Bruteforce

V. Balasubramanian
SSN College of Engineering

Brute Force Algorithms

- String Matching
- Closest Pair Problem
- Exhaustive search: TSP, Knapsack Problem

Sequential Search

ALGORITHM *SequentialSearch2*$(A[0..n], K)$

//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in $A[0..n-1]$ whose value is
// equal to K or -1 if no such element is found

$A[n] \leftarrow K$
$i \leftarrow 0$
while $A[i] \neq K$ **do**
  $i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

Sequential Search

- Sequential search provides an excellent illustration of the brute-force approach, with its characteristic strength (simplicity) and weakness (inferior efficiency).
- If elements are sorted, then Binary search can be employed

String Matching

- Pattern: a string of m characters to search for
- Text: a (longer) string of n characters to search in
- problem: find a substring in the text that matches the pattern

Brute-force algorithm

- Step 1: Align pattern at beginning of text
- Step 2: Moving from left to right, compare each character of pattern to the corresponding character in text until all characters are found to match (successful search); or a mismatch is detected
- Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

//Implements brute-force string matching
//Input: An array $T[0..n-1]$ of n characters representing a text and
//   an array $P[0..m-1]$ of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//   matching substring or -1 if the search is unsuccessful
for $i \leftarrow 0$ **to** $n-m$ **do**
    $j \leftarrow 0$
    while $j < m$ **and** $P[j] = T[i+j]$ **do**
        $j \leftarrow j+1$
    if $j = m$ **return** i
return -1

Match

$$
\begin{array}{cccccccccc}
t_0 & \cdots & t_i & \cdots & t_{i+j} & \cdots & t_{i+m-1} & \cdots & t_{n-1} & \text{text } T \\
 & & \updownarrow & & \updownarrow & & \updownarrow & & & \\
 & & p_0 & \cdots & p_j & \cdots & p_{m-1} & \text{pattern } P & &
\end{array}
$$

Complexity

- Note that for this example, the algorithm shifts the pattern almost always after a single character comparison.
- The worst case is much worse: the algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the $n - m + 1$ tries.
- In the worst case, the algorithm makes $m(n - m + 1)$ character comparisons, which puts it in the $O(nm)$ class

N	O	B	O	D	Y	_	N	O	T	I	C	E	D	_	H	I	M
N	**O**	**T**															
	N	O	T														
		N	O	T													
			N	O	T												
				N	O	T											
					N	O	T										
						N	O	T									
							N	**O**	**T**								

FIGURE 3.3 Example of brute-force string matching. (The pattern's characters that are compared with their text counterparts are in bold type.)

Example

text T
a b c a b a a b c a b a c
pattern P
$s = 3$
a b a a

a c a a b c
$s = 0$
a a b
a c a a b c
$s = 1$
a a b
a c a a b c
$s = 2$
a a b
a c a a b c
$s = 3$
a a b

Determine the number of character comparisons made by the brute-force algorithm in searching for the pattern SEARCH in the text

SORTING_ALGORITHM_CAN_USE_BRUTE_FORCE_METHOD

Assume that the length of the text—it is 43 characters long—is known before the search starts.

Example

How many comparisons (both successful and unsuccessful) will be made by the brute-force algorithm in searching for each of the following patterns in the binary text of one million zeros?

a. 01001 **b.** 00010 **c.** 01011

Matching

one position to the right:

0	0	0	0	0	0	0	0			0	0	0	0	0	0	0	0
0	1	0	0	1													
	0	1	0	0	1												
						etc.											
													0	1	0	0	1

The total number of character comparisons will be $c = 2 * 9, 99, 996$.

Example

Give an example of a text of length n and a pattern of length m that constitutes a worst-case input for the brute-force string-matching algorithm. Exactly how many character comparisons will be made for such input?

Comparisons

The text composed of n zeros and the pattern $\underbrace{0 \ldots 0}_{m-1}1$ is an example of the worst-case input. The algorithm will make $m(n - m + 1)$ character comparisons on such input.

Boyer Moore Horspool Algorithm

- Example:

Pattern	‘tooth’
Text	‘trusthardtoothbrushes’

Match Table Pre Processing

- Construct Bad Match Table

  Value = length – index – 1   (Every other letter = length)

T O O T H

0 1 2 3 4

Length = 5

Letter	T	O	H	*
Value				

Pre-Processing

T = 5 - 0 - 1

O = 5 - 1 - 1

Letter	T	O	H	*
Value	1	2	5	5

O = 5 - 2 - 1

T = 5 - 3 - 1

H = 5 Last letter = length if not already defined, else, leave

Matching

T R U S T H A R D T O O T H B R U S H E S

T O O T H

T R U S T H A R D T O O T H B R U S H E S

↑

T O O T H

Contd…

T R U S T H A R D T O O T H B R U S H E S

T O O T H

T R U S T H A R D T O O T H B R U S H E S

T O O T H

Contd...

T R U S T H A R D T O O T H B R U S H E S

T O O T H

T R U S T H A R D T O O T H B R U S H E S

T O O T H

Complexity

Worst case same as naïve example:

- 1^n input text (length n)
- 0111...1 pattern (length m)

Worst Case O (nm)

Best case

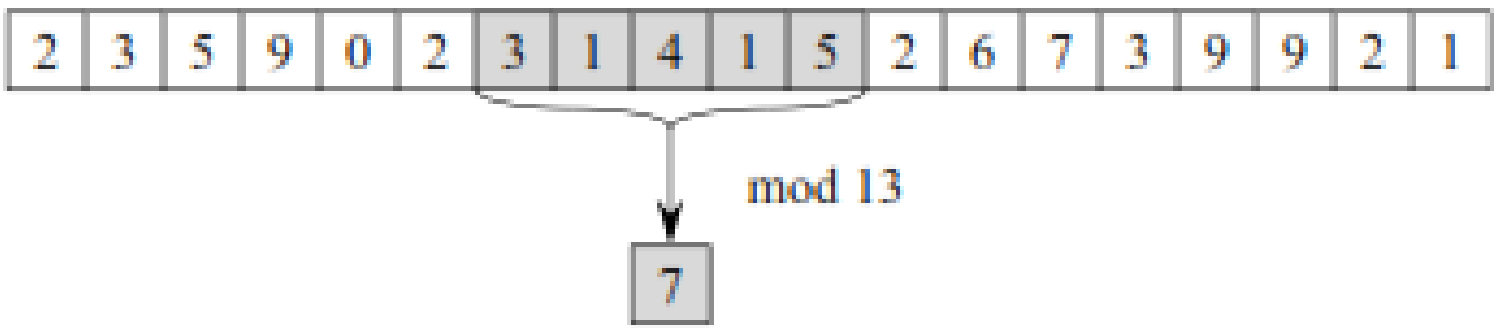
- 1^n input text (length n)
- 0^m pattern (length m)

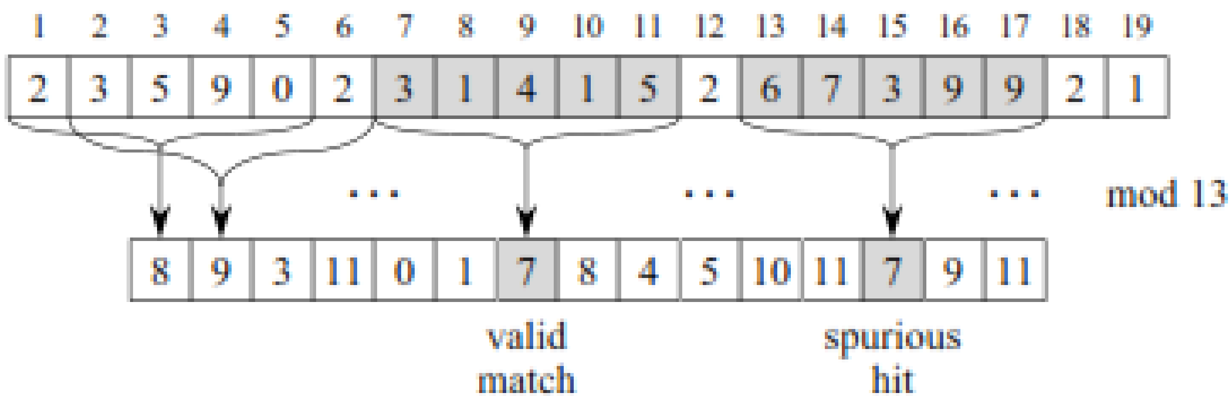
Rabin Karp Algorithm

Algorithm	Preprocessing time	Matching time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$

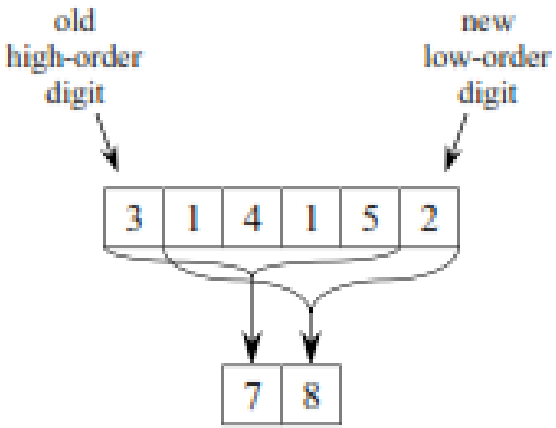
Given a pattern $P[1..m]$, let p denote its corresponding decimal value. In a similar manner, given a text $T[1..n]$, let t_s denote the decimal value of the length-m substring $T[s+1..s+m]$, for $s = 0, 1, \ldots, n-m$. Certainly, $t_s = p$ if and only if $T[s+1..s+m] = P[1..m]$; thus, s is a valid shift if and only if $t_s = p$. If we could compute p in time $\Theta(m)$ and all the t_s values in a total of $\Theta(n-m+1)$ time,[1]

Preprocessing





Preprocessing



$$\begin{aligned}14152 &\equiv (31415 - \overbrace{3}^{\text{old high-order digit}} \cdot 10000) \cdot \overbrace{10}^{\text{shift}} + \overbrace{2}^{\text{new low-order digit}} \pmod{13}\\ &\equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13}\\ &\equiv 8 \pmod{13}\end{aligned}$$

$$\begin{aligned} t_{s+1} &= 10(31415 - 10000 \cdot 3) + 2\\ &= 14152\,. \end{aligned}$$

Example

Working modulo $q = 11$, how many spurious hits does the Rabin-Karp matcher encounter in the text $T = 3141592653589793$ when looking for the pattern $P = 26$?

Solution

- Three spurious hits, $15 \equiv 59 \equiv 92 \equiv 26 \equiv 4 \mod 11$

Closest Pair Problem

- Find the two closest points in a set of n points (in the two-dimensional Cartesian plane).
- Brute-force algorithm
- Compute the distance between every pair of distinct points and return the indexes of the points for which the distance is the smallest.

Closest Pair Problem

ALGORITHM *BruteForceClosestPoints*(P)

//Finds two closest points in the plane by brute force
//Input: A list P of n ($n \geq 2$) points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$
//Output: Indices $index1$ and $index2$ of the closest pair of points

$dmin \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  for $j \leftarrow i + 1$ **to** n **do**
    $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ //$sqrt$ is the square root function
    if $d < dmin$
      $dmin \leftarrow d$; $index1 \leftarrow i$; $index2 \leftarrow j$
return $index1$, $index2$

Strictly increasing function

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 = 2 \sum_{i=1}^{n-1} (n-i)$$

$$= 2[(n-1) + (n-2) + \cdots + 1] = (n-1)n \in \Theta(n^2).$$

Bruteforce algorithms

- Strengths
  - wide applicability
  - simplicity
  - yields reasonable algorithms for some important problems
    (e.g., matrix multiplication, sorting, searching, string matching)
- Weaknesses
  - rarely yields efficient algorithms
  - some brute-force algorithms are unacceptably slow
  - not as constructive as some other design techniques

Exhaustive Search

- A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as permutations, combinations, or subsets of a set.

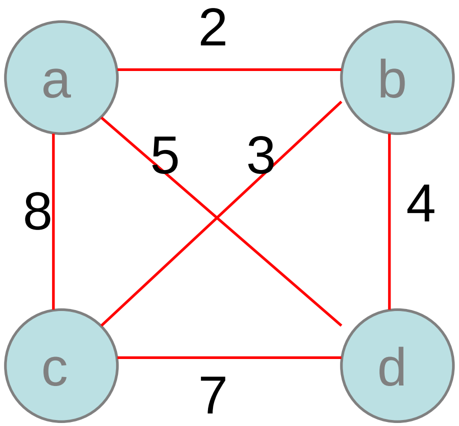
Method

Method:

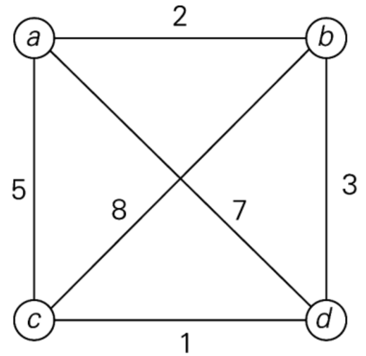
- generate a list of all potential solutions to the problem in a systematic
- evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
- when search ends, announce the solution(s) found

TSP

- Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city
- Alternatively: Find shortest *Hamiltonian circuit* in a weighted connected graph


2
a
b
5
3
8
4
c
d
7

TSP



Tour	Length	
$a \longrightarrow b \longrightarrow c \longrightarrow d \longrightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \longrightarrow b \longrightarrow d \longrightarrow c \longrightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \longrightarrow c \longrightarrow b \longrightarrow d \longrightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \longrightarrow c \longrightarrow d \longrightarrow b \longrightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \longrightarrow d \longrightarrow b \longrightarrow c \longrightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \longrightarrow d \longrightarrow c \longrightarrow b \longrightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

FIGURE 3.7 Solution to a small instance of the traveling salesman problem by exhaustive search

Knapsack

Given n items:

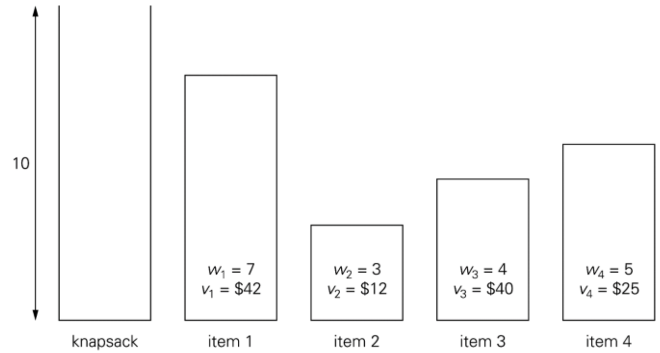
- weights: $w_1 \quad w_2 \; \ldots \; w_n$
- values: $v_1 \quad v_2 \; \ldots \; v_n$
- a knapsack of capacity W

Find most valuable subset of the items that fit into the knapsack

Example: Knapsack capacity W=16

item	weight	value
1	2	$20
2	5	$30
3	10	$50
4	5	$10

Knapsack



(a)

Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$36
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

(b)

FIGURE 3.8 (a) Instance of the knapsack problem. (b) Its solution by exhaustive search. (The information about the optimal selection is in bold.)

Assignment Problem

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4>	cost = 9 + 4 + 1 + 4 = 18
<1, 2, 4, 3>	cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4>	cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2>	cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3>	cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2>	cost = 9 + 7 + 1 + 6 = 23

etc.

FIGURE 3.9 First few iterations of solving a small instance of the assignment problem by exhaustive search

Assignment problem

There are *n* people who need to be assigned to *n* jobs, one person per job. The cost of assigning person *i* to job *j* is C[*i*,*j*]. Find an assignment that minimizes the total cost.

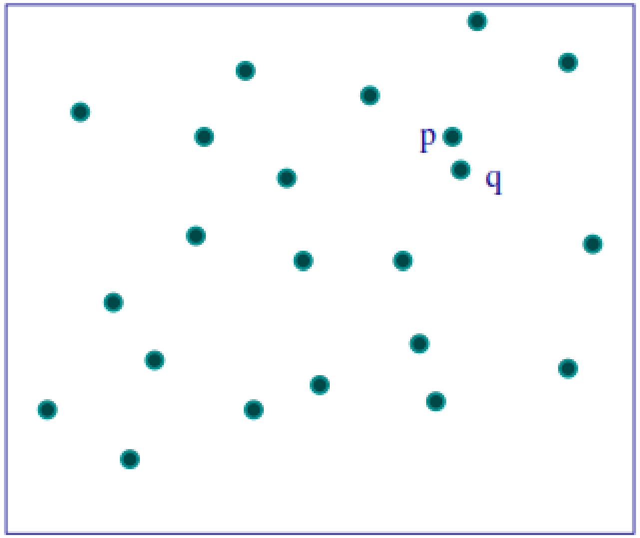
Exhaustive Search

- Exhaustive-search algorithms run in a realistic amount of time only on very small instances
- In some cases, there are much better alternatives!
  - Euler circuits
  - shortest paths
  - minimum spanning tree
  - assignment problem

Divide and Conquer / Dynamic Programming

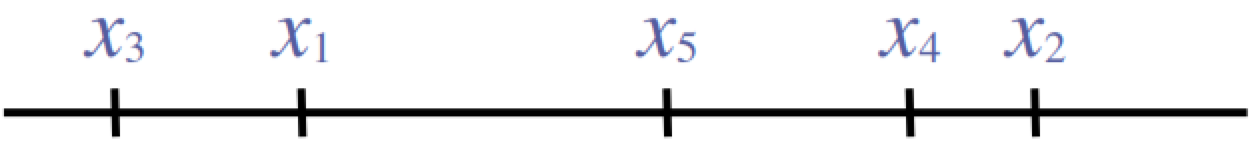
Presentation by
V. Balasubramanian
SSN College of Engineering

Closest Pair Problem



- Complexity = n^2
- For example, an air-traffic controller might be interested in two closest planes as the most probable collision candidates.
- A regional postal service manager - closest pair problem to find candidate post-office locations to be closed.

1D solution



Algorithm

- If n>3, we can divide the points into two subsets P_l and P_r of n/2 and n/2 points, respectively, by drawing a vertical line through the median m of their x coordinates so that n/2 points lie to the left of or on the line itself, and n/2 points lie to the right of or on the line. Then we can solve the closest-pair problem

Algorithm

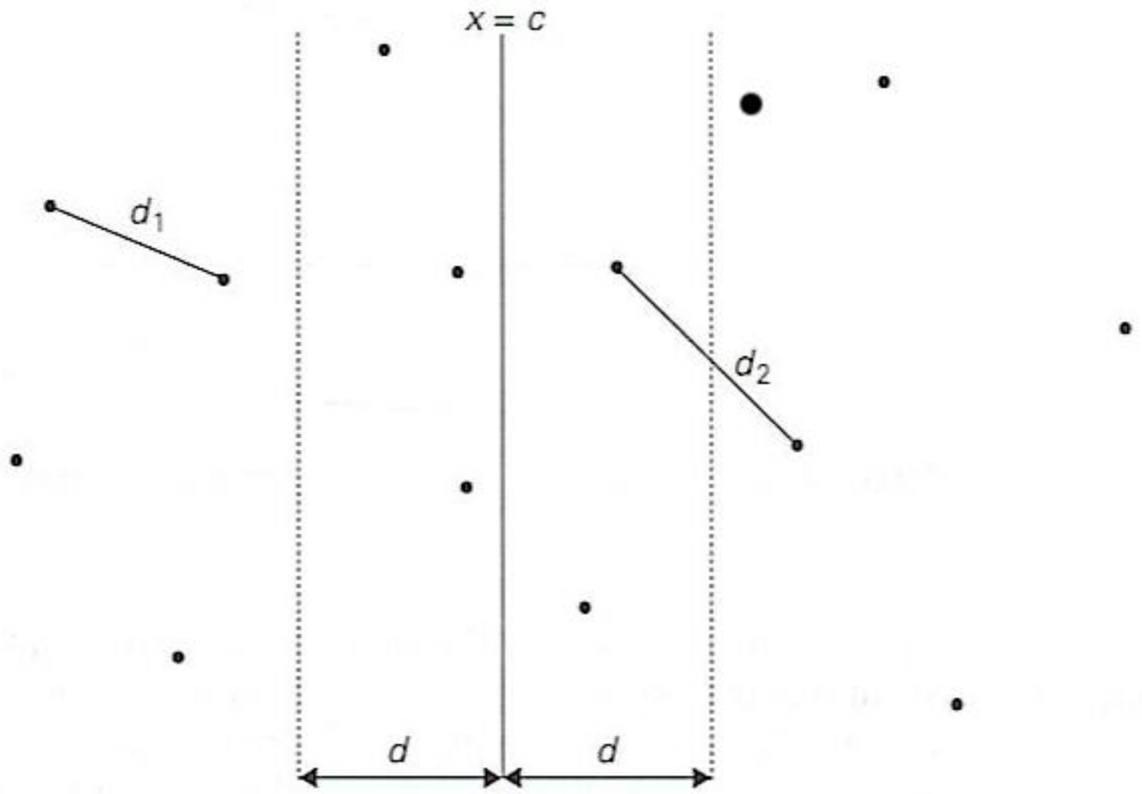
The algorithm:

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

Steps

Step 1 Divide the points given into two subsets S_1 and S_2 by a vertical line $x = c$ 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.

Contd…



Step 2

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

Step 3

- Step 3:Set d = min{d1, d2}

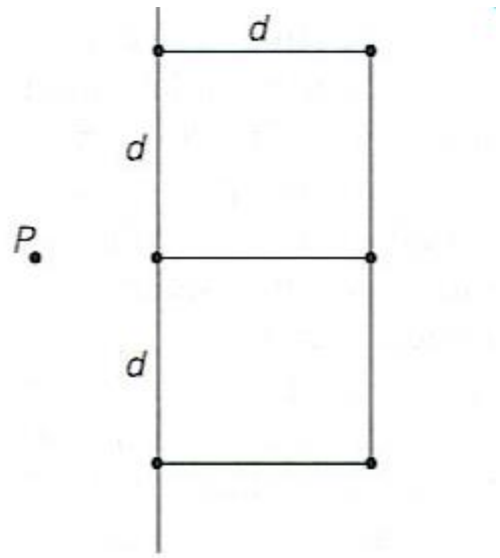
We can limit our attention to the points in the symmetric vertical strip of width 2d as possible closest pair. Let C1 and C2 be the subsets of points in the left subset S1 and of the right subset S2, respectively, that lie in this vertical strip. The points in C1 and C2 are stored in increasing order of their y coordinates, which is maintained by merging during the execution of the next step.

Step 4

- Step 4:

For every point $P(x,y)$ in C_1, we inspect points in C_2 that may be closer to P than d. There can be no more than 6 such points (because $d \leq d_2$)!

Worst Case Scenario



Contd…

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

Step 2: If S contains only one point, return infinity as its distance.

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

Contd…

- Step 4: Recursively apply Steps 2 and 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 $d = \min(d_L, d_R)$.

Contd…

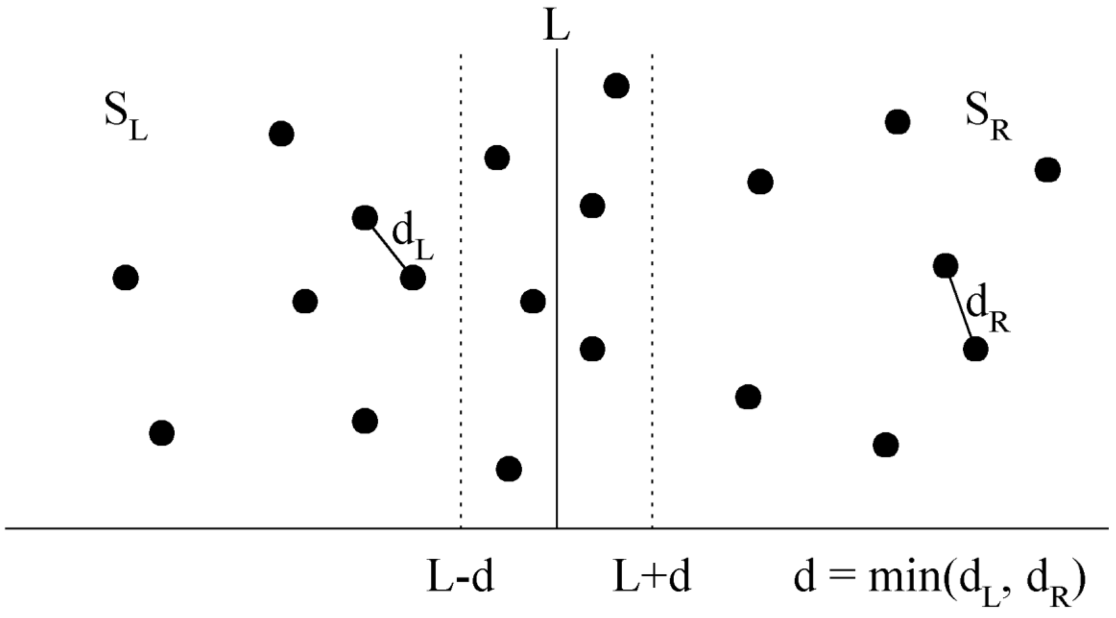
- Step 5: For a point P in the half-slab bounded by L-d and L, let its y-value be denoted as y_P . For each such P, find all points in the half-slab bounded by L and L+d whose y-value fall within y_P+d and y_P-d. If the distance d' between P and a point in the other half-slab is less than d, let $d=d'$.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

ALGORITHM *EfficientClosestPair(P, Q)*

```
//Solves the closest-pair problem by divide-and-conquer
//Input: An array P of n ≥ 2 points in the Cartesian plane sorted in
//        nondecreasing order of their x coordinates and an array Q of the
//        same points sorted in nondecreasing order of the y coordinates
//Output: Euclidean distance between the closest pair of points
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)
```

Contd…

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



L
S_L
S_R
d_L
d_R
L-d
L+d
$d = \min(d_L, d_R)$

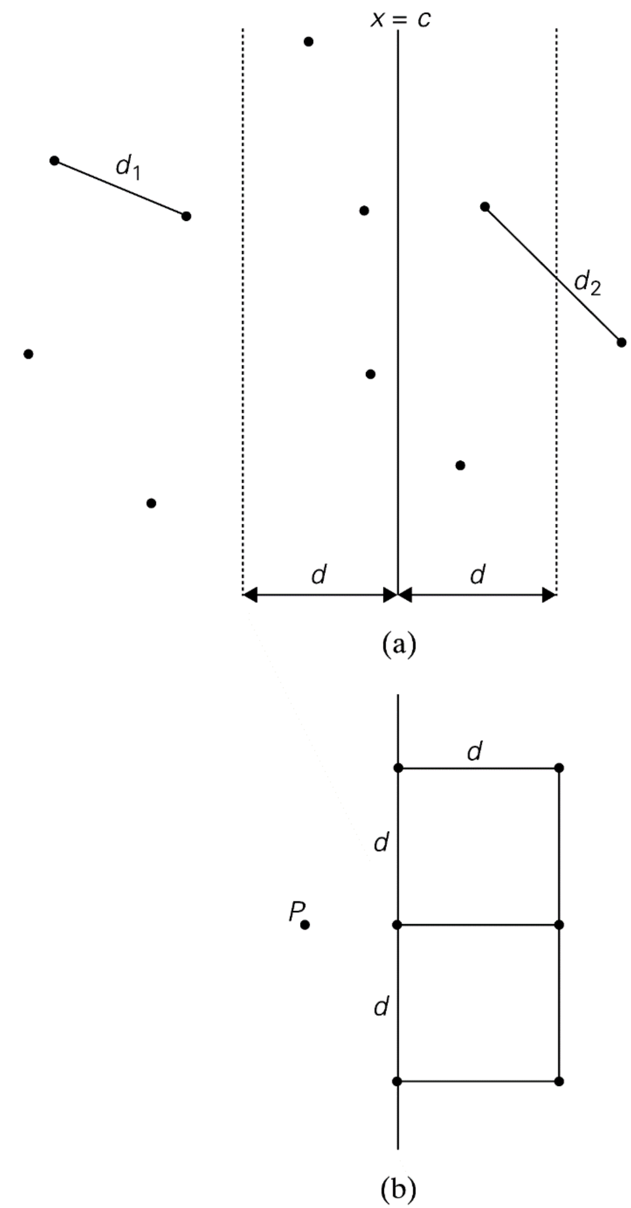


FIGURE 4.7 (a) Idea of the divide-and-conquer algorithm for the closest-pair problem. (b) The six points that may need to be examined for point P.

Analysis

Running time of the algorithm is described by

$$T(n) = 2T(n/2) + M(n), \text{ where}$$

$M(n) \in O(n)$

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

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

Exercise

1. a. For the one-dimensional version of the closest-pair problem, i.e., for the problem of finding two closest numbers among a given set of n real numbers, design an algorithm that is directly based on the divide-and-conquer technique and determine its efficiency class.

   b. Is it a good algorithm for this problem?

Solution

- O(n log n)
- Without employing Divide and Conquer, we can sort it and do it.

  O(n log n)

Exercise

2. Prove that the divide-and-conquer algorithm for the closest-pair problem examines, for every point p in the vertical strip (see Figures 5.7a and 5.7b), no more than seven other points that can be closer to p than d_{min}, the minimum distance between two points encountered by the algorithm up to that point.