CSC 411
Design and Analysis of Algorithms

Chapter 5 Divide and Conquer - Part 2

Instructor: Minhee Jun

Divide-and-Conquer Examples

- Sorting: mergesort and quicksort (5.1 & 5.2)
- Binary tree traversals (5.3)
- **Multiplication of large integers and**
  Matrix multiplication: Strassen's algorithm (5.4)
- **Closest-pair and**
  convex-hull algorithms (5.5)

5.4 Multiplication of Large Integers

Consider the problem of multiplying two (large) n-digit integers represented by arrays of their digits such as:

A = 12345678901357986429   B = 87654321284820912836

The grade-school algorithm:

$$
\begin{array}{r}
a_1 \;\; a_2 \ldots \;\; a_n \\
b_1 \;\; b_2 \ldots \;\; b_n \\
\hline
(d_{10}) \;\; d_{11} \;\; d_{12} \;\; \ldots \;\; d_{1n} \\
(d_{20}) \; d_{21} \;\; d_{22} \ldots \;\; d_{2n} \;\;\;\; \\
\ldots \ldots \ldots \ldots \ldots \ldots \ldots \;\;\;\;\;\;\;\; \\
(d_{n0}) \; d_{n1} d_{n2} \ldots d_{nn} \;\;\;\;\;\;\;\;\;\;\;\; \\
\hline
\end{array}
$$

Questions:
Can we reduce the number of one-digit multiplications?

Efficiency: n^2 one-digit multiplications

Example of Large-Integer Multiplication

To demonstrate the basic idea of the algorithm, let us start with a case of two-digit integers, say, 23 and 14. These numbers can be represented as follows:

$$23 = 2 \cdot 10^1 + 3 \cdot 10^0 \quad \text{and} \quad 14 = 1 \cdot 10^1 + 4 \cdot 10^0.$$

Now let us multiply them:

$$\begin{aligned} 23 * 14 &= (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0) \\ &= (2 * 1)10^2 + \underline{(2 * 4 + 3 * 1)}10^1 + (3 * 4)10^0. \end{aligned}$$

$$2 * 4 + 3 * 1 = (2 + 3) * (1 + 4) - 2 * 1 - 3 * 4.$$

The idea is to decrease the number of multiplications from 4 to 3.

Large-Integer Multiplication

For any pair of two-digit numbers $a = a_1a_0$ and $b = b_1b_0$, their product c can be computed by the formula

$$c = a * b = c_2 10^2 + c_1 10^1 + c_0,$$

where

$c_2 = a_1 * b_1$ is the product of their first digits,

$c_0 = a_0 * b_0$ is the product of their second digits,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's digits and the sum of the b's digits minus the sum of c_2 and c_0.

Large-Integer Multiplication

Multiplying two n-digit integers a and b where n is a positive even number, let us divide both numbers in the middle. The first half of the a's digits by a_1 and the second half by a_0; for b, the notations are b_1 and b_0, respectively. In these notations, $a = a_1 a_0$ implies that $a = a_1 10^{n/2} + a_0$ and $b = b_1 b_0$ implies that $b = b_1 10^{n/2} b_0$.

$$
\begin{aligned}
c = a * b &= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0) \\
&= (a_1 * b_1) 10^n + (a_1 * b_0 + a_0 * b_1) 10^{n/2} + (a_0 * b_0) \\
&= c_2 10^n + c_1 10^{n/2} + c_0,
\end{aligned}
$$

where

$c_2 = a_1 * b_1$ is the product of their first halves,

$c_0 = a_0 * b_0$ is the product of their second halves,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0.

Multiplication of Large Integers

How many digit multiplications does this algorithm make? Since multiplication of n-digit numbers requires three multiplications of $n/2$-digit numbers, the recurrence for the number of multiplications $M(n)$ is

$$M(n) = 3M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Master Theorem If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence (5.1), then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the O and Ω notations, too.

$$M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}.$$

(Better than n^2)

Additions and subtractions of Large Integers

But what about additions and subtractions? Have we not decreased the number of multiplications by requiring more of those operations? Let $A(n)$ be the number of digit additions and subtractions executed by the above algorithm in multiplying two n-digit decimal integers. Besides $3A(n/2)$ of these operations needed to compute the three products of $n/2$-digit numbers, the above formulas require five additions and one subtraction. Hence, we have the recurrence

$$A(n) = 3A(n/2) + cn \quad \text{for } n > 1, \quad A(1) = 1.$$

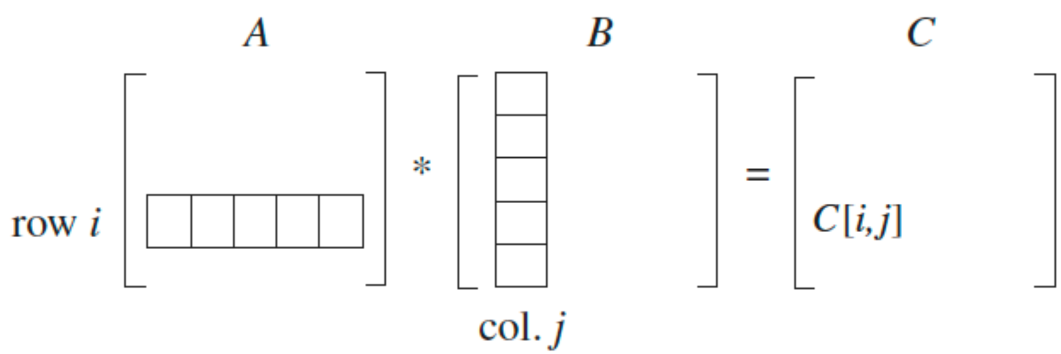
Applying the Master Theorem, which was stated in the beginning of the chapter, we obtain $A(n) \in \Theta(n^{\log_2 3})$, which means that the total number of additions and subtractions have the same asymptotic order of growth as the number of multiplications.

Exercise 5.4

2. Compute $2101 * 1130$ by applying the divide-and-conquer algorithm outlined in the text.

Matrix Multiplication (From Chap 2.3)

EXAMPLE 3 Given two $n \times n$ matrices A and B, find the time efficiency of the definition-based algorithm for computing their product $C = AB$. By definition, C is an $n \times n$ matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B:



where $C[i, j] = A[i, 0]B[0, j] + \cdots + A[i, k]B[k, j] + \cdots + A[i, n-1]B[n-1, j]$ for every pair of indices $0 \le i, j \le n-1$.

Strassen's Matrix Multiplication

Question: can we reduce the number of operations in two matrices multiplications C = A × B?

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix},$$

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11}),$$

$$m_2 = (a_{10} + a_{11}) * b_{00},$$

$$m_3 = a_{00} * (b_{01} - b_{11}),$$

$$m_4 = a_{11} * (b_{10} - b_{00}),$$

$$m_5 = (a_{00} + a_{01}) * b_{11},$$

$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01}),$$

$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$

Strassen's Matrix Multiplication

Strassen observed [1969] that the product of two matrices n×n can be computed as follows:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right].$$

- Let A and B be two n × n matrices where n is a power of 2.

  (If n is not a power of 2, matrices can be padded with rows and columns of zeros.)
- We can divide A, B, and their product C into four n/2 × n/2 submatrices

Formulas for Strassen's Algorithm

$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$

$M_2 = (A_{10} + A_{11}) * B_{00}$

$M_3 = A_{00} * (B_{01} - B_{11})$

$M_4 = A_{11} * (B_{10} - B_{00})$

$M_5 = (A_{00} + A_{01}) * B_{11}$

$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$

$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$

The size of A_{00}, A_{11}, B_{00}, B_{11} etc.: n/2

Growth function=?

$M(n) = 7M(n/2)$, $M(1) = 1$

Analysis of Strassen's Algorithm

Let us evaluate the asymptotic efficiency of this algorithm. If $M(n)$ is the number of multiplications made by Strassen's algorithm in multiplying two $n \times n$ matrices (where n is a power of 2), we get the following recurrence relation for it:

$$M(n) = 7M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Master Theorem If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence (5.1), then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the O and Ω notations, too.

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807},$$

vs. $\Theta(n^3)$ of brute-force algorithm.

Analysis of Strassen's Algorithm

Since this savings in the number of multiplications was achieved at the expense of making extra additions, we must check the number of additions $A(n)$ made by Strassen's algorithm. To multiply two matrices of order $n > 1$, the algorithm needs to multiply seven matrices of order $n/2$ and make 18 additions/subtractions of matrices of size $n/2$; when $n = 1$, no additions are made since two numbers are simply multiplied. These observations yield the following recurrence relation:

$$A(n) = 7A(n/2) + 18(n/2)^2 \quad \text{for } n > 1, \quad A(1) = 0.$$

Master Theorem If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence (5.1), then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the O and Ω notations, too.

$$A(n) \in \Theta(n^{\log_2 7})$$

Exercise 5.4-7

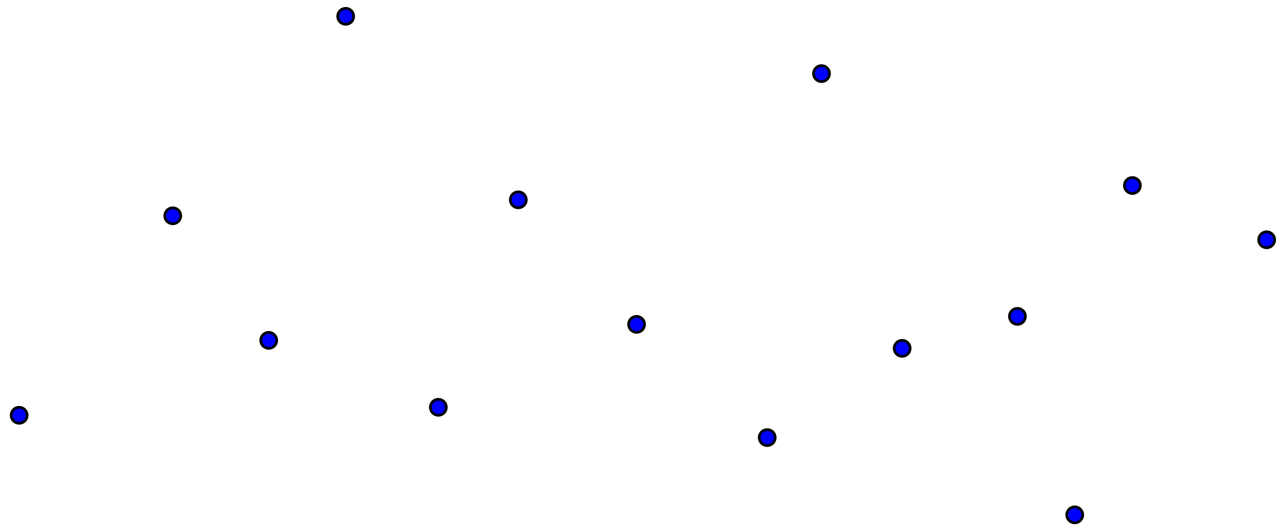
7. Apply Strassen's algorithm to compute

$$\begin{bmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

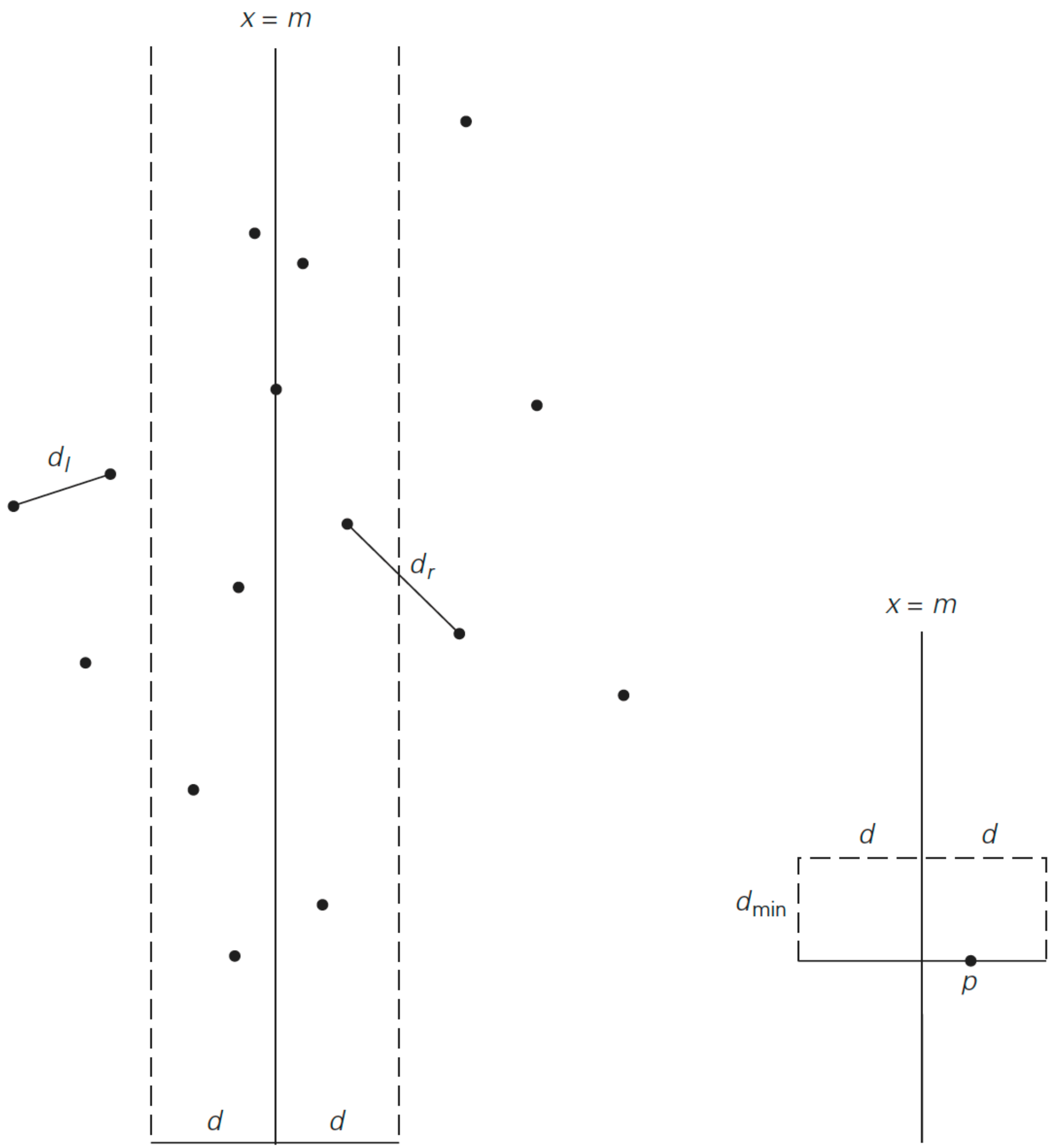
exiting the recursion when $n = 2$, i.e., computing the products of 2×2 matrices by the brute-force algorithm.

5.5 Closest-Pair Problem

Given: A set of points in 2-D



5.5

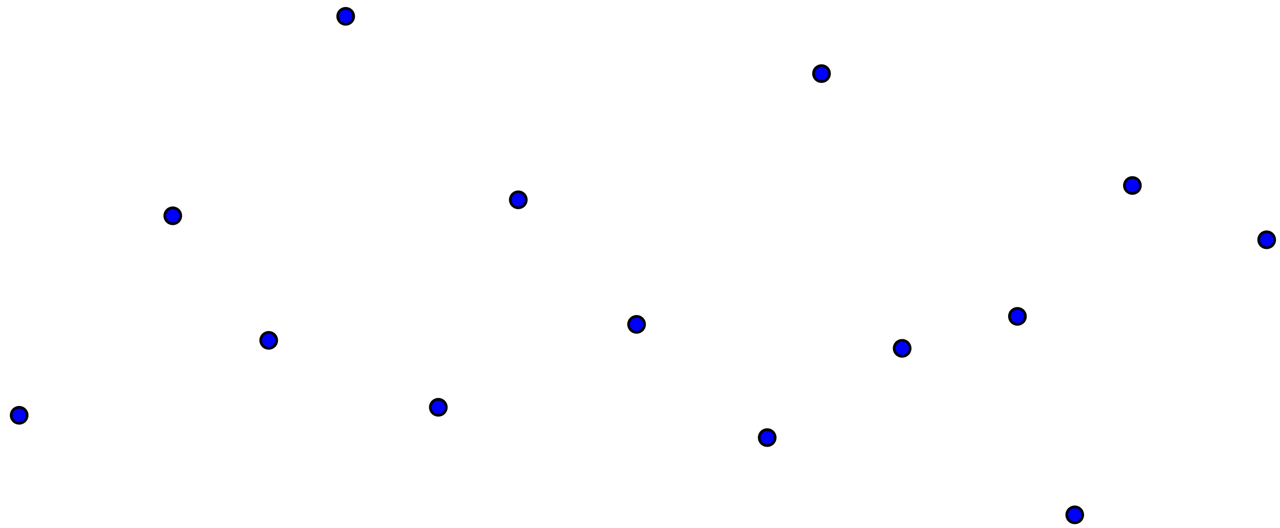


(a)

(b)

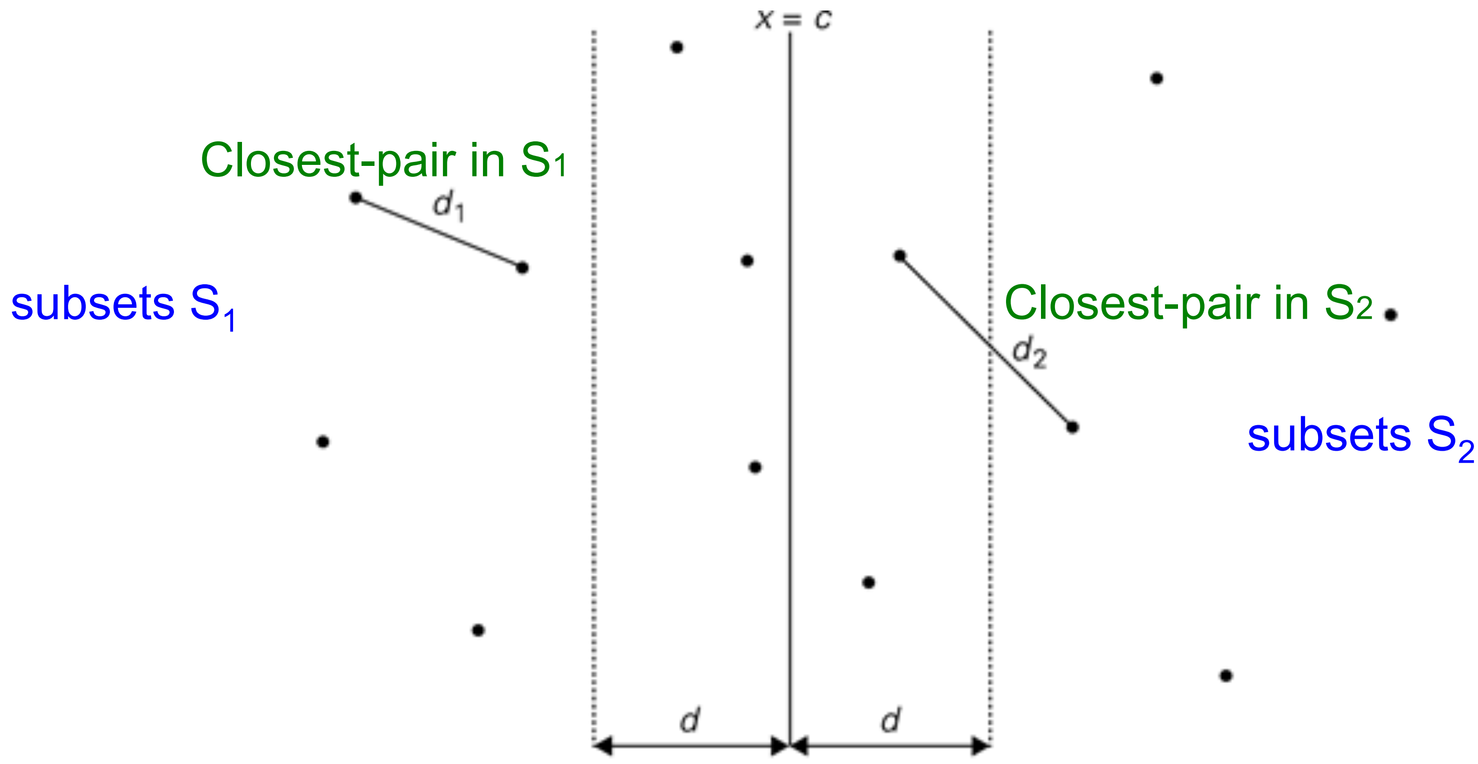
Closest-Pair Problem by Divide-and-Conquer

Step 1: Sort the points in one D



Closest-Pair Problem by Divide-and-Conquer

Step 1 Divide the points given into two subsets S_1 and S_2 by a vertical line $\underline{x = c}$ so that half the points lie to the left or on the line and half the points lie to the right or on the line.



$d = \min\{d_1, d_2\}$

Closest-Pair Problem by Divide-and-Conquer

Step 2 Find recursively the closest pairs for the left and right subsets.

Step 3 Set $d = \min\{d_1, d_2\}$

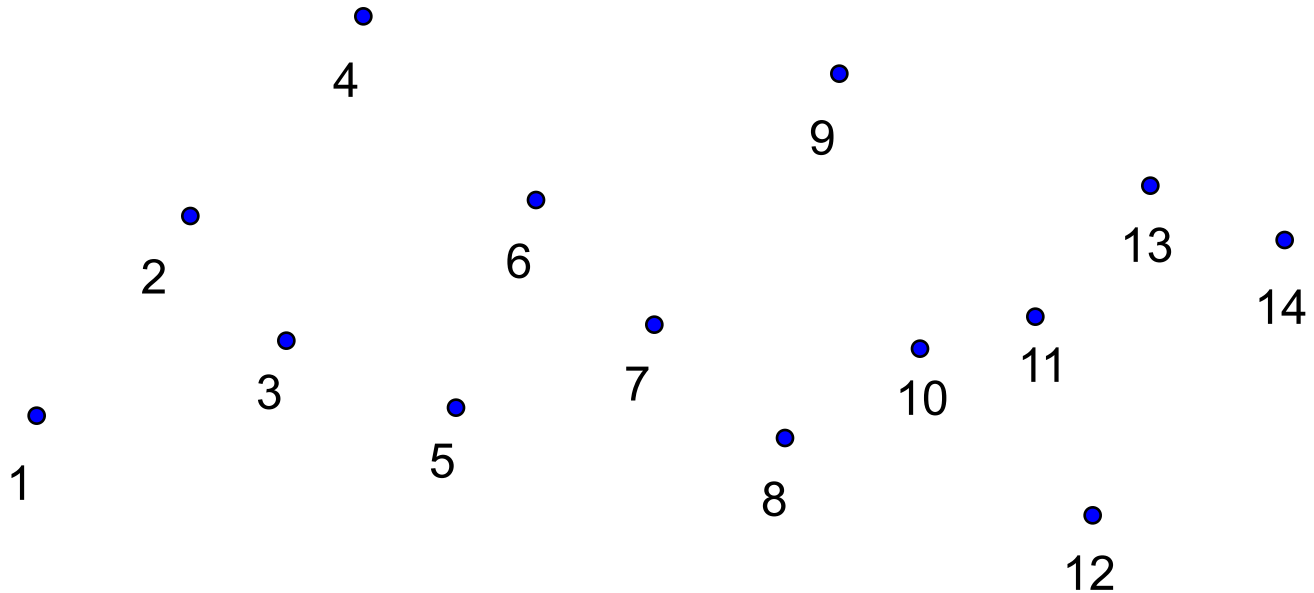
We can limit our attention to the points in the symmetric vertical strip of width $2d$ as possible closest pair. Let P_1 and P_2 be the subsets of points in the left subset S_1 and of the right subset S_2, respectively, that lie in this vertical strip. The points in P_1 and P_2 are stored in increasing order of their y coordinates, which is maintained by merging during the execution of the next step.

Step 4 For every point $p(x,y)$ in P_1, we inspect points in P_2 that may be closer to p than d.

There can be no more than 6 such points (because $d \leq d_2$)!

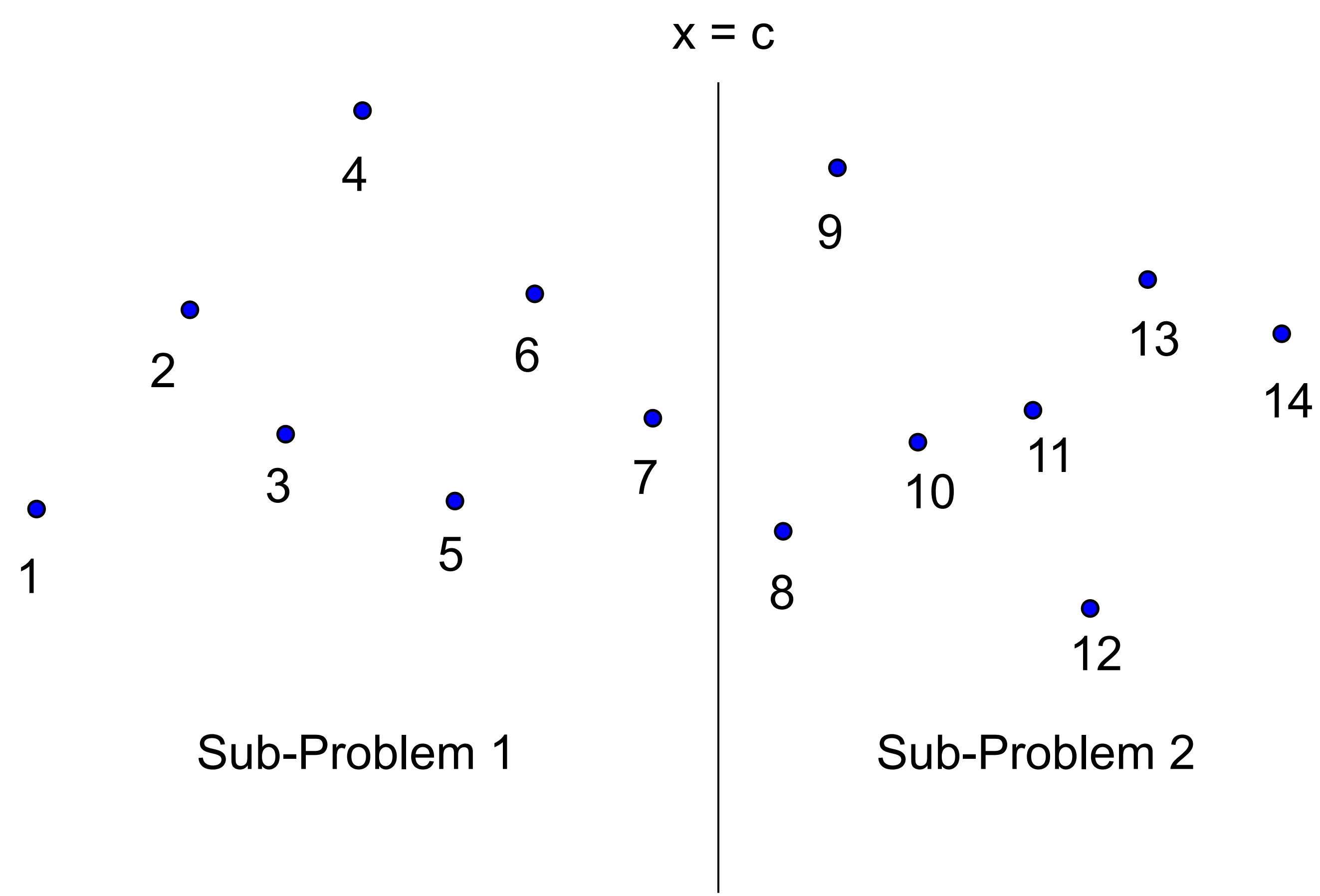
Closest-Pair Problem by Divide-and-Conquer

1. Lets sort based on the X-axis
O(n log n) using quicksort or mergesort



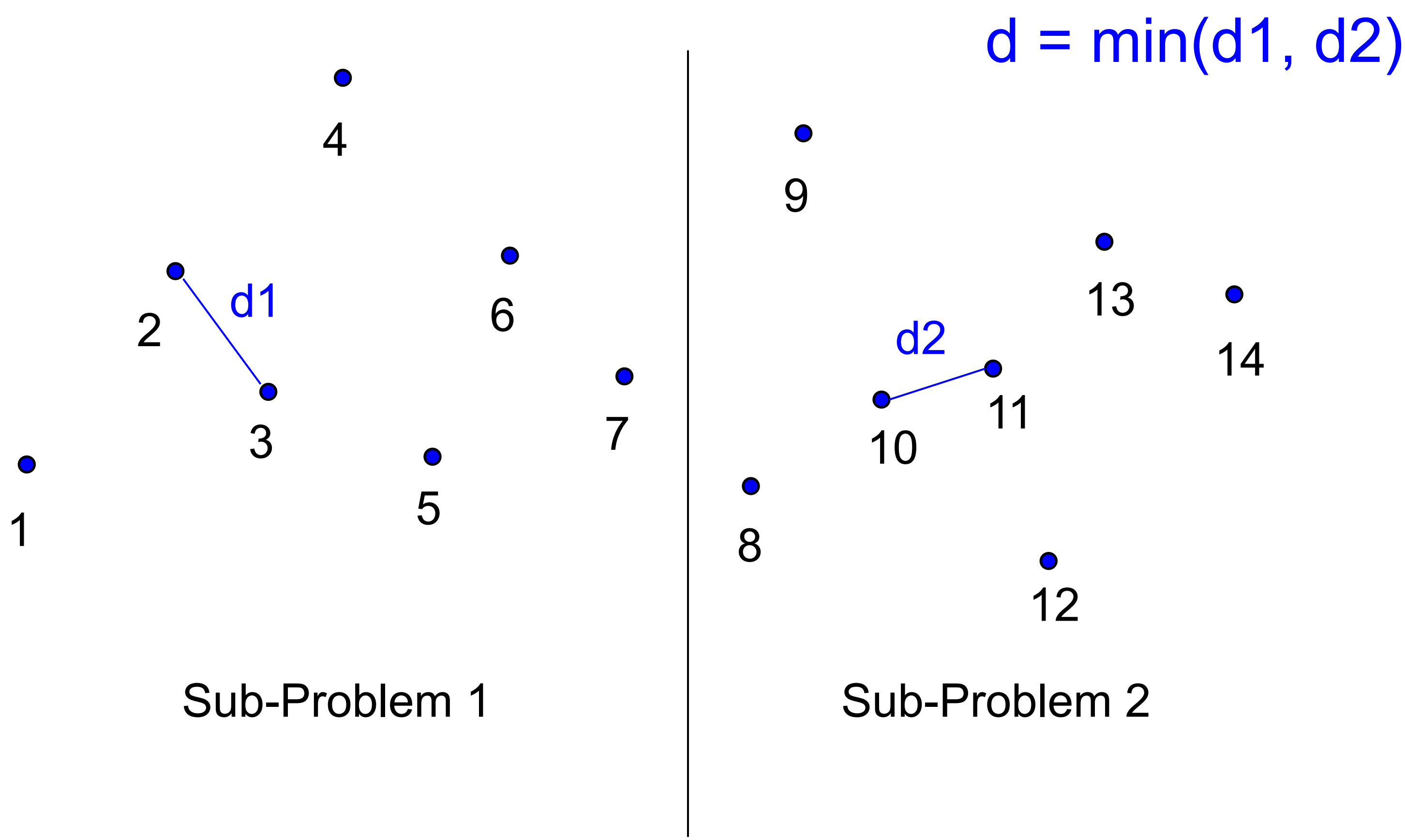
Closest-Pair Problem by Divide-and-Conquer

2. Split the points, i.e.,
Draw a line at the mid-point between 7 and 8



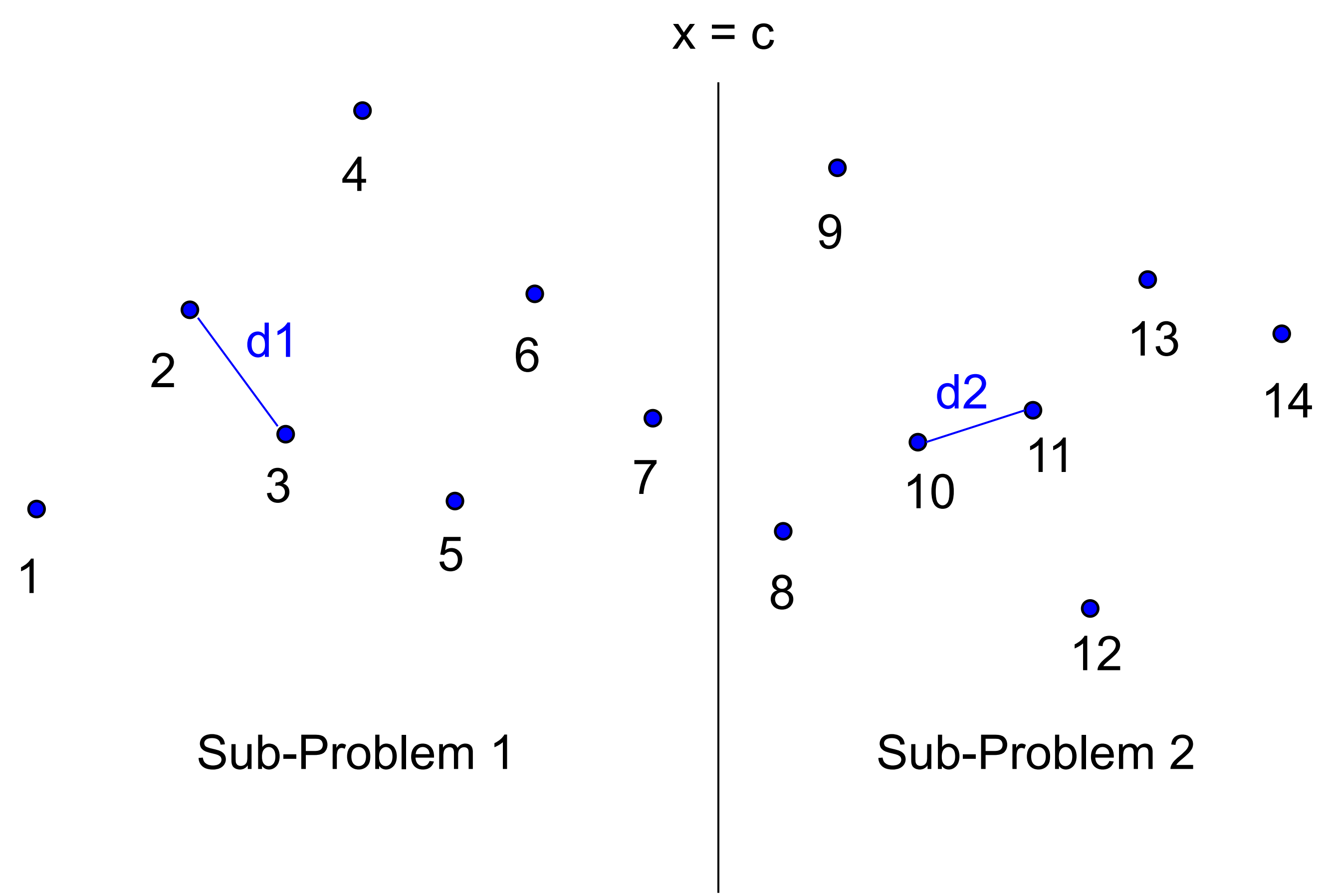
Closest-Pair Problem by Divide-and-Conquer

Advantage: With just one split we cut the number of comparisons in half. Obviously, we gain an even greater advantage if we split the sub-problems.



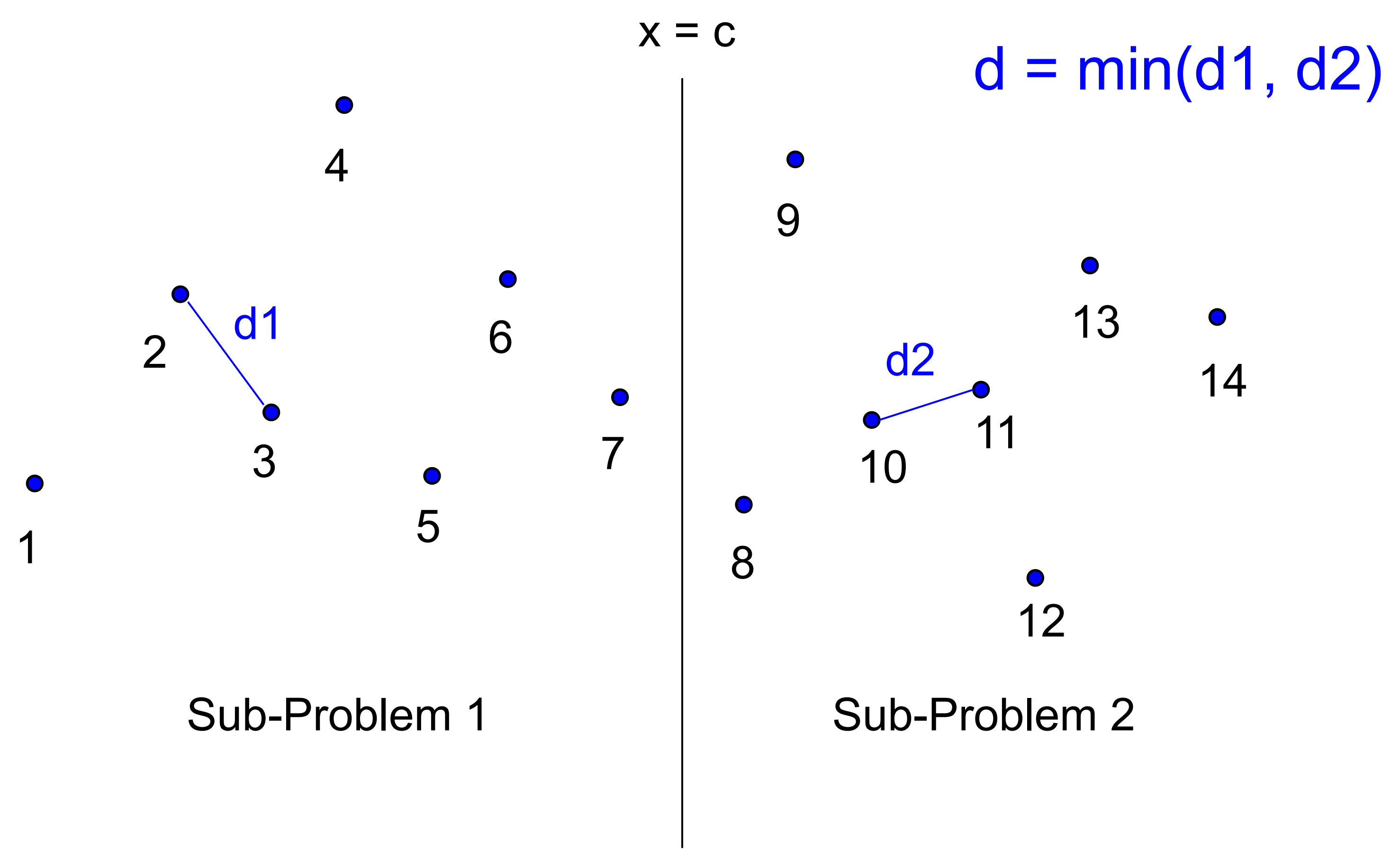
Closest-Pair Problem by Divide-and-Conquer

- Normally, we'd have to compare each of the 14 points with every other point. $(n-1) \times n/2 = 13 \times 14/2$ **= 91 comparisons**
- **Advantage**: Now, we have two sub-problems of half the size. Thus, we have to do $6 \times 7/2$ comparisons twice, which is **42 comparisons**



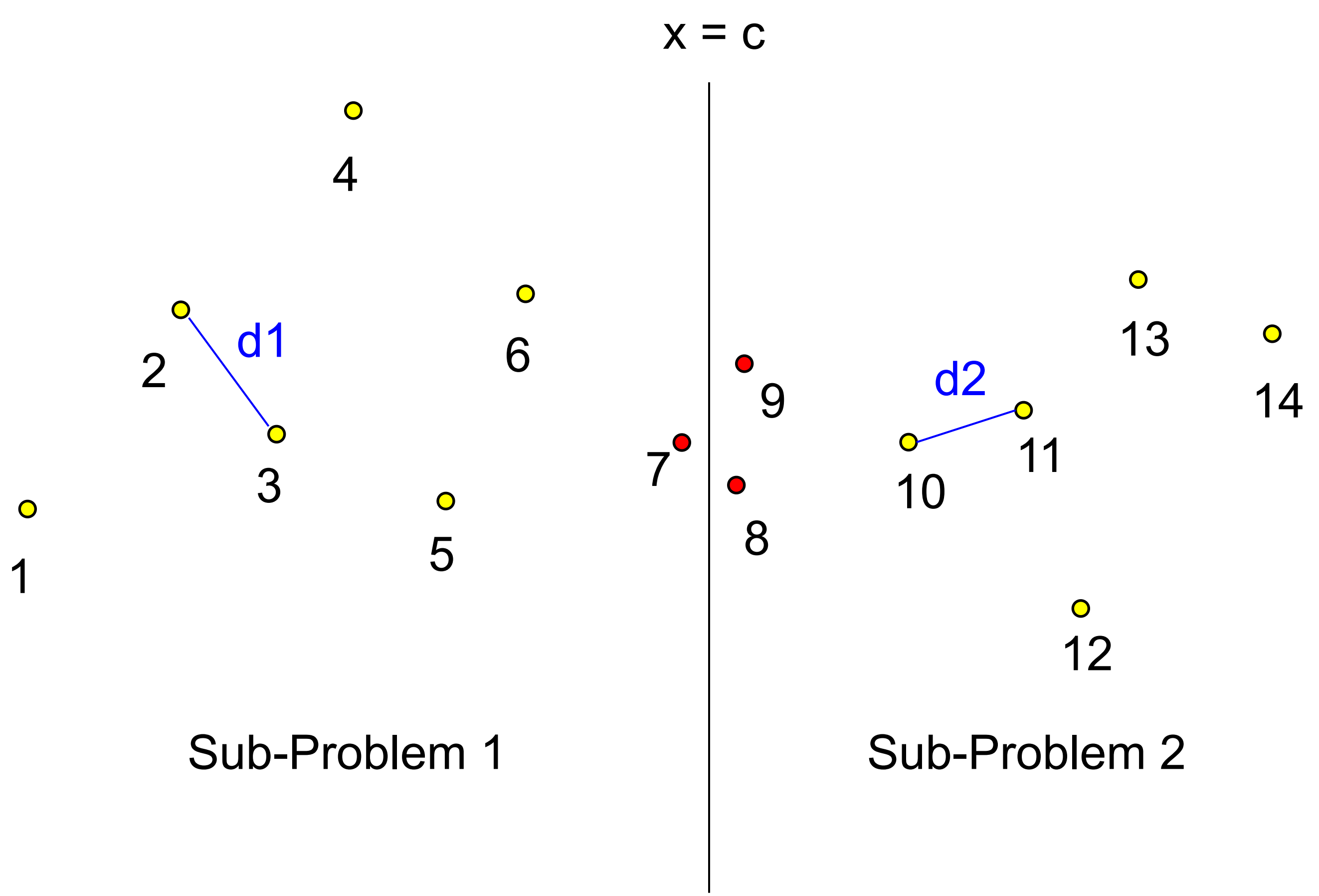
Closest-Pair Problem by Divide-and-Conquer

Problem: However, what if the closest two points are each from different sub-problems?



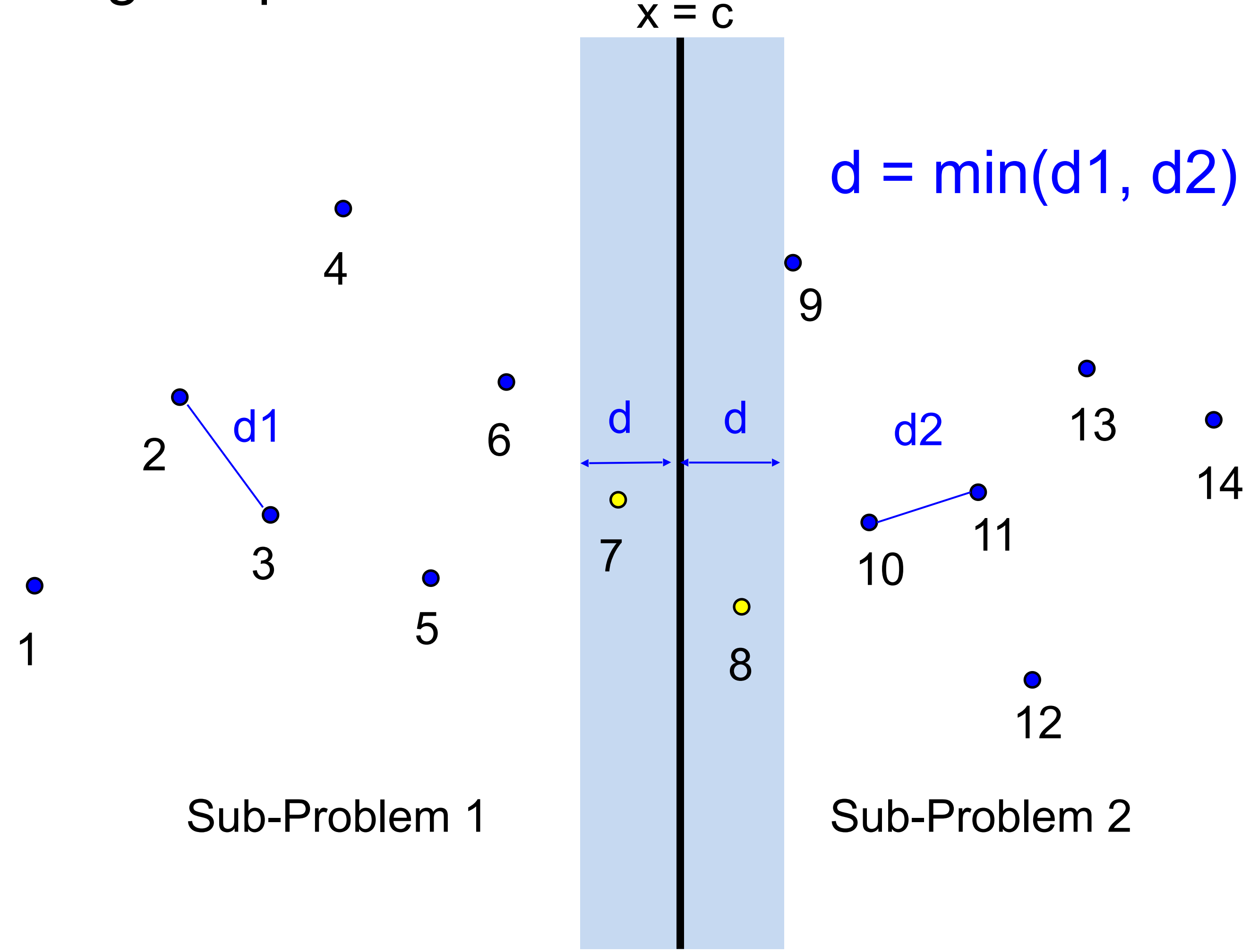
Closest-Pair Problem by Divide-and-Conquer

Here is an example where we have to compare points from sub-problem 1 to the points in sub-problem 2.



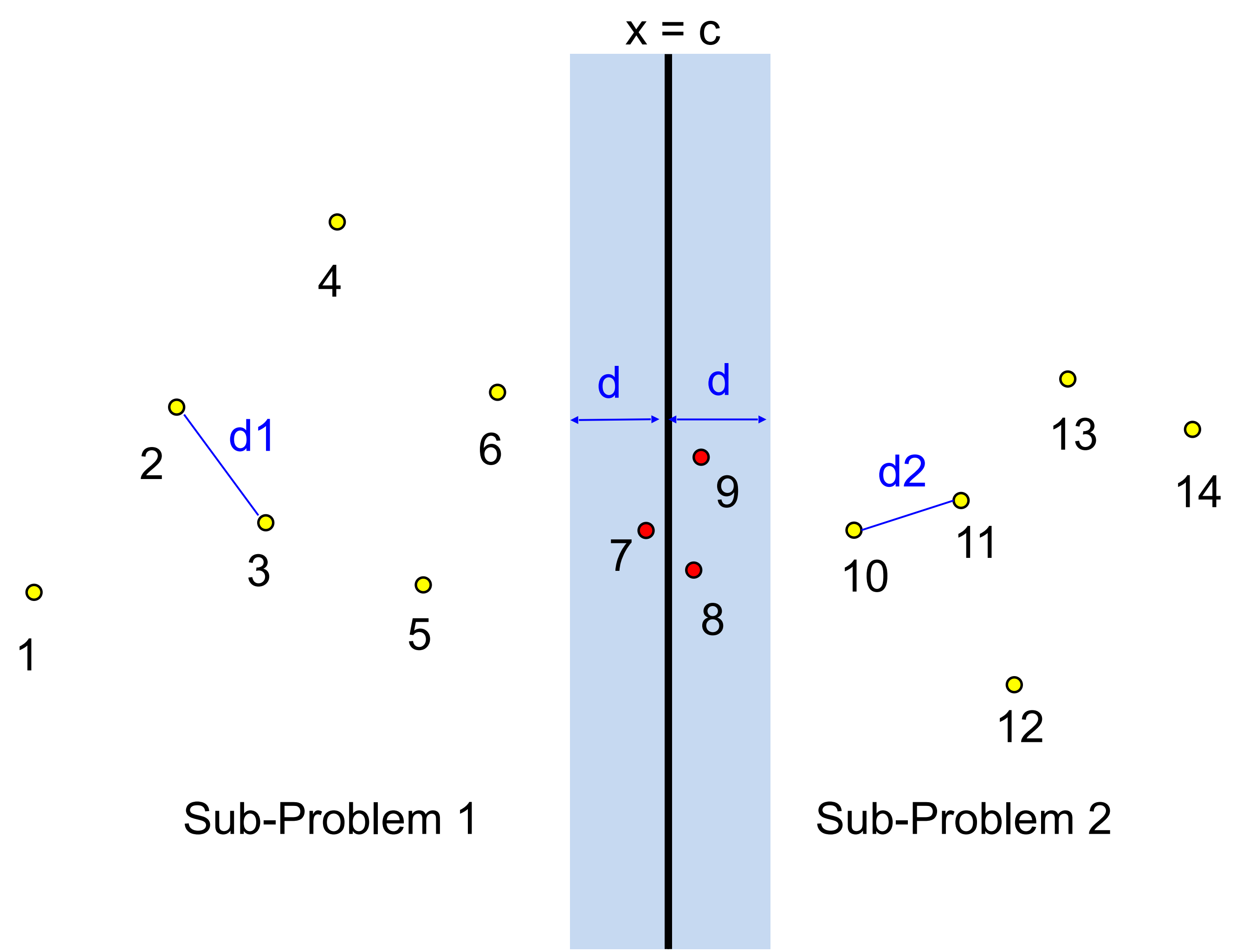
Closest-Pair Problem by Divide-and-Conquer

However, we only have to compare points inside the following “strip.”



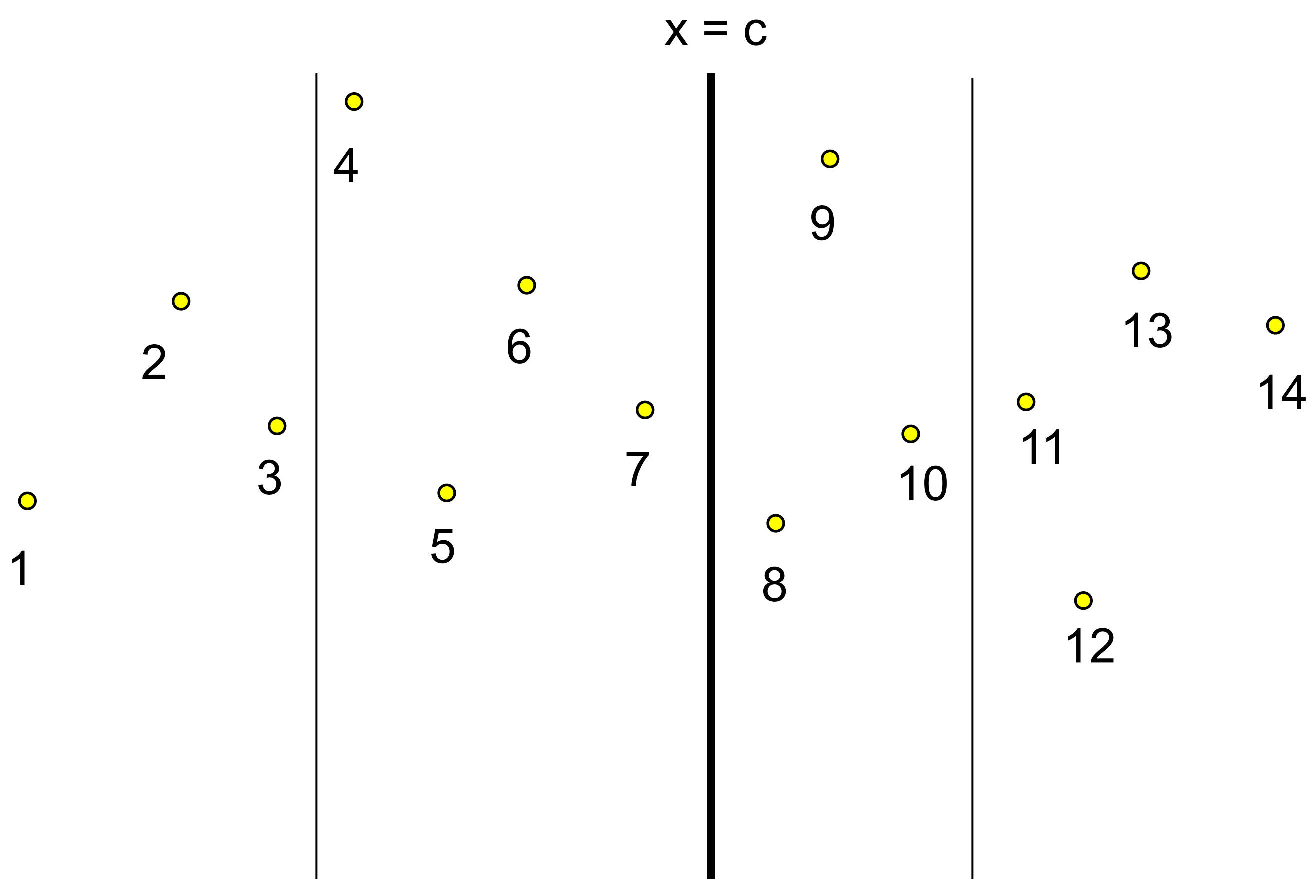
Closest-Pair Problem by Divide-and-Conquer

Distance between point 7 and 8 is less than d2.



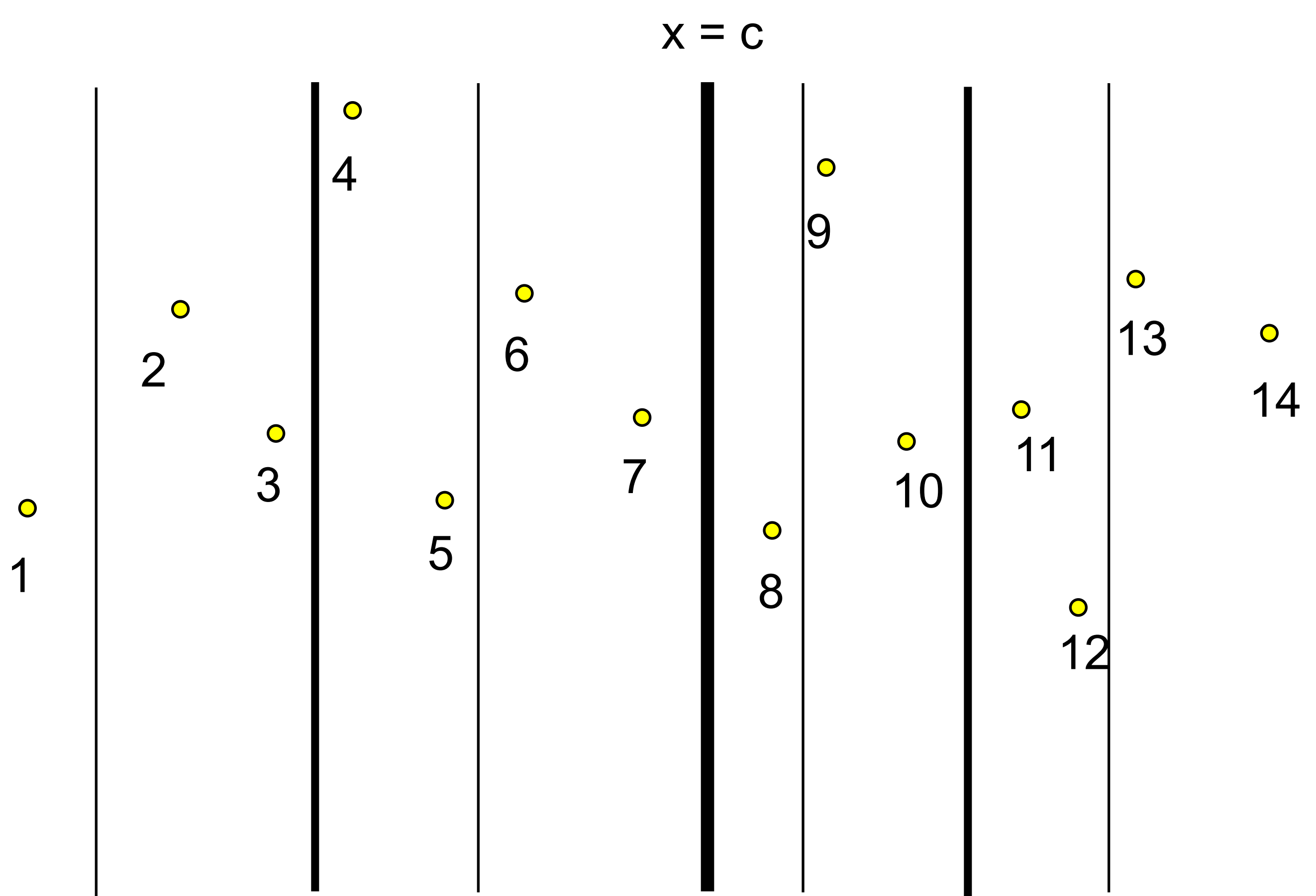
Closest-Pair Problem by Divide-and-Conquer

3. But, we can continue the advantage by splitting the sub-problems.



Closest-Pair Problem by Divide-and-Conquer

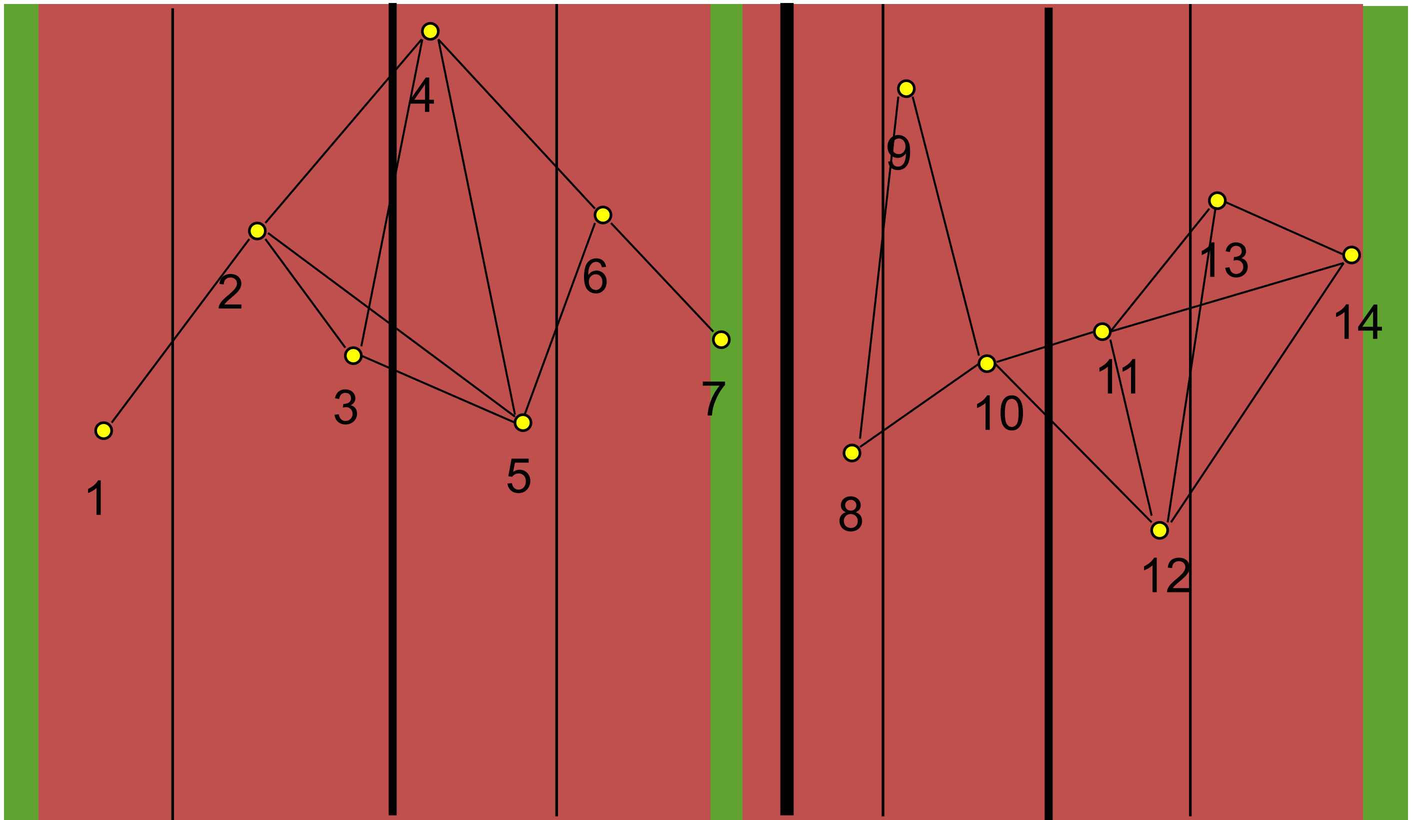
In fact we can continue to split until each sub-problem is trivial, i.e., takes one comparison.



Closest-Pair Problem by Divide-and-Conquer

4. The solution to each sub-problem is combined until the final solution is obtained

x = c



Pseudocode of the Closest-Pair Algorithm

ALGORITHM *EfficientClosestPair*(P, Q)

```
if n ≤ 3
    return the minimal distance found by the brute-force algorithm
else
    copy the first ⌈n/2⌉ points of P to array P_l
    copy the same ⌈n/2⌉ points from Q to array Q_l
    copy the remaining ⌊n/2⌋ points of P to array P_r
    copy the same ⌊n/2⌋ points from Q to array Q_r
    d_l ← EfficientClosestPair(P_l, Q_l)
    d_r ← EfficientClosestPair(P_r, Q_r)
    d ← min{d_l, d_r}
    m ← P[⌈n/2⌉ − 1].x
    copy all the points of Q for which |x − m| < d into array S[0..num − 1]
    dminsq ← d^2
    for i ← 0 to num − 2 do
        k ← i + 1
        while k ≤ num − 1 and (S[k].y − S[i].y)^2 < dminsq
            dminsq ← min((S[k].x − S[i].x)^2 + (S[k].y − S[i].y)^2, dminsq)
            k ← k + 1
return sqrt(dminsq)
```

Efficiency of the Closest-Pair Algorithm

The algorithm spends linear time both for dividing the problem into two problems half the size and combining the obtained solutions. Therefore, assuming as usual that n is a power of 2, we have the following recurrence for the running time of the algorithm:

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

where $f(n) \in \Theta(n)$. Applying the Master Theorem (with $a = 2$, $b = 2$, and $d = 1$), we get $T(n) \in \Theta(n \log n)$. The necessity to presort input points does not change the overall efficiency class if sorting is done by a $O(n \log n)$ algorithm such as mergesort. In fact, this is the best efficiency class one can achieve, because it has been proved that any algorithm for this problem must be in $\Omega(n \log n)$ under some natural assumptions about operations an algorithm can perform (see [Pre85, p. 188]).

Efficiency of the Closest-Pair Algorithm

Running time of the algorithm is described by

$$T(n) = 2T(n/2) + \underline{M(n)}, \quad \text{where } M(n) \in O(6*n/2) \in O(n)$$

Time required to merge the solutions from sub-problems

By the Master Theorem (with $a = 2$, $b = 2$, $d = 1$)

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

Review: Closest-pair problem using brute force approach

- Brute force approach requires comparing every point with every other point
- Given n points, we must perform (n-1) + (n-2) + …. + 3 + 2 + 1 comparisons.

$$\sum_{k=1}^{n-1} k = \frac{(n-1)\cdot n}{2}$$

- Brute force → O(n^2)

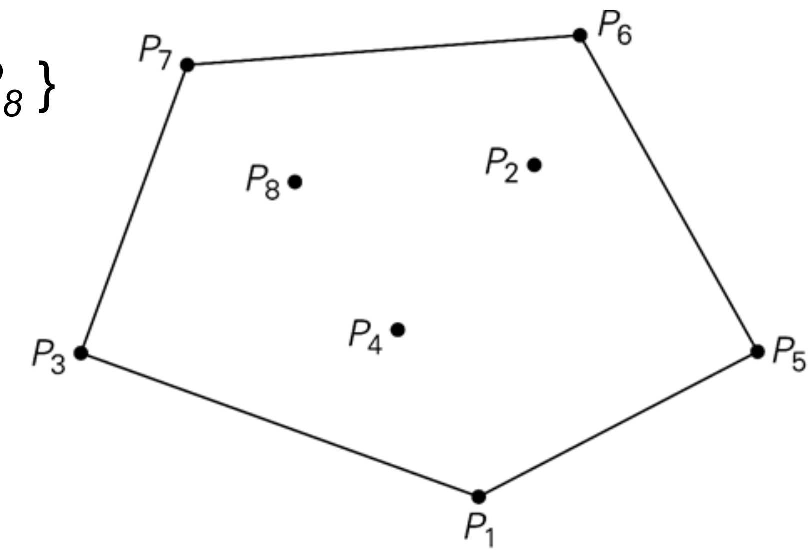
- The Divide and Conquer algorithm yields → O($n \log n$)
- Reminder: if n = 1,000,000 then
  - `n² = 1,000,000,000,000 whereas`
  - `n logn = 20,000,000`

5.5 Convex-Hull Problem

- Recall the Brute-Force Convex-Hull Algorithm

- Find the pairs of points (P_i, P_j) from a set of n points
  - The line segment connecting P_i *and* P_j is a part of its convex hull's boundary if and only if the other points of the set lie on the same side of the straight line through P_i *and* P_j

- Input: A set S of planar points
  - $S = \{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8\}$
- Output : A convex hull for S
  - $\{P_1, P_3, P_5, P_6, P_7\}$



Quickhull Algorithm

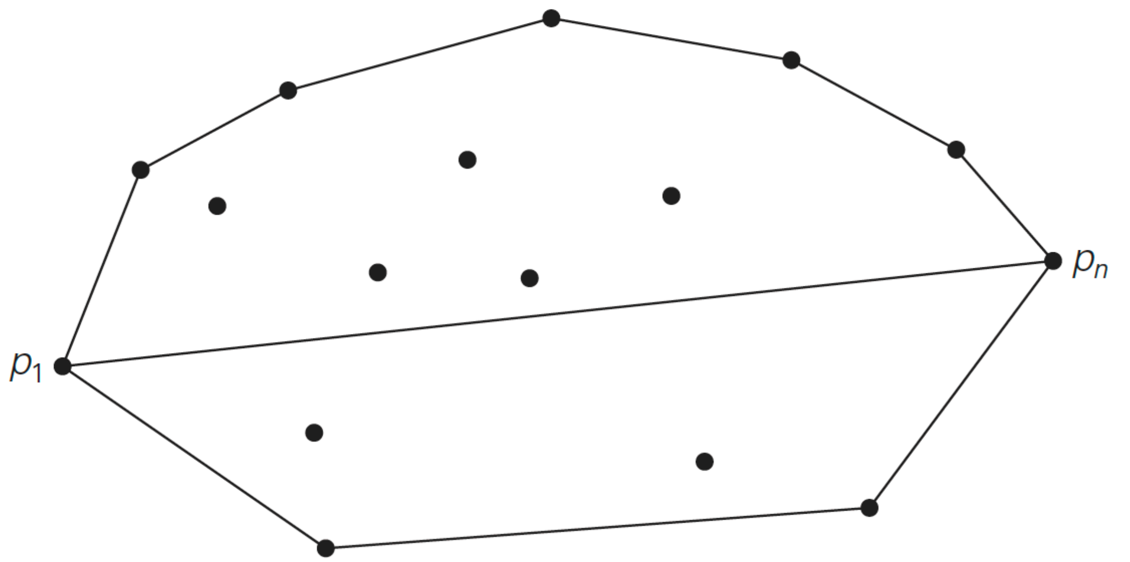
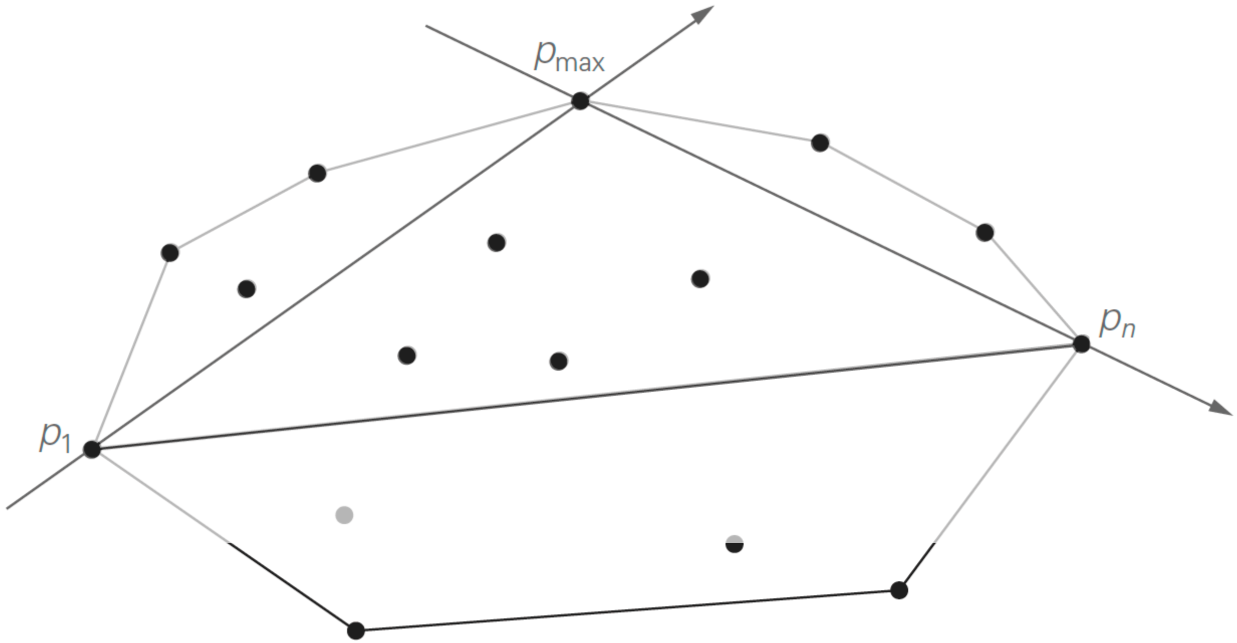


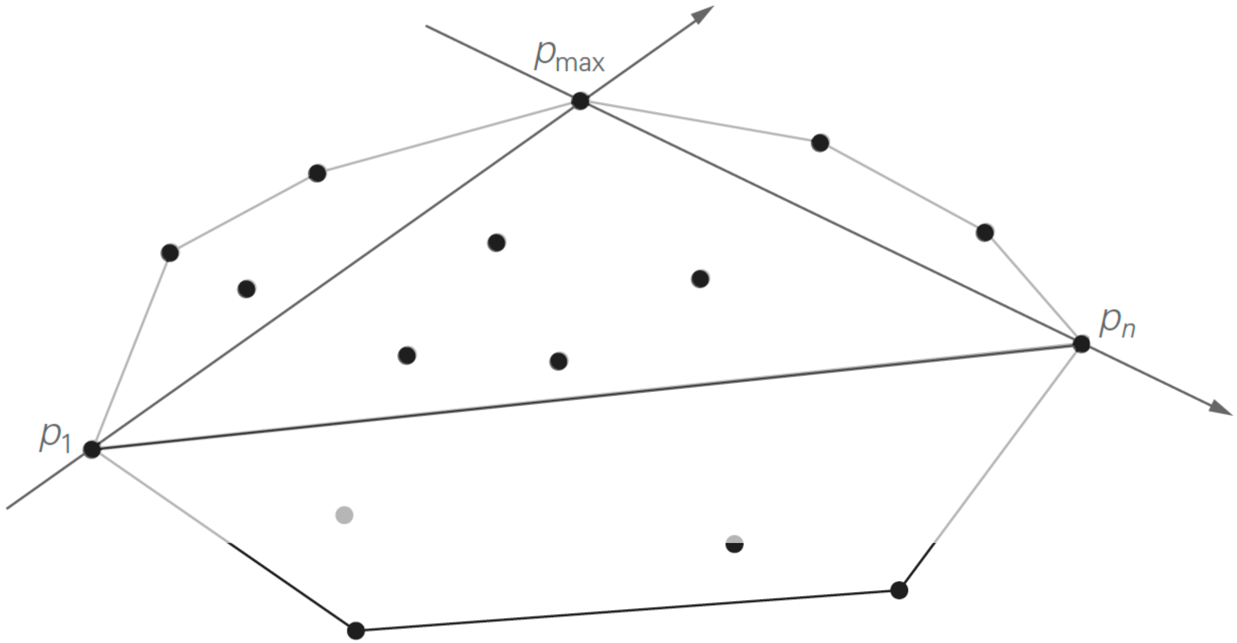
FIGURE 5.8 Upper and lower hulls of a set of points.

Quickhull Algorithm



- $p_{\max}$ is a vertex of the upper hull.
- The points inside $\triangle p_1 p_{\max} p_n$ cannot be vertices of the upper hull (and hence can be eliminated from further consideration).
- There are no points to the left of both lines $\overrightarrow{p_1 p_{\max}}$ and $\overrightarrow{p_{\max} p_n}$.

Quickhull Algorithm

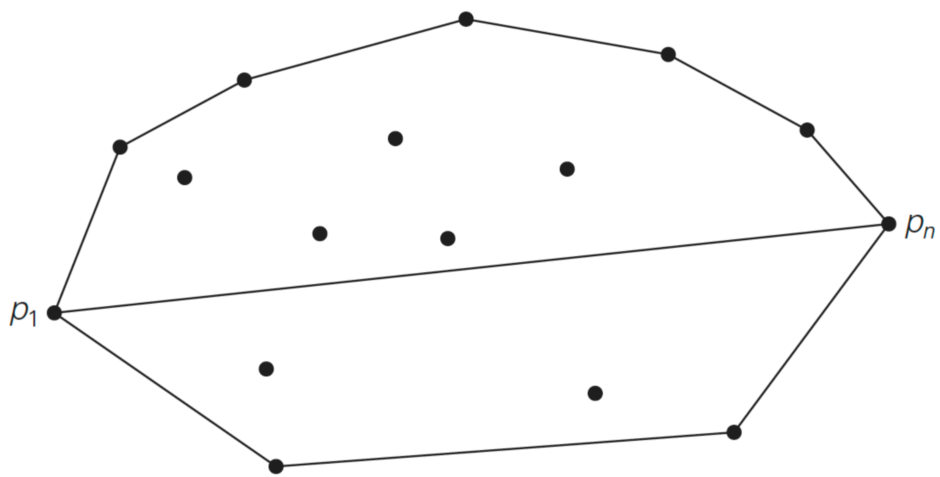


Convex hull: smallest convex set that includes given points

- find point P_{max} that is farthest away from line P_1P_2
- compute the upper hull of the points to the left of line P_1P_{max}
- compute the upper hull of the points to the left of line $P_{max}P_2$

Quickhull Algorithm

- Assume points are sorted by *x*-coordinate values
- Identify *extreme points* P_1 and P_2 (leftmost and rightmost)
- Compute *upper hull* recursively:
  - find point P_{max} that is farthest away from line P_1P_2
  - compute the upper hull of the points to the left of line P_1P_{max}
  - compute the upper hull of the points to the left of line $P_{max}P_2$
- Compute *lower hull* in a similar manner



Quickhull Algorithm

If $q_1(x_1, y_1)$, $q_2(x_2, y_2)$, and $q_3(x_3, y_3)$ are three arbitrary points in the Cartesian plane, then the area of the triangle $\triangle q_1q_2q_3$ is equal to one-half of the magnitude of the determinant

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 + x_3y_1 + x_2y_3 - x_3y_2 - x_2y_1 - x_1y_3,$$

Using this formula, we can check in constant time whether a point lies to the left of the line determined by two other points as well as find the distance from the point to the line.

Efficiency of Quickhull Algorithm

- Finding point farthest away from line P_1P_2 can be done in linear time
- Time efficiency:
  - worst case: $\Theta(n^2)$ (as quicksort)
  - average case: $\Theta(n)$ (under reasonable assumptions about distribution of points given)

- If points are not initially sorted by *x*-coordinate value, this can be accomplished in O(n log n) time

- Several O(n log n) algorithms for convex hull are known

Exercise 5.5

7. Explain how one can find point $p_{\max}$ in the quickhull algorithm analytically.