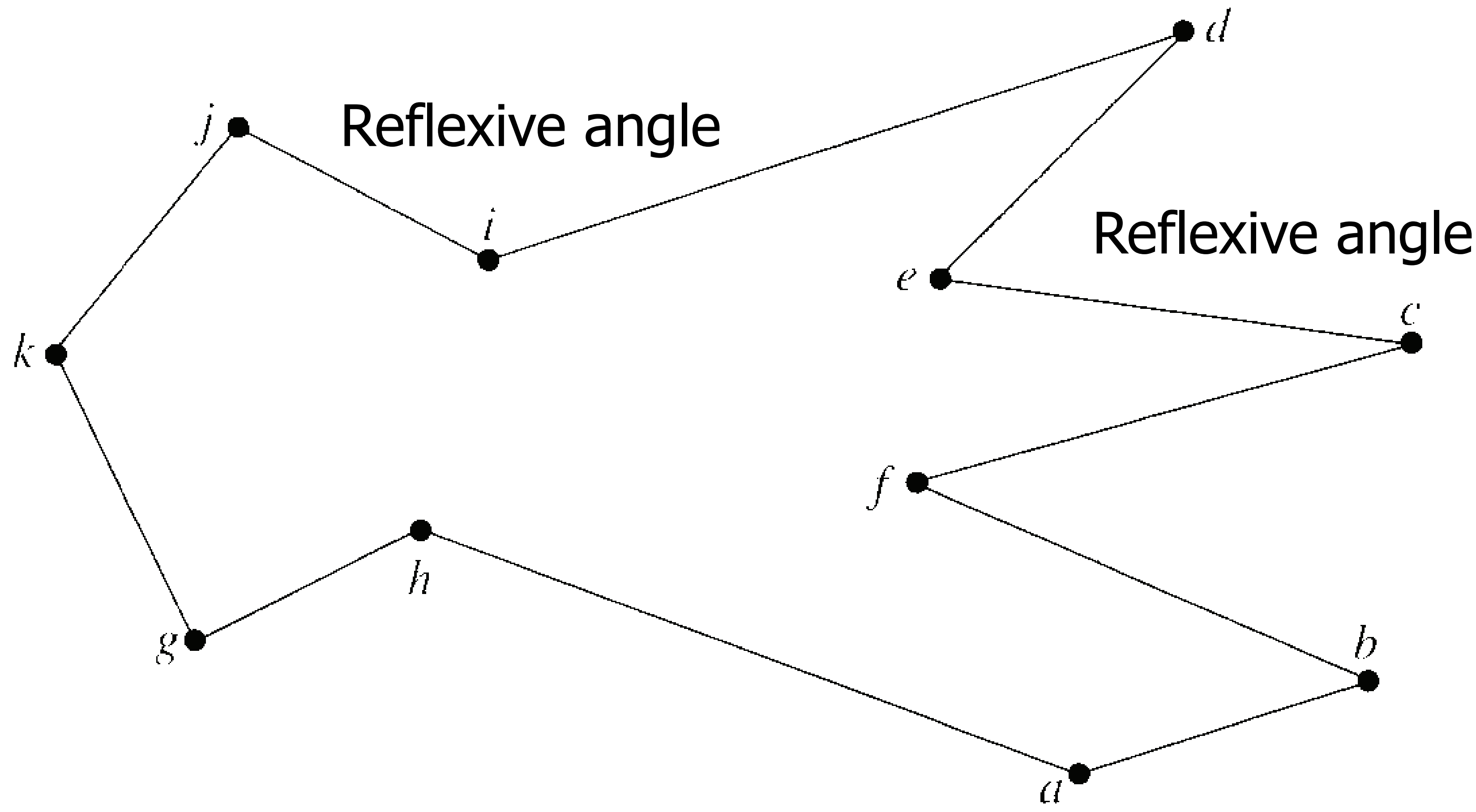


convex hull problem

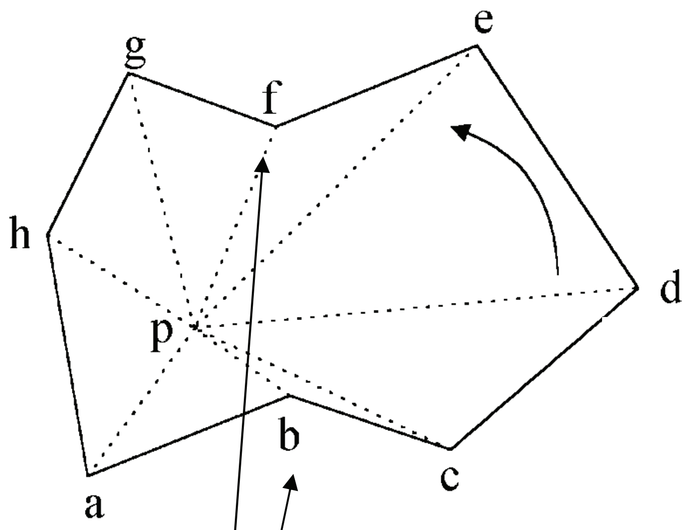
- To find a convex hull, we may use the divide-and-conquer.
- The set of planar points is divided into two subsets S_L and S_R by a line perpendicular to the x-axis.
- Convex hulls for S_L and S_R are now constructed and they are denoted as Hull(S_l),*Hull(S_r)* respectively.
- To combine Hull(S*L*) and Hull(S***R***) into one convey 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, as illustrated in Figure 4-10.

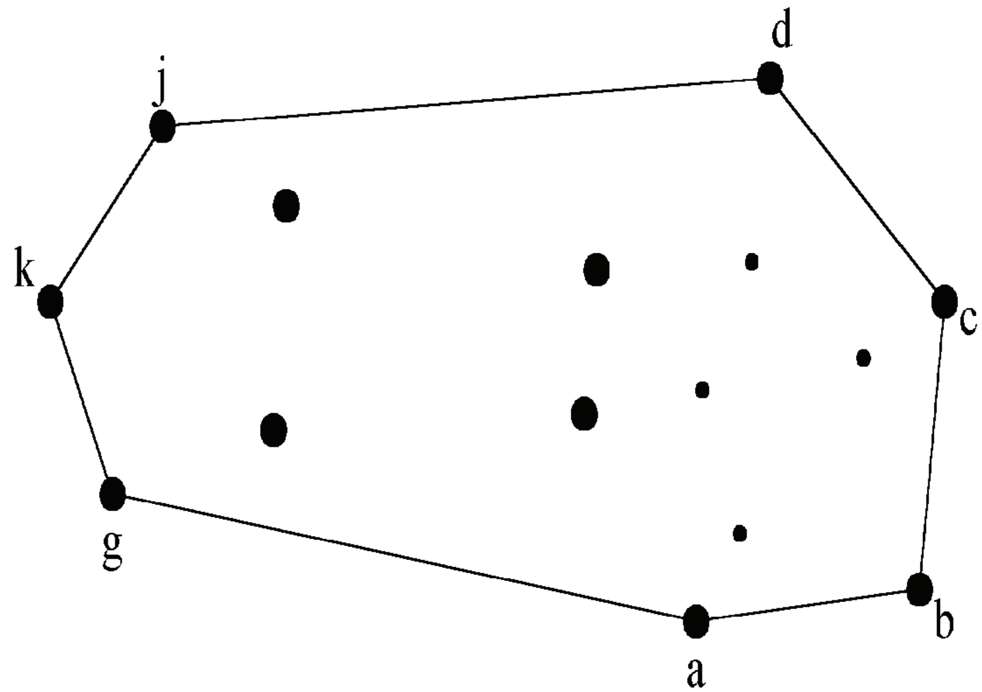

Reflexive angle
Reflexive angle
a
b
c
d
e
f
g
h
i
j
k

- e.g. points b and f need to be deleted.



reflexive angles

Final result:



- <u>The merging procedure:</u>

1. Select an interior point p.
2. There are 3 sequences of points which have increasing polar angles with respect to p.

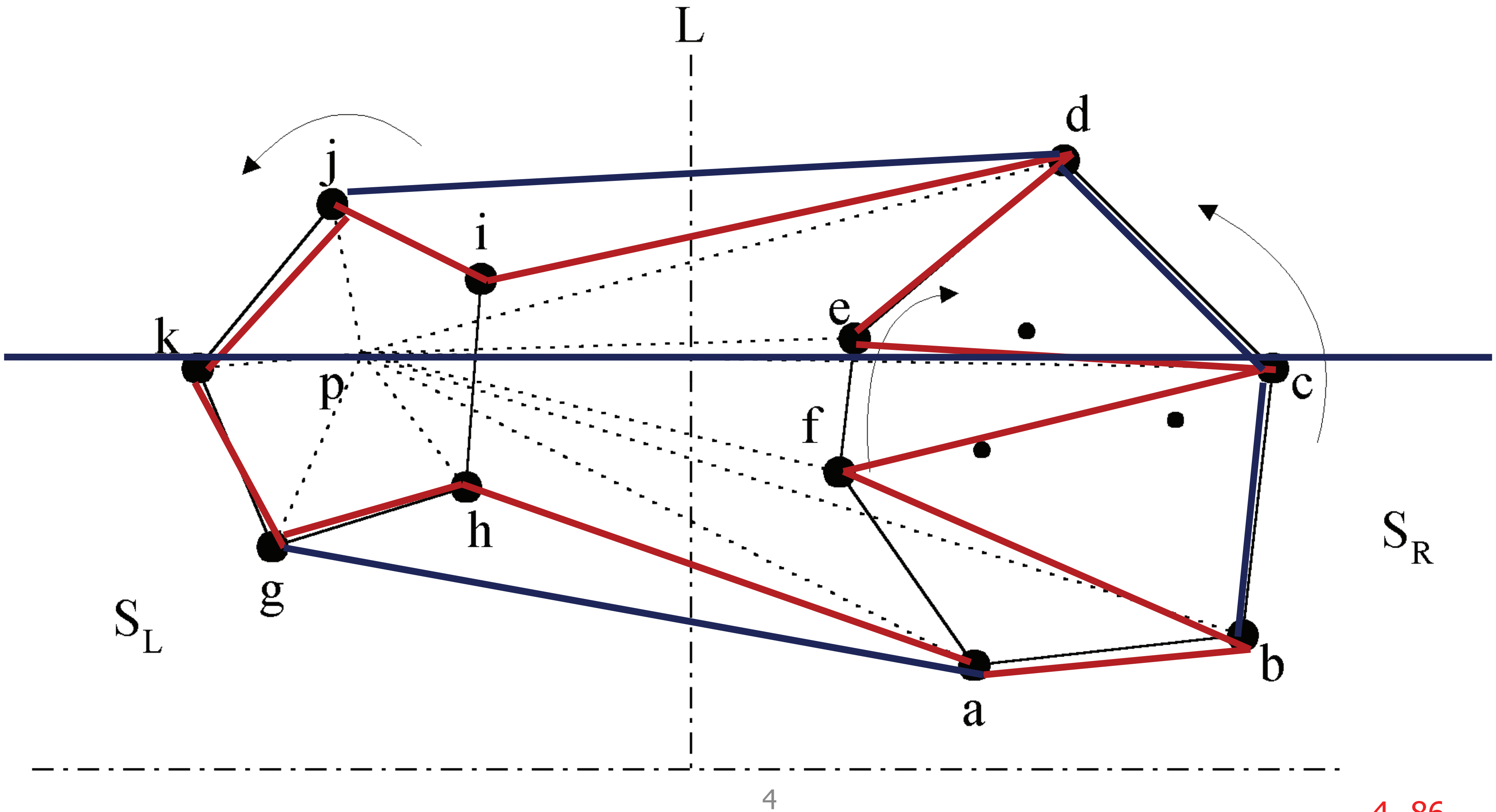
   (1) g, h, i, j, k

   (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 <u>Graham scan</u> to examine the points one by one and eliminate the points which cause <u>reflexive angles</u>.

- 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.

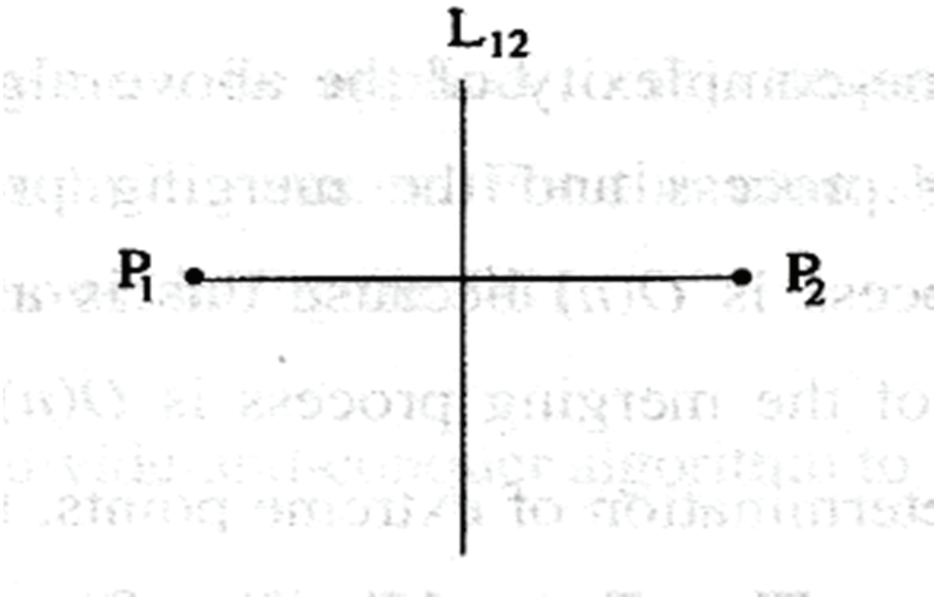
- Step 4: Apply the merging procedure to merge Hull(S_L) and Hull(S_R) together to form a convex hull.

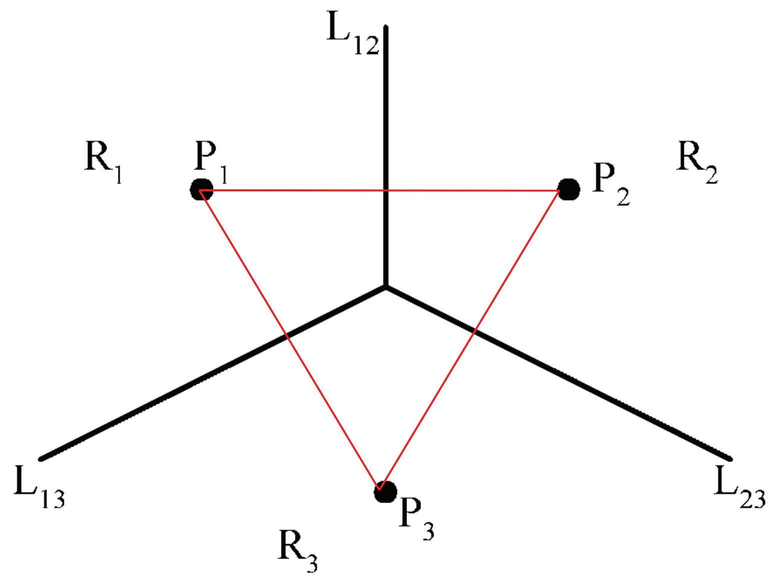
- Time complexity:

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

4.4 The Voronoi diagram problem

- e.g. The Voronoi diagram for two & three points



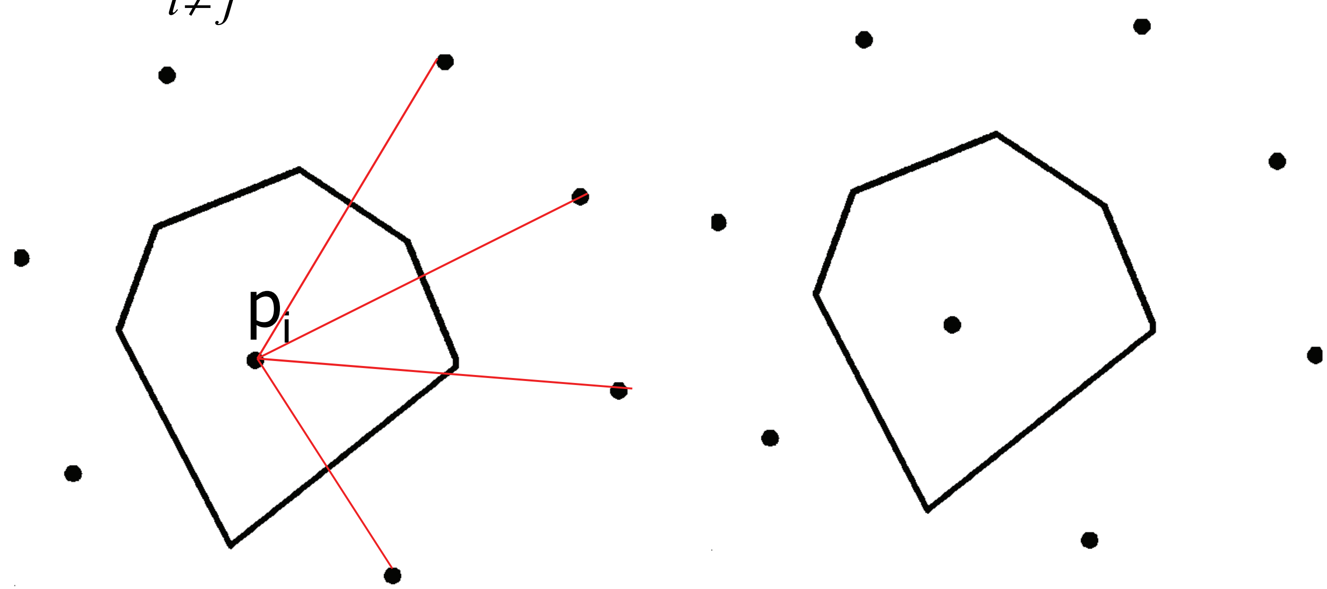


Each L_{ij} is the perpendicular bisector of the line.

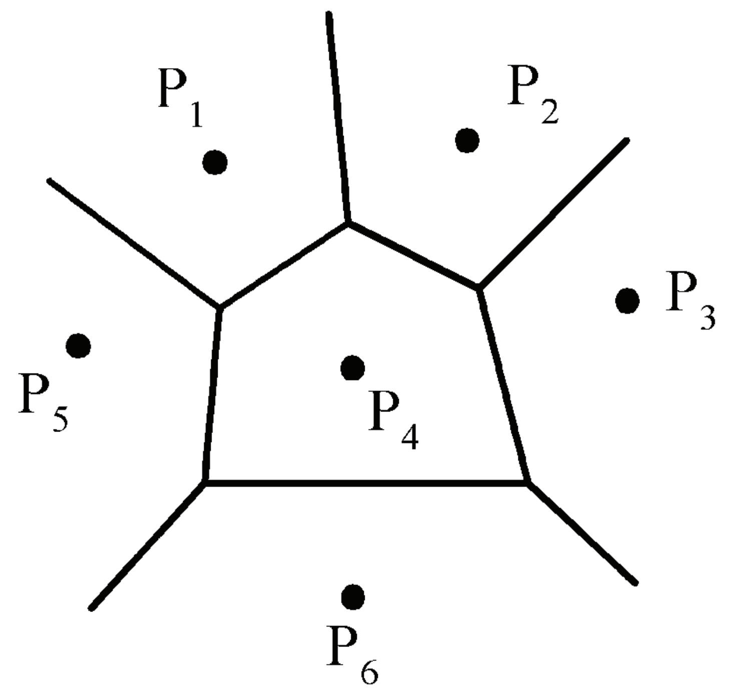
Definition of Voronoi diagrams

- **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

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

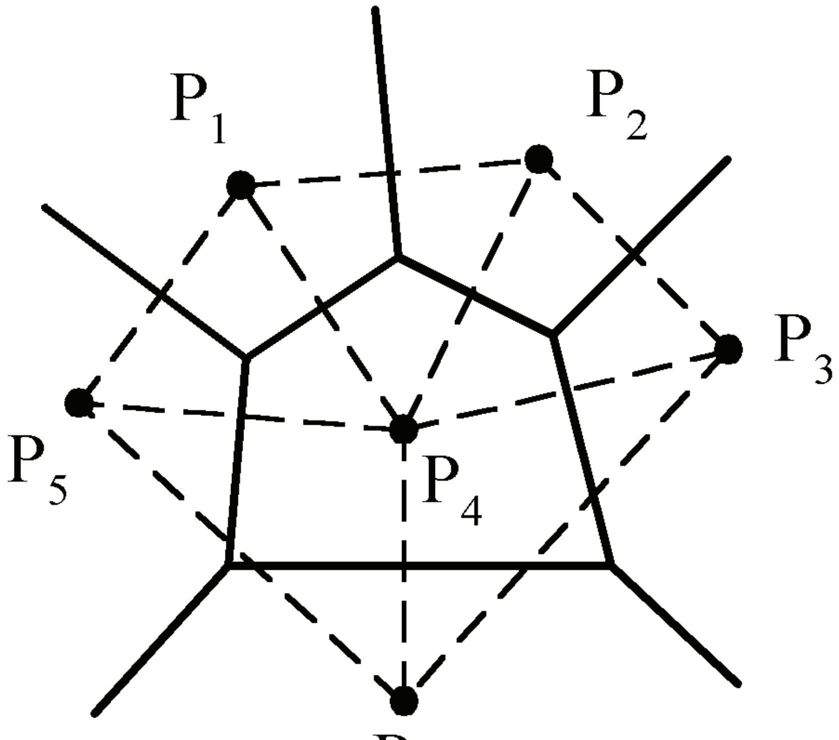


- Given a set of n points, the Voronoi diagram consists of all the Voronoi polygons of these points.



- 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.

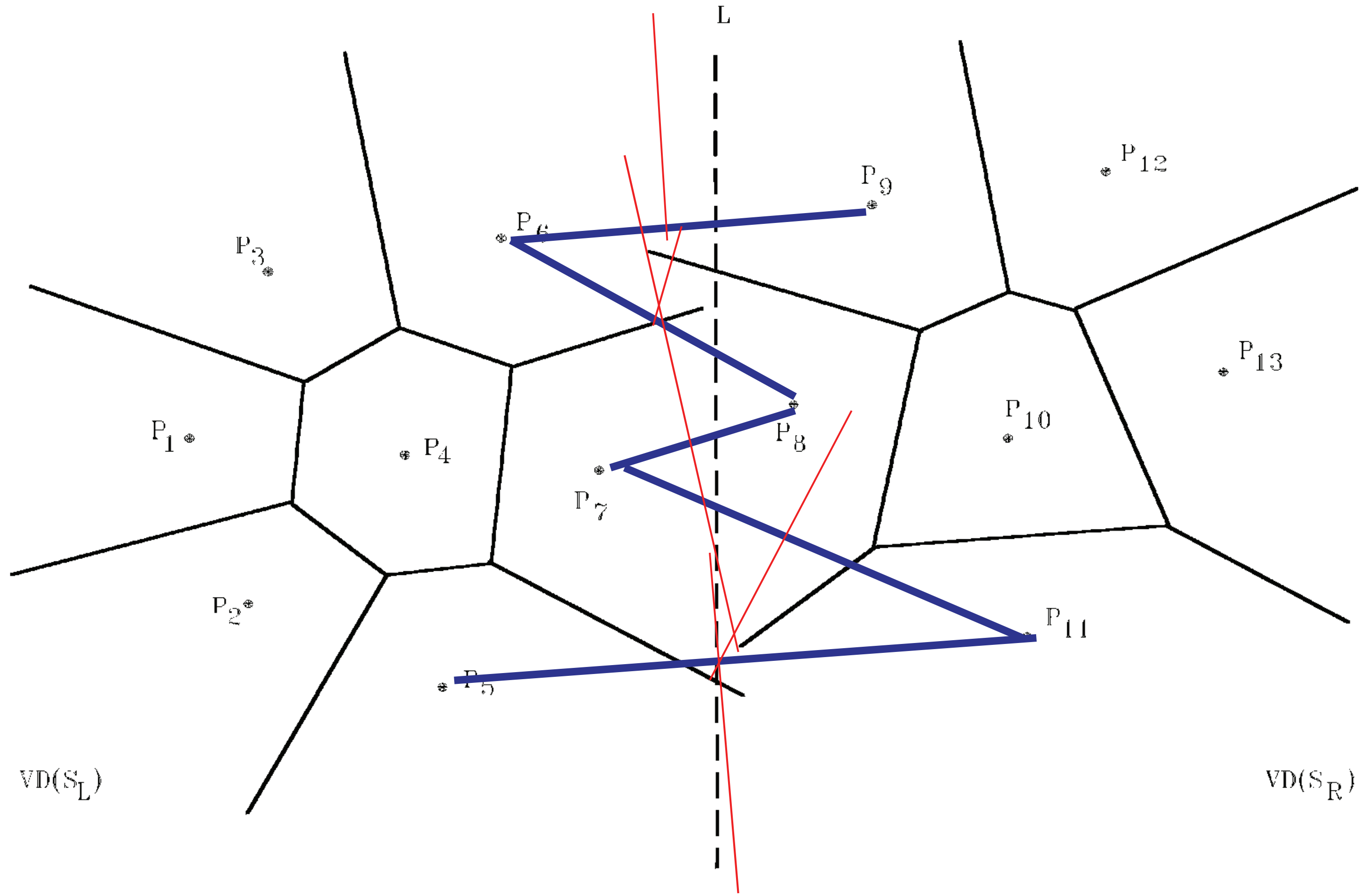


Fig. 5–17: Two Voronoi Diagrams After Step 2

Merging two Voronoi diagrams

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

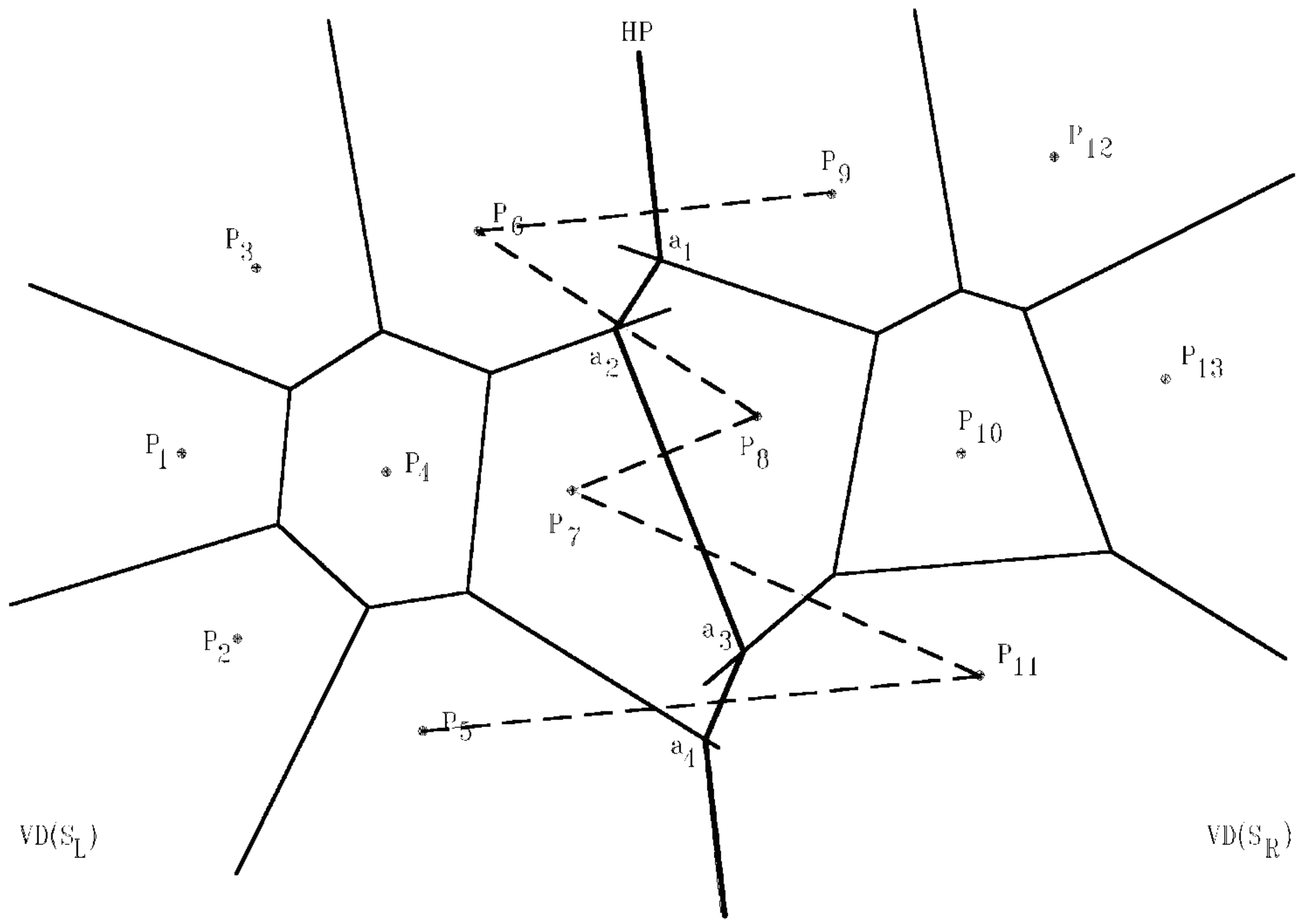


Fig. 5-18: The Piecewise Linear Hyperplane for the set of Points Shown in Fig. 5-17.

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(SL)* to the right of HP and all of VD(S*R*) to the left of HP, we obtain the resulting Voronoi diagram as shown in Figure 5-19.

The final Voronoi diagram

- After merging

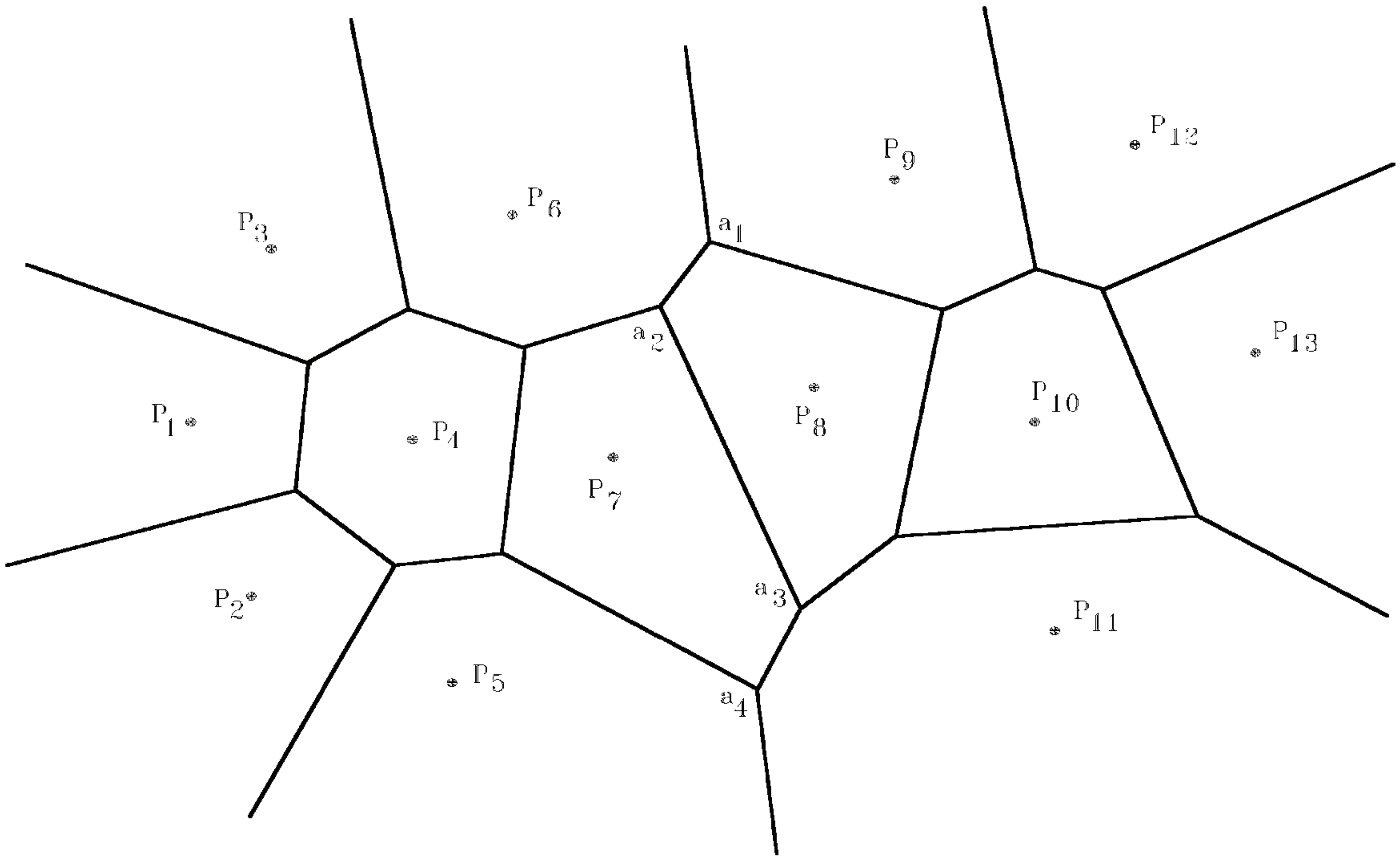


Fig. 5-19: The Voronoi Diagram of the Points in Fig. 5-17.

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.

(See details on the next page.)

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 by given later.)

Step 2: Find segments $\overline{P_aP_b}$ and $\overline{P_cP_d}$ which join $HULL(S_L)$ and $HULL(S_R)$ 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 $\overline{P_aP_b}$ lies above $\overline{P_cP_d}$. Let x = a, y = b, SG= $\overline{P_xP_y}$ and HP = $\varnothing$.

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

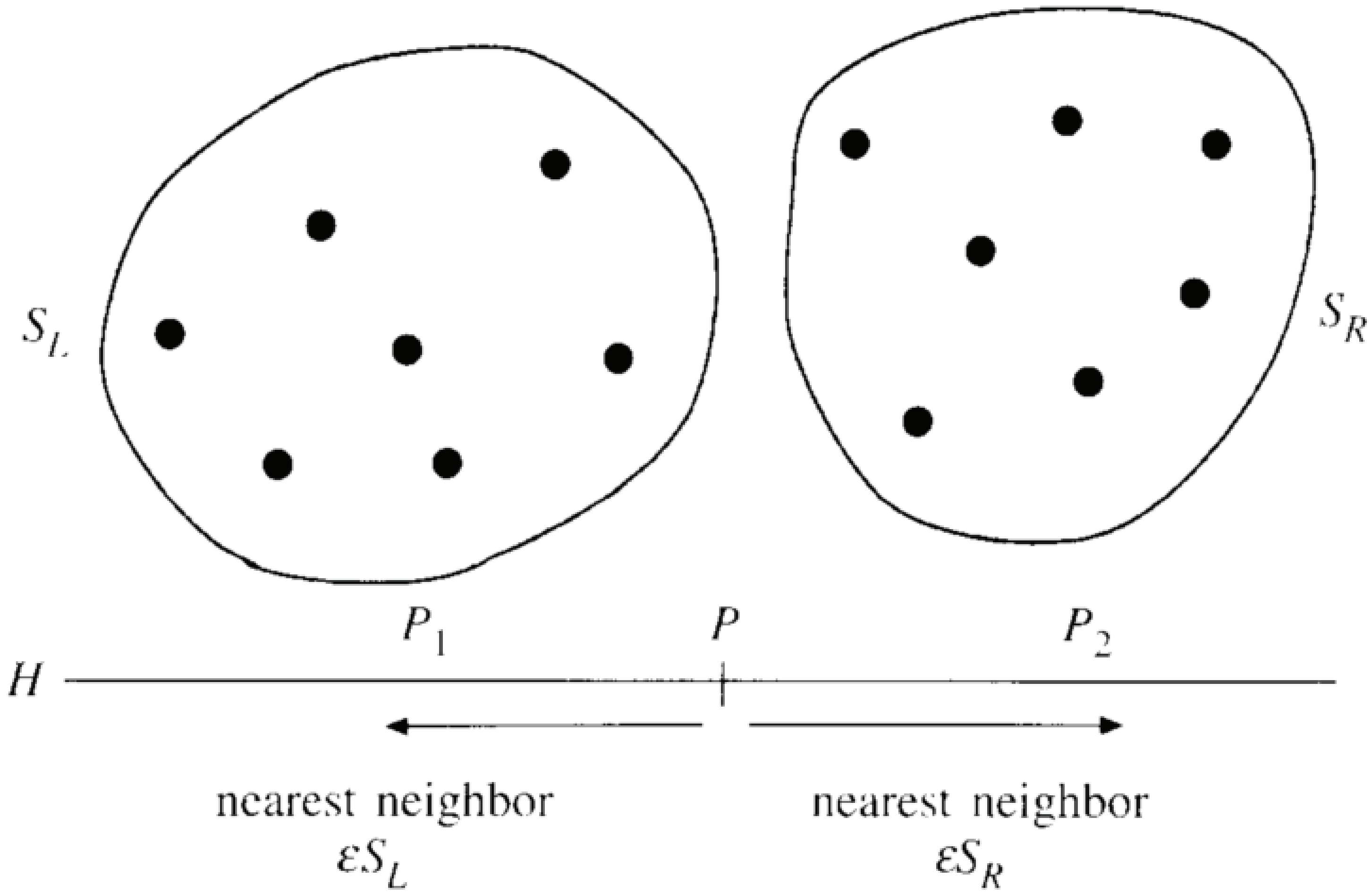
Step 4: The ray from VD(S_L) and VD(S_R) which BS first intersects with must be a perpendicular bisector of either $\overline{P_x P_z}$ or $\overline{P_y P_z}$ for some z. If this ray is the perpendicular bisector of $\overline{P_y P_z}$, then let SG = $\overline{P_x P_z}$; otherwise, let SG = $\overline{P_z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

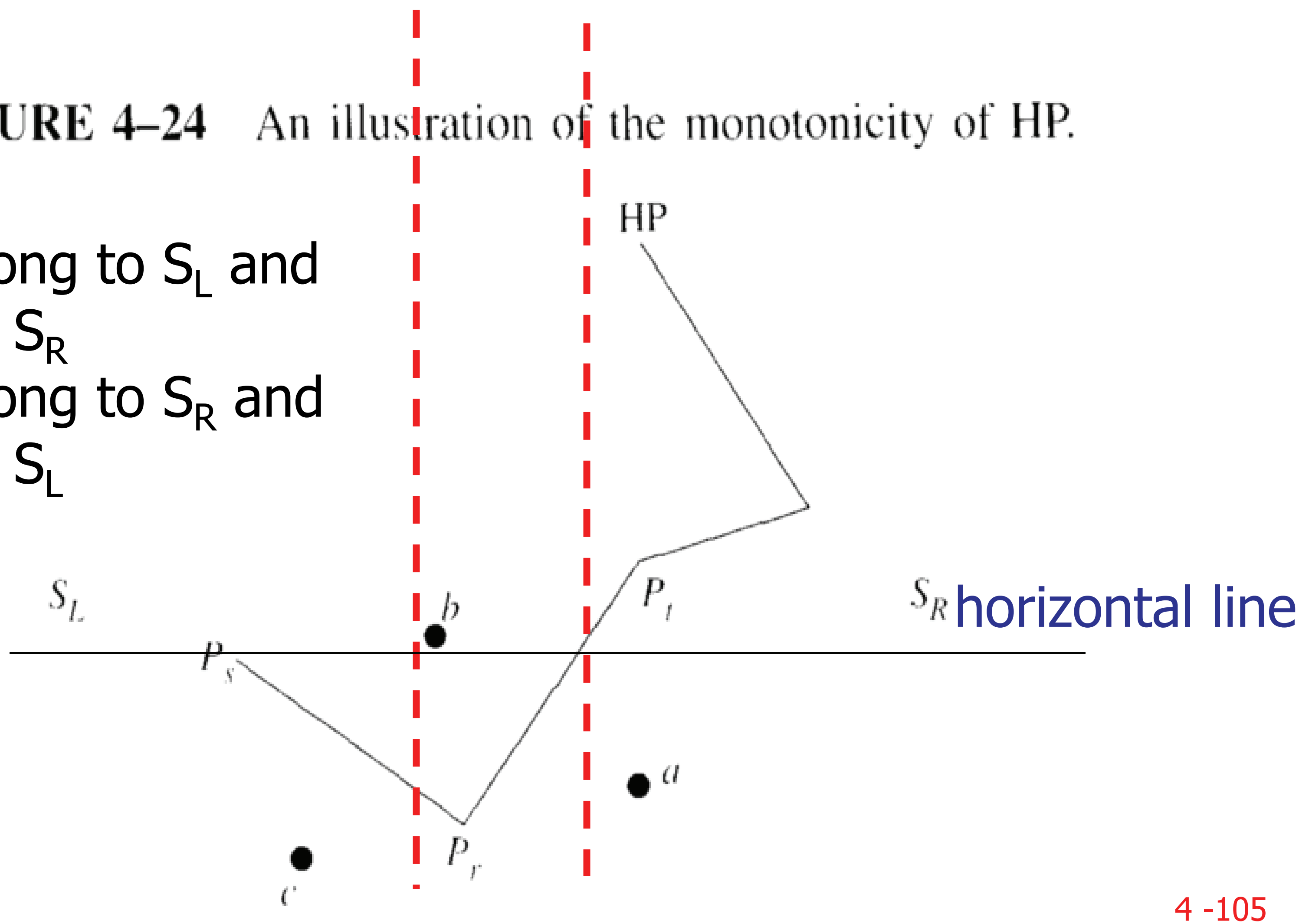


Each horizontal line H intersects with HP at one and only on point.

The HP is monotonic in y.

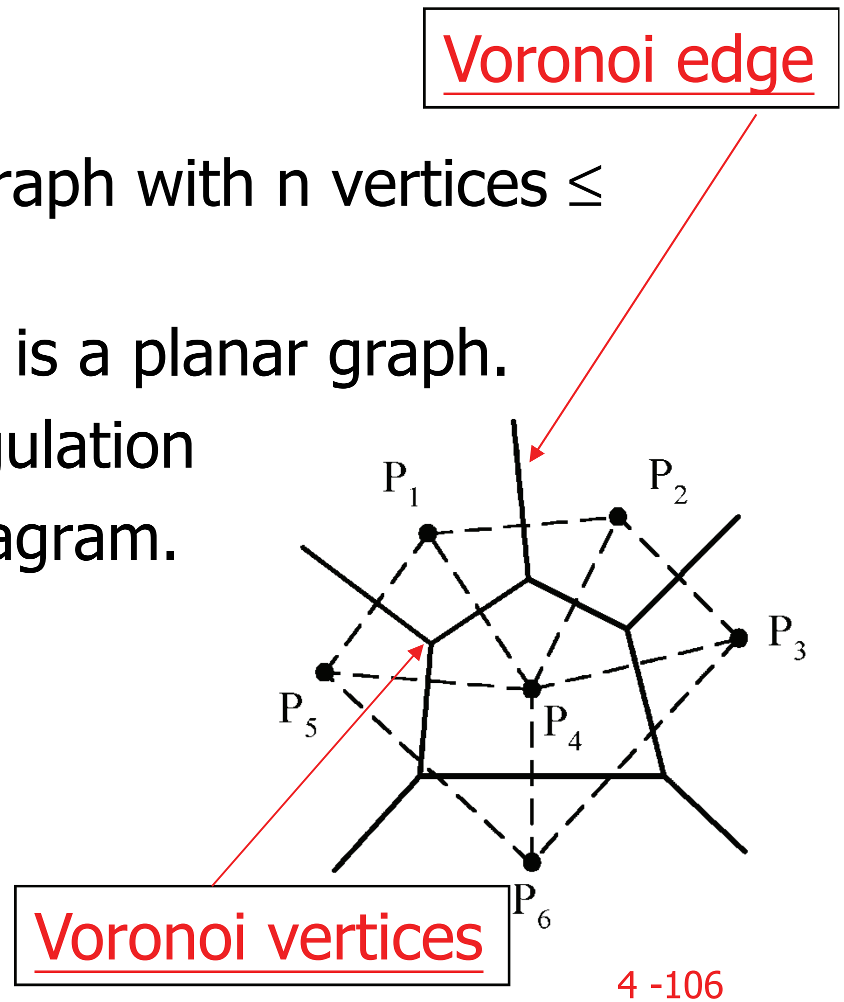
FIGURE 4–24 An illustration of the monotonicity of HP.

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



of Voronoi edges

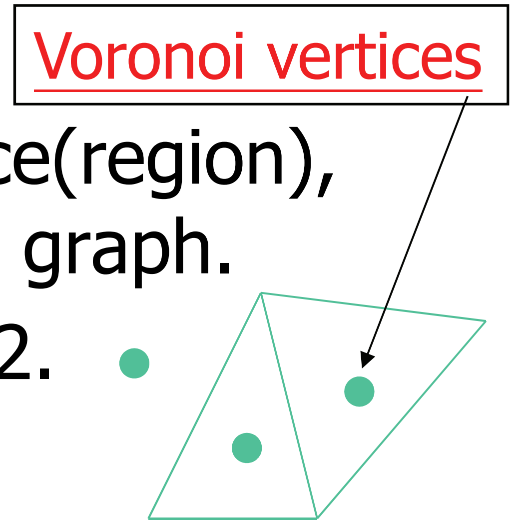
- # of edges of a Voronoi diagram $\leq$ 3n - 6, where n is # of points.
- Reasoning:
  i. # of edges of a planar graph with n vertices $\leq$ 3n - 6.
  ii. A Delaunay triangulation is a planar graph.
  iii. Edges in Delaunay triangulation $\xleftrightarrow{1-1}$ edges in Voronoi diagram.



Corollary: If G is a connected planar simple graph with E edges and V vertices where V≧3, then E≦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,
     $V = n$, $E \leq 3n - 6$
     $\Rightarrow F = E - V + 2 \leq 3n - 6 - n + 2 = 2n - 4$.



Reference: Rosen pp. 606~607.

Construct a convex hull from a Voronoi diagram

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

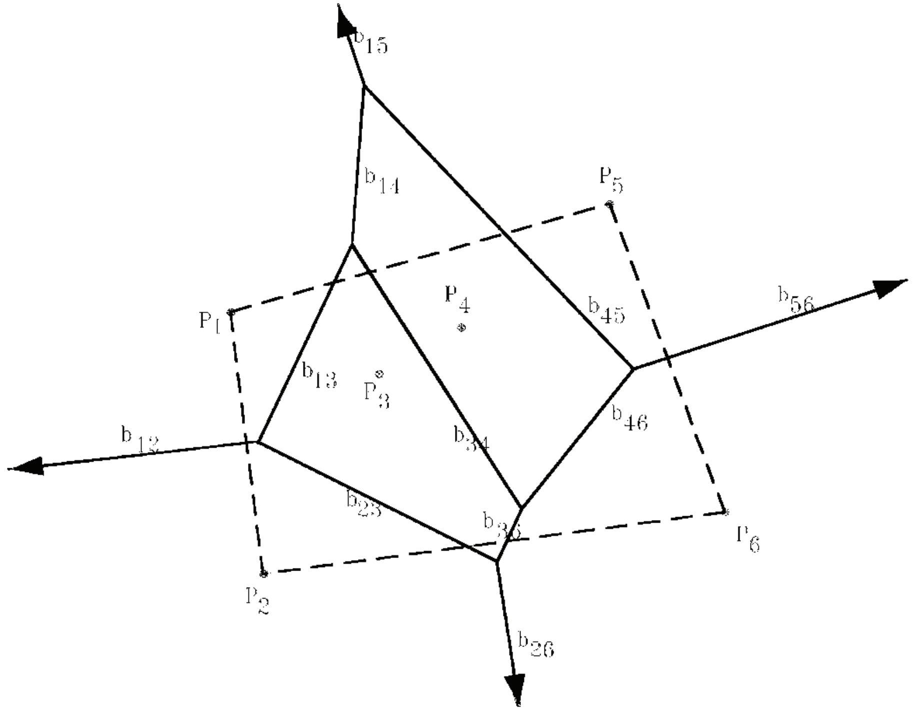


Fig. 5–25: Constructing a Convex Hull from a Voronoi Diagram

4

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.

$$
\begin{tikzcd}
\text{instance of A} \arrow[r, "\text{transformation}", "T(tr_1)"'] \arrow[d, "T(A)"'] & \text{instance of B} \arrow[d, "T(B)"', "\text{solver of B}"] \\
\text{answer of A} & \text{answer of B} \arrow[l, "\text{transformation}"', "T(tr_2)"]
\end{tikzcd}
$$

Note: $T(tr1) + T(tr2) < T(B)$

$T(A) \le T(tr1) + T(tr2) + T(B) \sim O(T(B))$

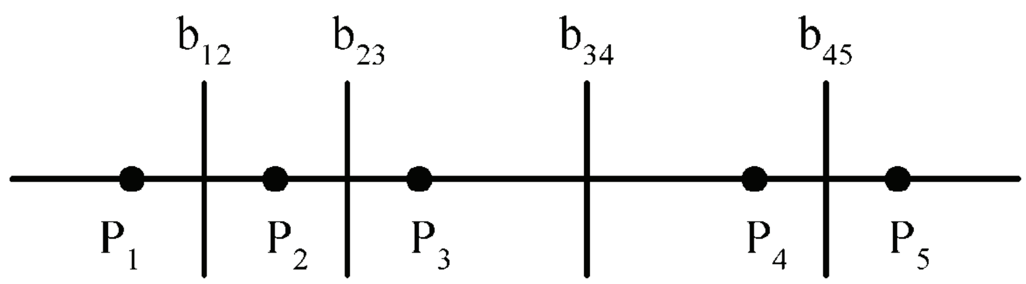
Lower bound of the Voronoi diagram

- 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 as shown in Figure 5-26.
- 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 $Q(n\log n)$ 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

5.5 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: $P_1, P_2, \ldots, P_n$, 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(logn)** with **preprocessing time O(nlogn).**

- Note that the Voronoi diagram divides the entire plane into regions $R_1, R_2 \quad R_n$. 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,…, 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.

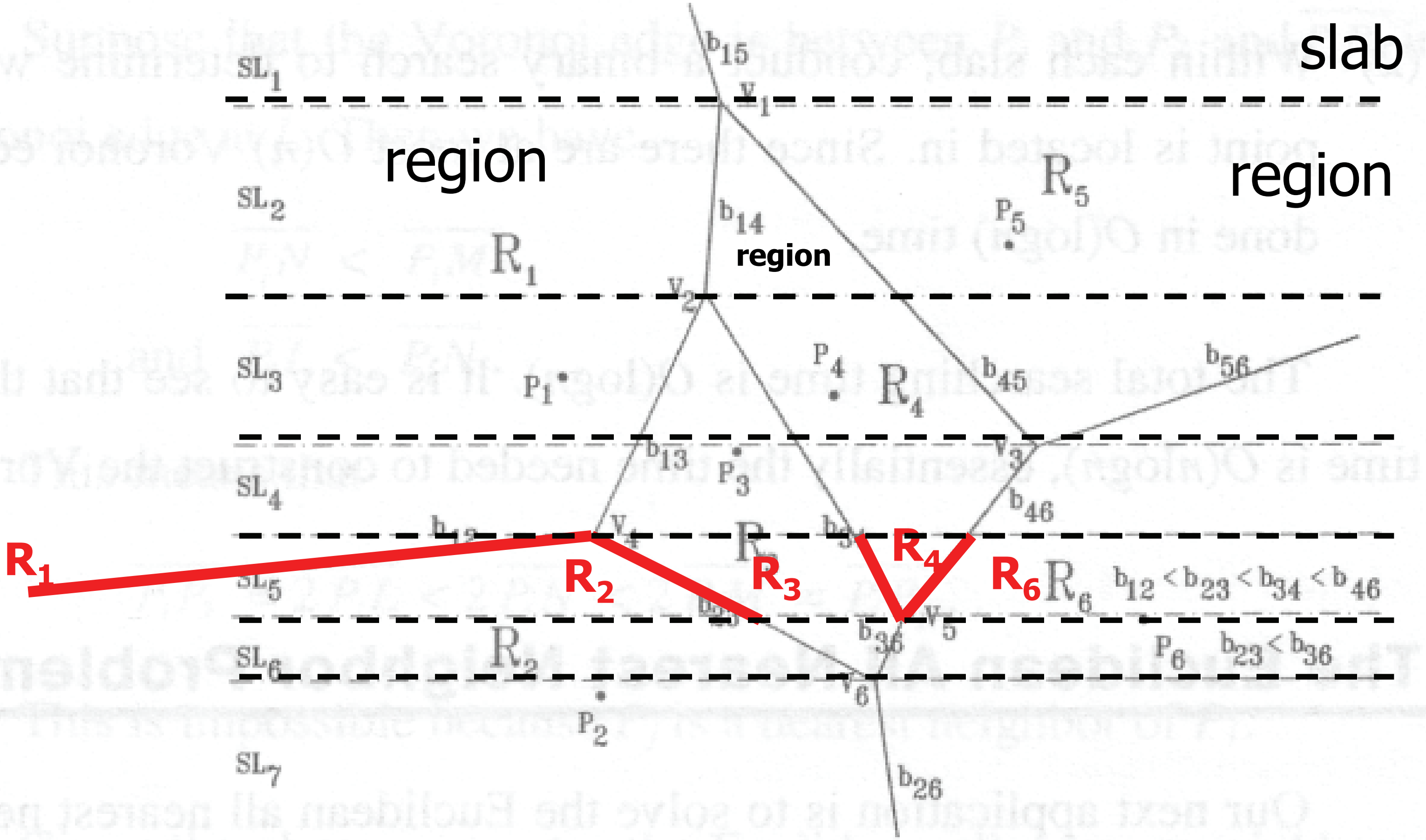


Figure 5-27 The Application of Voronoi Diagrams to Slove the Euclidean Nearest Neighbor Searching Problem.

Euclidean nearest neighbor searching algorithm

- Conduct a binary search to determine which slab this testing point is located. Since there are at most *0(n)* Voronoi vertices, this can be done in O(logn) time.
- Within each slab, conduct a binary search to determine which region this point is located in. Since there are at most *0(n)* Voronoi edges, this can be done in O(logn) time.
- The total searching time is O(logn).
- It is easy to see that the preprocessing time is O(nlogn), essentially the time needed to construct the Voronoi diagram.

The Euclidean all nearest neighbor problem.

- We are given a set of n planar points $P_1, P_2, \ldots, P_n$.
- 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_iP_j must be outside of the Voronoi polygon associated with P_i.
- Let P, Pi intersect the bisector at M and some Voronoi edge at N, as illustrated in Figure 5-28.

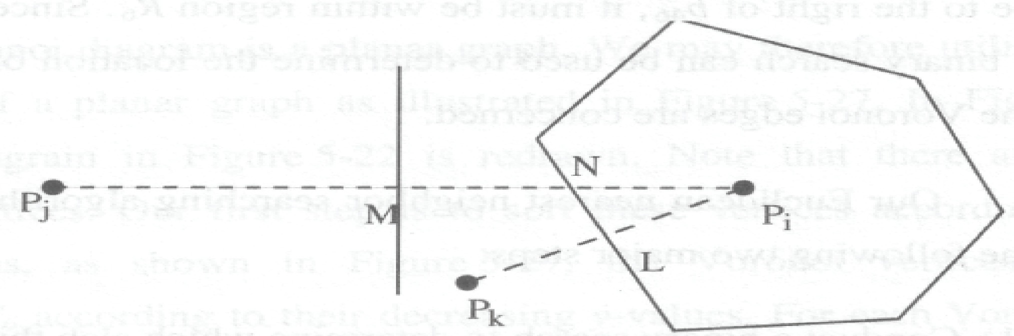


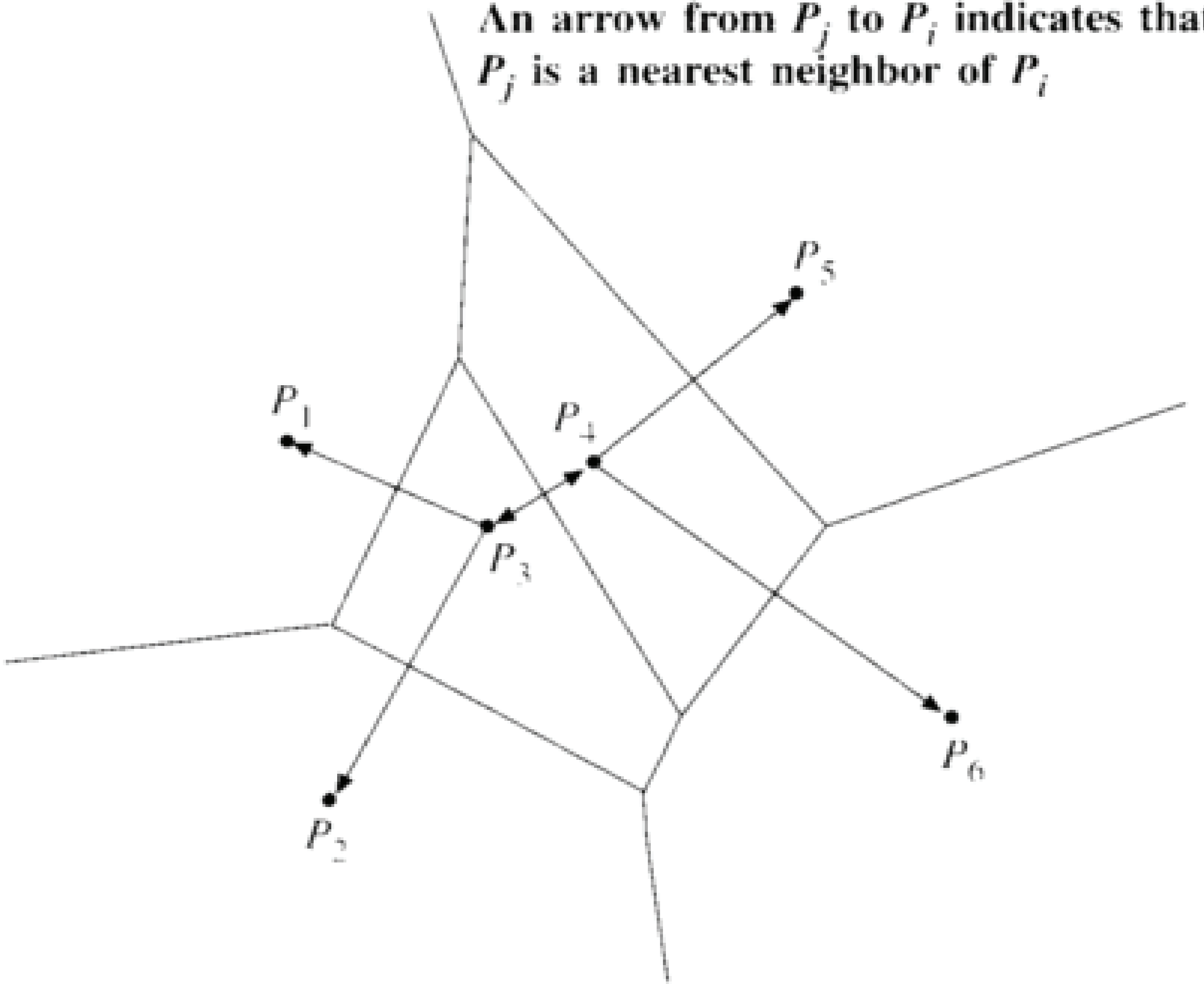
Figure 5-28 An Illustration Showing the Nearest Neighbor Property of Voronoi Diagrams.

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(nlogn) time**.

FIGURE 4–29 The all nearest neighbor relationship.

An arrow from P_j to P_i indicates that P_j is a nearest neighbor of P_i



5.8 Matrix multiplication

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

$$C = AB$$

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

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

$$\overbrace{\begin{bmatrix} \boxed{a_{11} \quad a_{12} \quad a_{13}} \\ a_{21} \quad a_{22} \quad a_{23} \\ a_{31} \quad a_{32} \quad a_{33} \end{bmatrix}}^{n個} \underbrace{\begin{bmatrix} \boxed{\begin{matrix} b_{11} \\ b_{21} \\ b_{31} \end{matrix}} & \begin{matrix} b_{12} \\ b_{22} \\ b_{32} \end{matrix} & \begin{matrix} b_{13} \\ b_{23} \\ b_{33} \end{matrix} \end{bmatrix}}_{n個} = \begin{bmatrix} \boxed{c_{11}} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

$$c_{ij} = \sum^{n} a_{ik} \cdot b_{kj}$$

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

- $C = AB$

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

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

- Time complexity:

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

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

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

Pseudo code:

Square-Matrix-Multiply-Recursive(A,B)

n=A.rows

let C be a new n x n matrix

if n==1 **Base case** $T(1) = \Theta($

$c_{11} = a_{11} \cdot b_{11}$ $\Theta(1)$

else partition the matrix into 4 n/2 x n/2 matrices

Recursive case

C_{11} =Square-Matrix-Multiply-Recursive(A_{11},B_{11})
+Square-Matrix-Multiply-Recursive(A_{12},B_{21})

C_{12} =Square-Matrix-Multiply-Recursive(A_{11},B_{12})
+Square-Matrix-Multiply-Recursive(A_{12},B_{22})

C_{21} =Square-Matrix-Multiply-Recursive(A_{21},B_{11})
+Square-Matrix-Multiply-Recursive(A_{22},B_{21})

C_{22} =Square-Matrix-Multiply-Recursive(A_{21},B_{12})
+Square-Matrix-Multiply-Recursive(A_{22},B_{22})

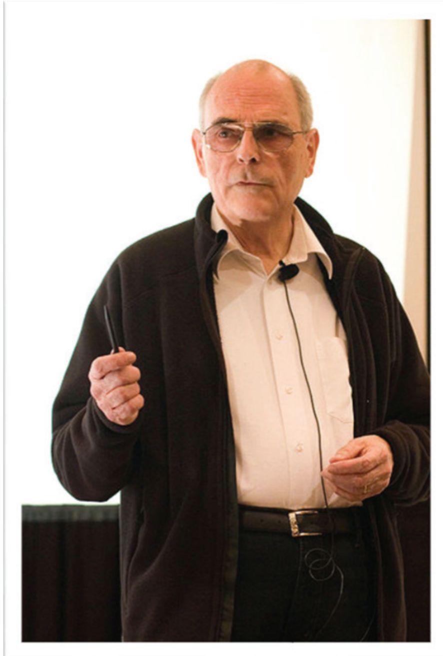
return C

$8T(\frac{n}{2})$

Combine $\Theta(n^2)$

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:

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

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