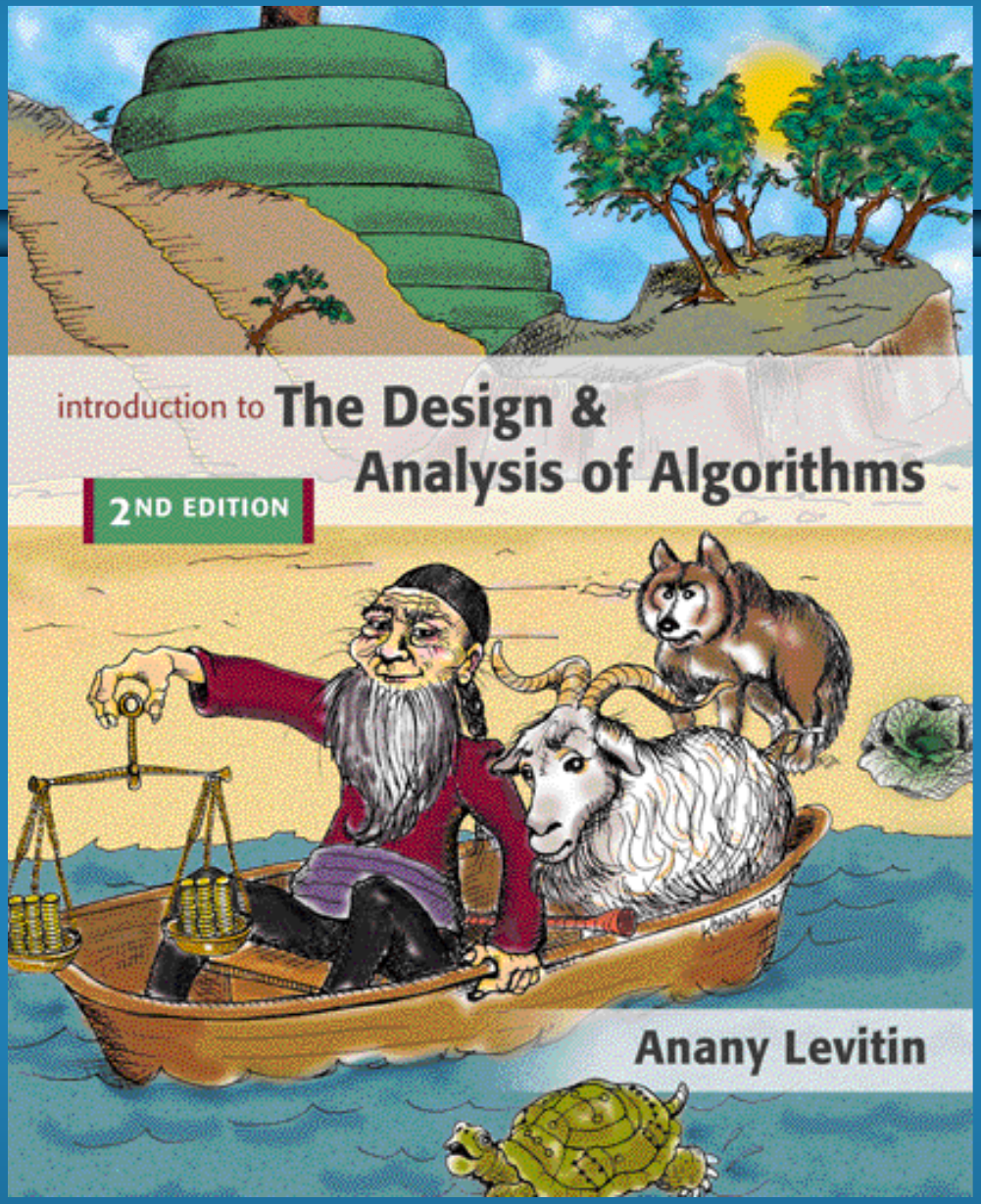
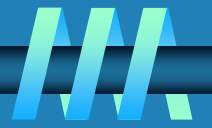
# Chapter 3

## Brute Force and Exhaustive Search



# Brute Force



**A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved**

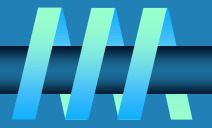
**Examples:**

1. **Computing $a^n$ ($a > 0$, $n$ a nonnegative integer)**

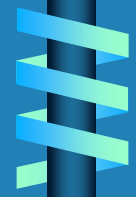
2. **the consecutive integer checking algorithm for computing** $\gcd(m, n)$

3. **Multiplying two matrices**

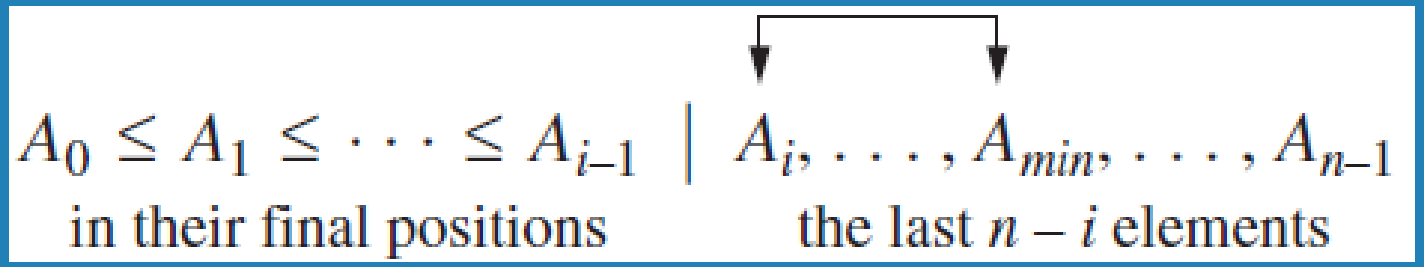
# Brute-Force Sorting Algorithm



- The two algorithms discussed here are:
  - Selection sort
  - Bubble sort



# Selection sort




| | | | | | | |
|---|---|---|---|---|---|---|
| 89 | 45 | 68 | 90 | 29 | 34 | **17** |
| 17 | 45 | 68 | 90 | **29** | 34 | 89 |
| 17 | 29 | 68 | 90 | 45 | **34** | 89 |
| 17 | 29 | 34 | 90 | **45** | 68 | 89 |
| 17 | 29 | 34 | 45 | 90 | **68** | 89 |
| 17 | 29 | 34 | 45 | 68 | 90 | **89** |
| 17 | 29 | 34 | 45 | 68 | 89 | 90 |

**FIGURE 3.1** Example of sorting with selection sort.

# Analysis of Selection Sort

**ALGORITHM** *SelectionSort*$(A[0..n-1])$

```
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]  min ← j
    swap A[i] and A[min]
```

$$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 =$$

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

- **The input size is n.**
- **The basic operation is comparison.**
- **The number of times it is executed $C(n) = \frac{n(n-1)}{2} \in \theta(n^2)$**

  **However, the number of key swaps is only $\theta(n)$**

# Bubble sort

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid A_{n-i} \leq \cdots \leq A_{n-1}$$

in their final positions

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 89 | $\overset{?}{\leftrightarrow}$ | 45 | | 68 | | 90 | | 29 | | 34 | | 17 |
| 45 | | 89 | $\overset{?}{\leftrightarrow}$ | 68 | | 90 | | 29 | | 34 | | 17 |
| 45 | | 68 | | 89 | $\overset{?}{\leftrightarrow}$ | 90 | $\overset{?}{\leftrightarrow}$ | 29 | | 34 | | 17 |
| 45 | | 68 | | 89 | | 29 | | 90 | $\overset{?}{\leftrightarrow}$ | 34 | | 17 |
| 45 | | 68 | | 89 | | 29 | | 34 | | 90 | $\overset{?}{\leftrightarrow}$ | 17 |
| 45 | | 68 | | 89 | | 29 | | 34 | | 17 | \| | 90 |
| 45 | $\overset{?}{\leftrightarrow}$ | 68 | $\overset{?}{\leftrightarrow}$ | 89 | $\overset{?}{\leftrightarrow}$ | 29 | | 34 | | 17 | \| | 90 |
| 45 | | 68 | | 29 | | 89 | $\overset{?}{\leftrightarrow}$ | 34 | | 17 | \| | 90 |
| 45 | | 68 | | 29 | | 34 | | 89 | $\overset{?}{\leftrightarrow}$ | 17 | \| | 90 |
| 45 | | 68 | | 29 | | 34 | | 17 | \| | 89 | | 90 |
| | | | | | | etc. | | | | | | |

**FIGURE 3.2** First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17.

# Analysis of Bubble sort

**ALGORITHM** *BubbleSort*$(A[0..n-1])$

//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
**for** $i \leftarrow 0$ **to** $n-2$ **do**
  **for** $j \leftarrow 0$ **to** $n-2-i$ **do**
    **if** $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

- The input size is n.
- The basic operation is comparison.
- The number of times it is executed $C(n) = \frac{n(n-1)}{2} \in \theta(n^2)$
- Also, $S_{Worst}(n) = \frac{n(n-1)}{2} \in \theta(n^2)$
- Bubble sort can be improved.

**ALGORITHM** *BubbleSort*$(A[0..n-1])$

//Sorts a given array by bubble sort

//Input: An array $A[0..n-1]$ of orderable elements

//Output: Array $A[0..n-1]$ sorted in nondecreasing order

**for** $i \leftarrow 0$ **to** $n-2$ **do**

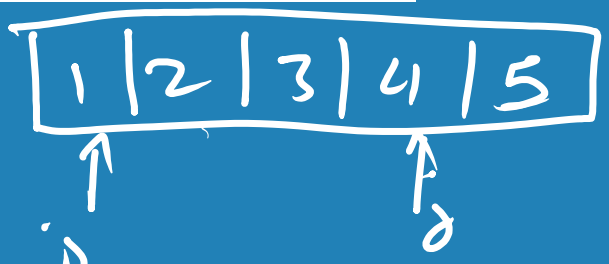
  **for** $j \leftarrow 0$ **to** $n-2-i$ **do**

    **if** $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

Worst $n^2$

best $n^2$

improvement



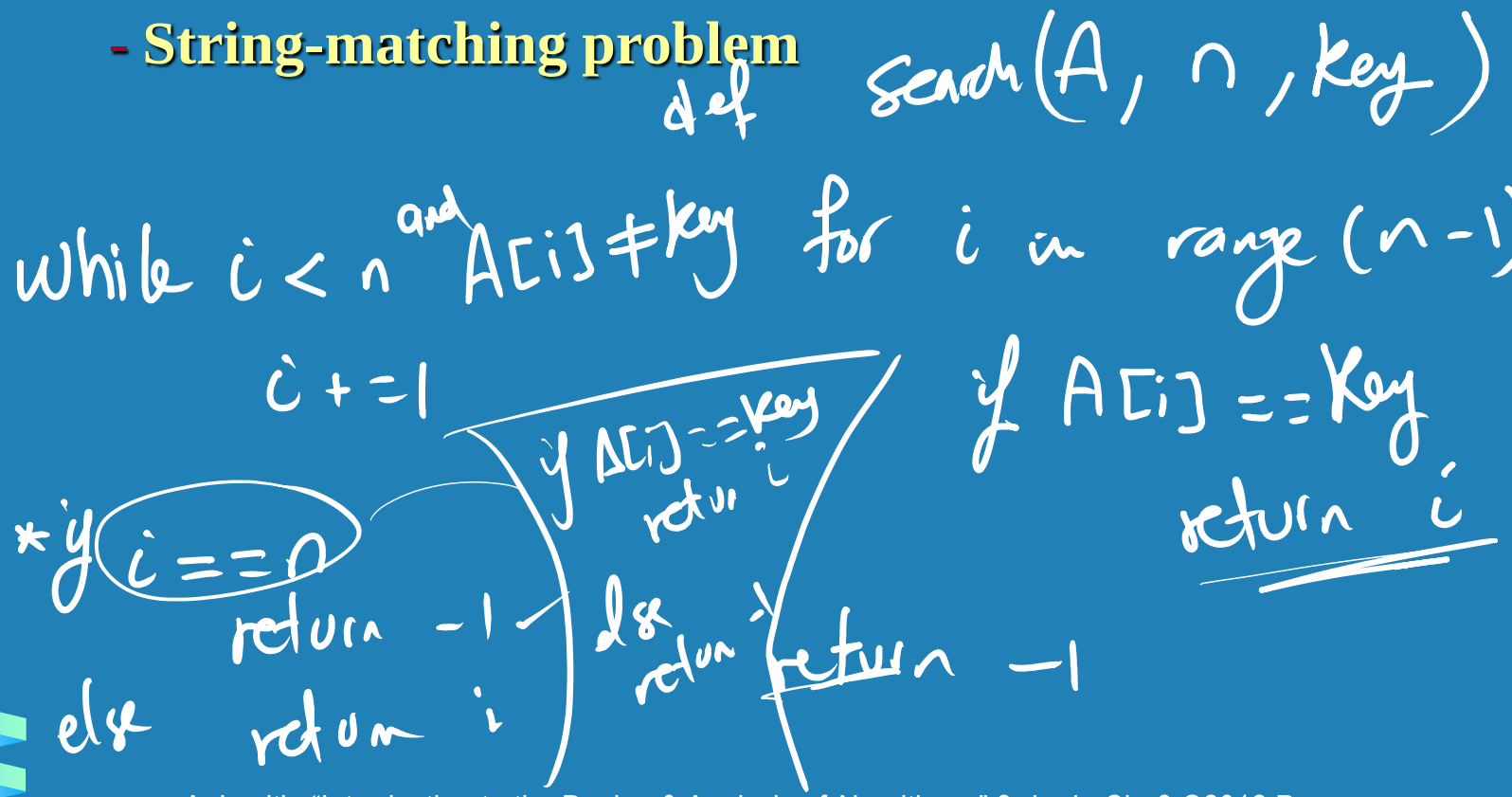

```
for i ← 0 to n-2 do
    flag = false
    for j ← 0 to n-2-i do
        if A[j+1] > A[j] { swap( ——— )
                          f = true ;
                        }
    if (flag == false) break
```

Worst $n^2$

best $n$

# Sequential Search and Brute-Force String Matching

- Here we apply brute-force strategy to the problem of searching:
  - Searching for an item a list
  - String-matching problem



# Sequential Search

- The algorithm compares successive elements of a list with a search key until either a match is encountered or the list is exhausted.
- One improvement is adding the search key to the end of the list.

**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] ← K
i ← 0
while A[i] ≠ K do
        i ← i + 1
if i < n return i
else return −1
```

Worst n
best 1

# Sequential Search

- **If a list is sorted, another is to stop the search if an element greater than or equal to the search key is encountered.**

**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
i ← 0
while A[i] < K do
        i ← i + 1
if A[i] = K return i
else return −1
```

# String Matching

- **Find a substring of a text that matches a pattern.**

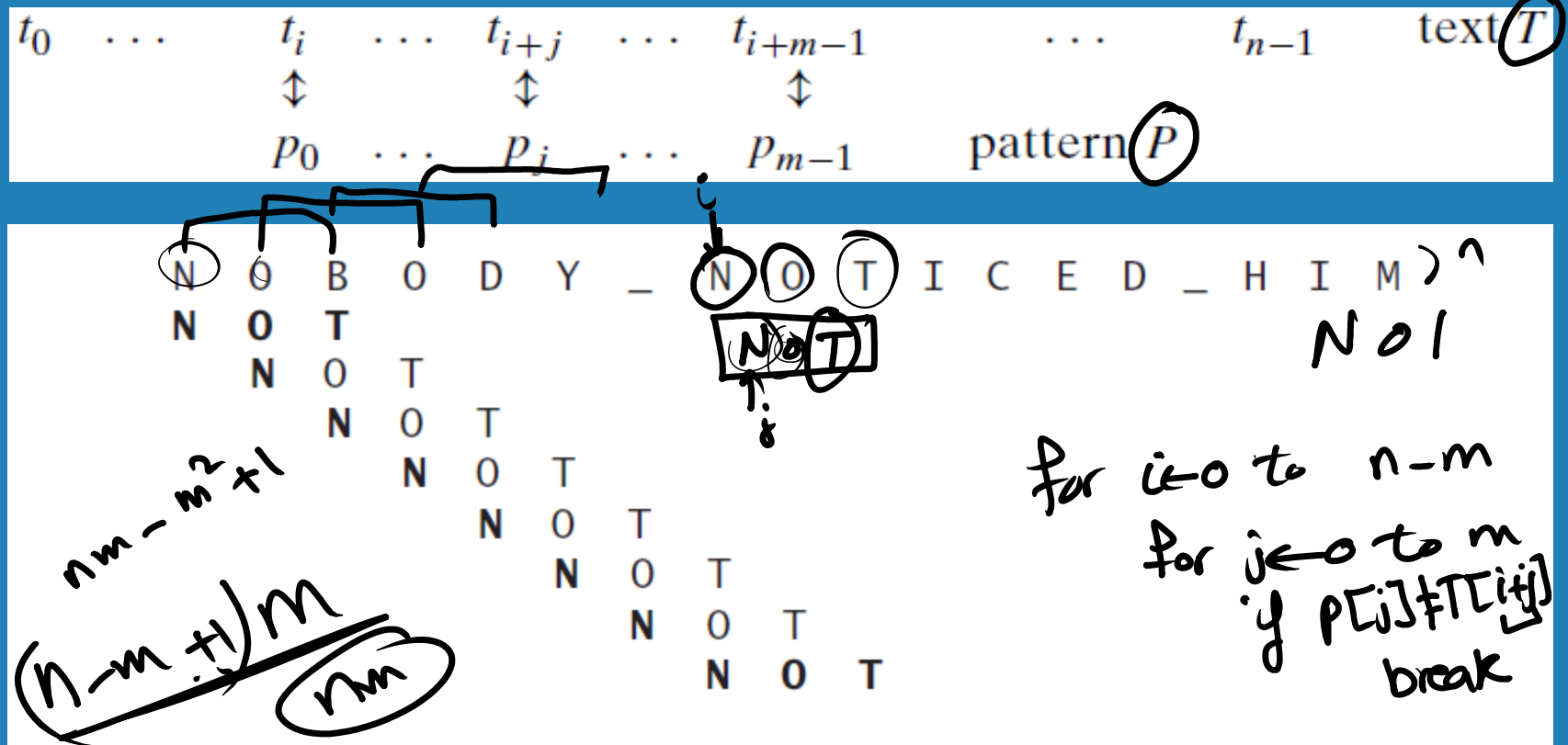



**FIGURE 3.3** Example of brute-force string matching. The pattern's characters that are compared with their text counterparts are in bold type.

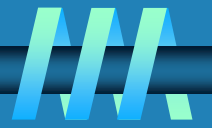
# String Matching

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

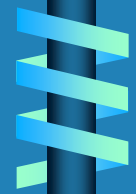
```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search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

$$C(n,m) = O(nm); if\ n = m \Longrightarrow O(n)$$

# Closest-Pair Problem



- Find the two closest points in a set of $n$ points.
- Points can represent airplanes, post offices, database records, and DNA sequences.
- Can be used in restructuring post office network, air-traffic control, and clustering.
- The metrics used are Euclidean distance (numerical data) and Hamming distance (nonnumerical data).
  - the Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different. For example, the hamming distance between the two binary 11011001, 10011101 = 2) are used.



# Closest-Pair Problem

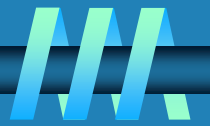
**ALGORITHM** *BruteForceClosestPair*$(P)$

//Finds distance between 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: The distance between the closest pair of points
$d \leftarrow \infty$
**for** $i \leftarrow 1$ **to** $n - 1$ **do**
    **for** $j \leftarrow i + 1$ **to** $n$ **do**
        $d \leftarrow \min(d, sqrt((x_i - x_j)^2 + (y_i - y_j)^2))$ //*sqrt* is square root
**return** $d$

- **The basic operation of the algorithm is computing the square root or squaring.**

- $C(n) = n(n-1) \epsilon\, \theta(n^2)$

# Convex-Hull Problem



- Convex-hull can be used in many applications.
  - Speeds up collision detection.
  - Detecting outliers
  - Computing a diameter of a set of points

# Convex-Hull Problem

- **Convex and non-convex sets**

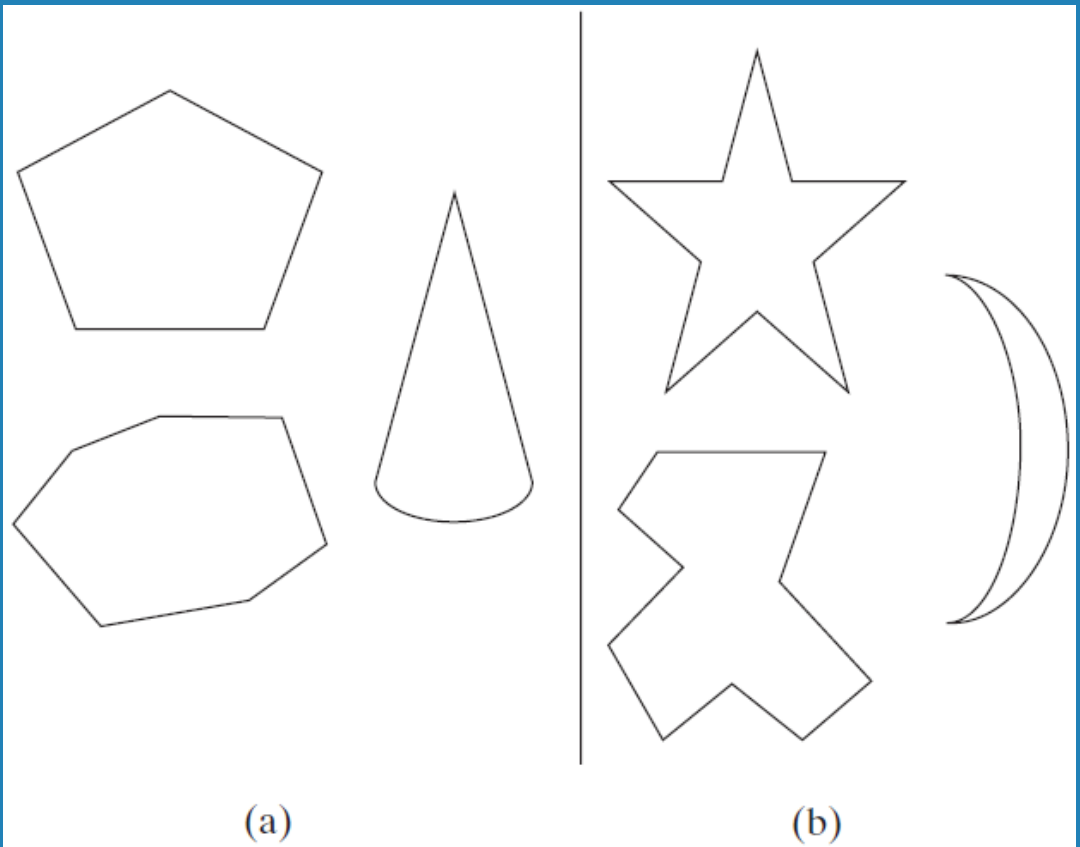



**FIGURE 3.4** (a) Convex sets. (b) Sets that are not convex.

# Convex-Hull Problem

- **Convex hull of a set of points *S***

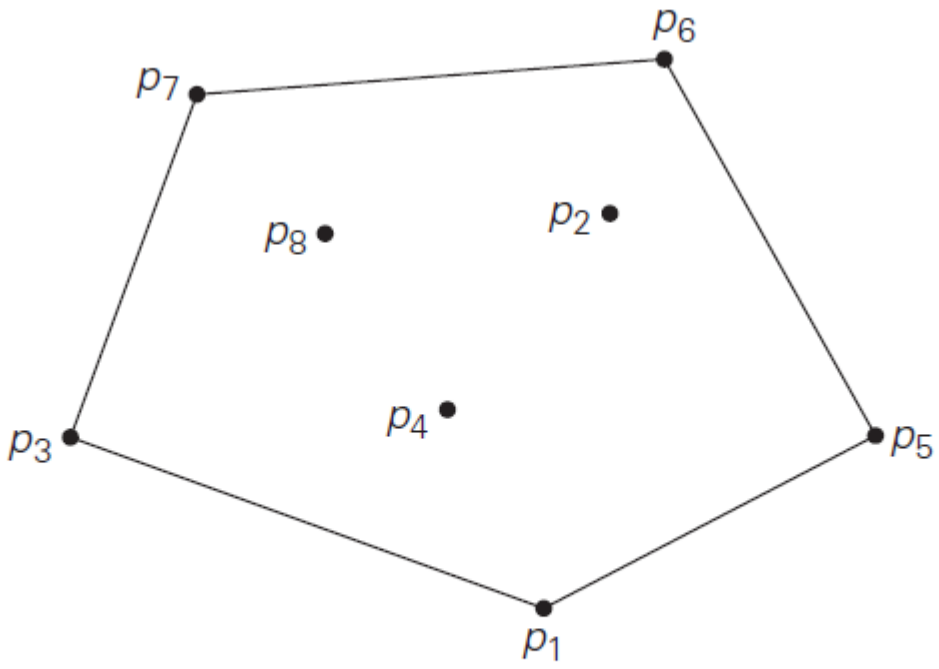
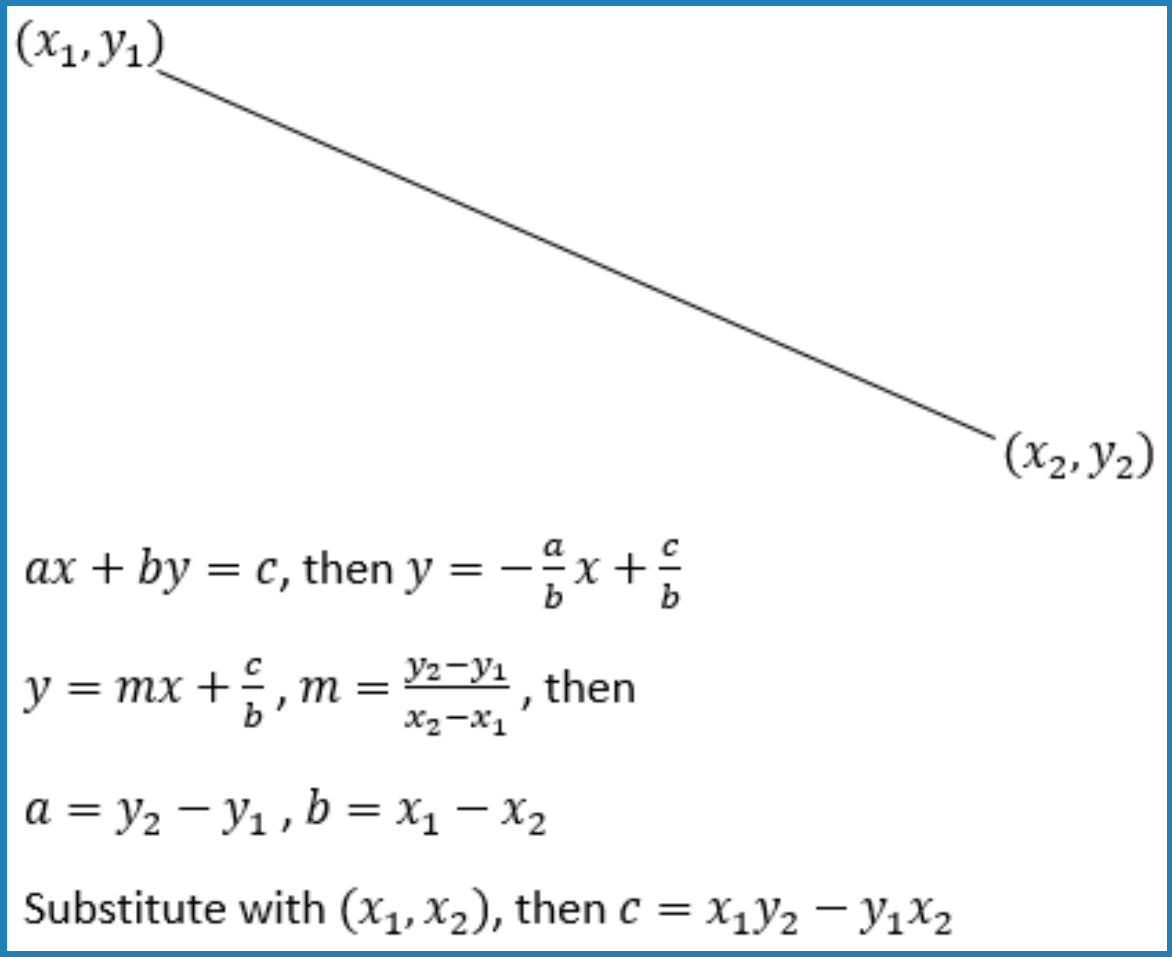



**FIGURE 3.6** The convex hull for this set of eight points is the convex polygon with vertices at $p_1$, $p_5$, $p_6$, $p_7$, and $p_3$.

# Convex-Hull Problem



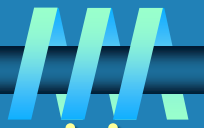

- **The time efficiency of this algorithm is in $O(n^3)$**

# Exhaustive Search

- Exhaustive search is a brute-force approach to combinatorial problems by generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element.

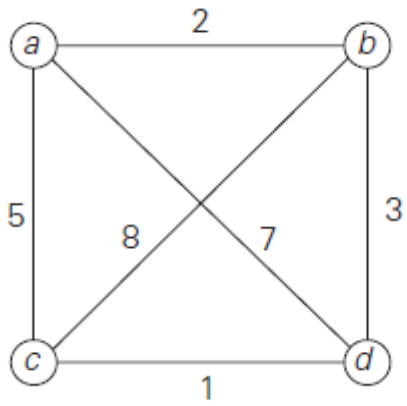
- We illustrate exhaustive search by applying it to three important problems:
  - The traveling salesman problem
  - The knapsack problem
  - The assignment problem.

# Traveling Salesman Problem (TSP)



- Find the shortest tour through a given set of $n$ cities that visits each city exactly once.
- Can be modeled by a weighted graph.
- Assume that all circuits start and end at one particular vertex, we can get all the tours by generating all the permutations of $n - 1$ intermediate cities, and find the shortest length among them.

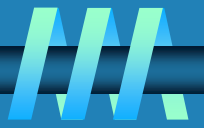
# Traveling Salesman Problem (TSP)




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

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

# Traveling Salesman Problem (TSP)



- The total number of permutations needed is $\frac{1}{2}(n-1)!$
- The exhaustive-search approach is impractical for all but small values of $n$.
- Considering all vertices to start from, increases the number of permutations, by a factor of $n$.

# Knapsack Problem

- **Find the most valuable subset of the items that fit into the knapsack.**

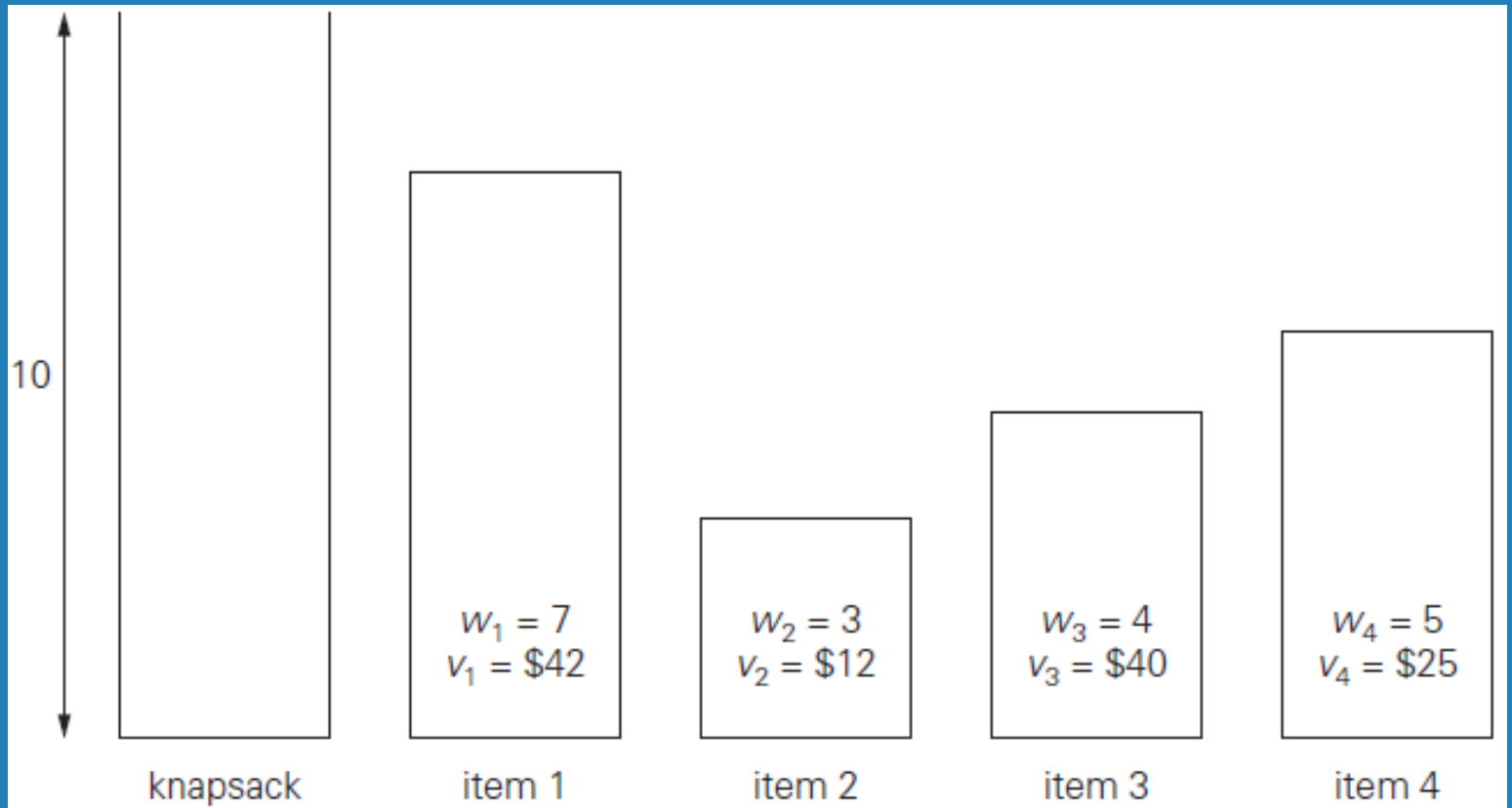



**FIGURE 3.8** (a) Instance of the knapsack problem.

# Knapsack Problem

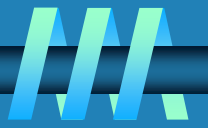
- It is $\Omega(2^n)$ algorithm
- NP-hard problem

| Subset | Total weight | Total value |
|---|---|---|
| ∅ | 0 | $ 0 |
| {1} | 7 | $42 |
| {2} | 3 | $12 |
| {3} | 4 | $40 |
| {4} | 5 | $25 |
| {1, 2} | 10 | $54 |
| {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** (b) Its solution by exhaustive search. The information about the optimal selection is in bold.

# Assignment Problem



- Assign $n$ people to execute $n$ jobs

| | Job 1 | Job 2 | Job 3 | Job 4 |
|---|---|---|---|---|
| Person 1 | 9 | 2 | 7 | 8 |
| Person 2 | 6 | 4 | 3 | 7 |
| Person 3 | 5 | 8 | 1 | 8 |
| Person 4 | 7 | 6 | 9 | 4 |

- Describe feasible solutions to the assignment problem as n-tuples $\langle j_1, \ldots, j_n \rangle$
- The solution requires generating all the permutations of integers $1, 2, \ldots, n$

# Assignment Problem

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

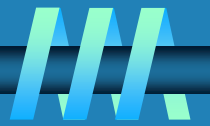
| | |
|---|---|
| <1, 2, 3, 4> | cost = 9 + 4 + 1 + 4 = 18 |
| <1, 2, 4, 3> | cost = 9 + 4 + 8 + 9 = 30 |
| <1, 3, 2, 4> | cost = 9 + 3 + 8 + 4 = 24 |
| <1, 3, 4, 2> | cost = 9 + 3 + 8 + 6 = 26 |
| <1, 4, 2, 3> | cost = 9 + 7 + 8 + 9 = 33 |
| <1, 4, 3, 2> | cost = 9 + 7 + 1 + 6 = 23 |

etc.

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

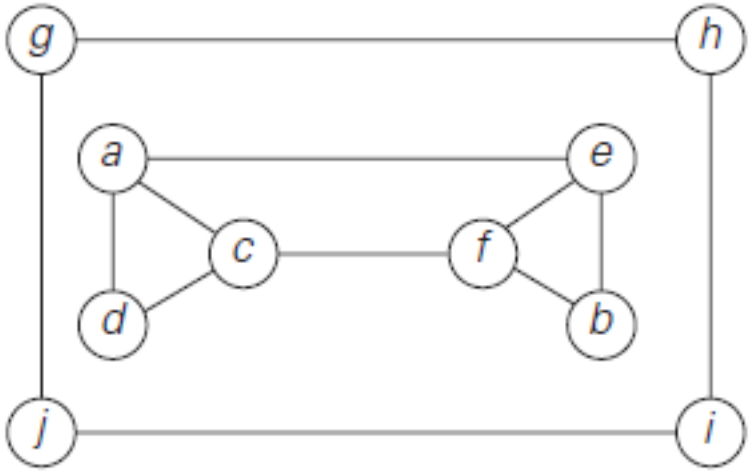
- **The number of permutations is *n!***
- **Exhaustive search is impractical for all but very small instances.**

# Depth-First and Breadth-First Search



- Exhaustive search can also be applied to
  - Depth-first search (DFS)
  - Breadth-first search (BFS)

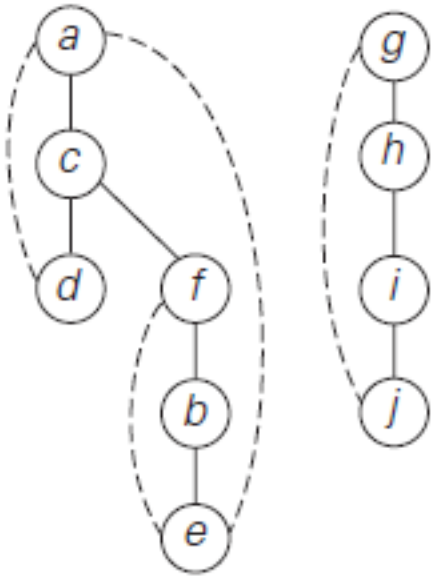
# Depth-First Search




(a)

$e_{6,2}$
$b_{5,3}$ $j_{10,7}$
$d_{3,1}$ $f_{4,4}$ $i_{9,8}$
$c_{2,5}$ $h_{8,9}$
$a_{1,6}$ $g_{7,10}$

(b)




(c)

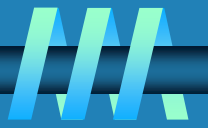
**FIGURE 3.10** Example of a DFS traversal. (a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex is visited, i.e., pushed onto the stack; the second one indicates the order in which it becomes a dead-end, i.e., popped off the stack). (c) DFS forest with the tree and back edges shown with solid and dashed lines, respectively.

# Depth-First Search

**ALGORITHM** *DFS*$(G)$

```
//Implements a depth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//        in the order they are first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        dfs(v)

dfs(v)
//visits recursively all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are encountered
//via global variable count
count ← count + 1;   mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        dfs(w)
```

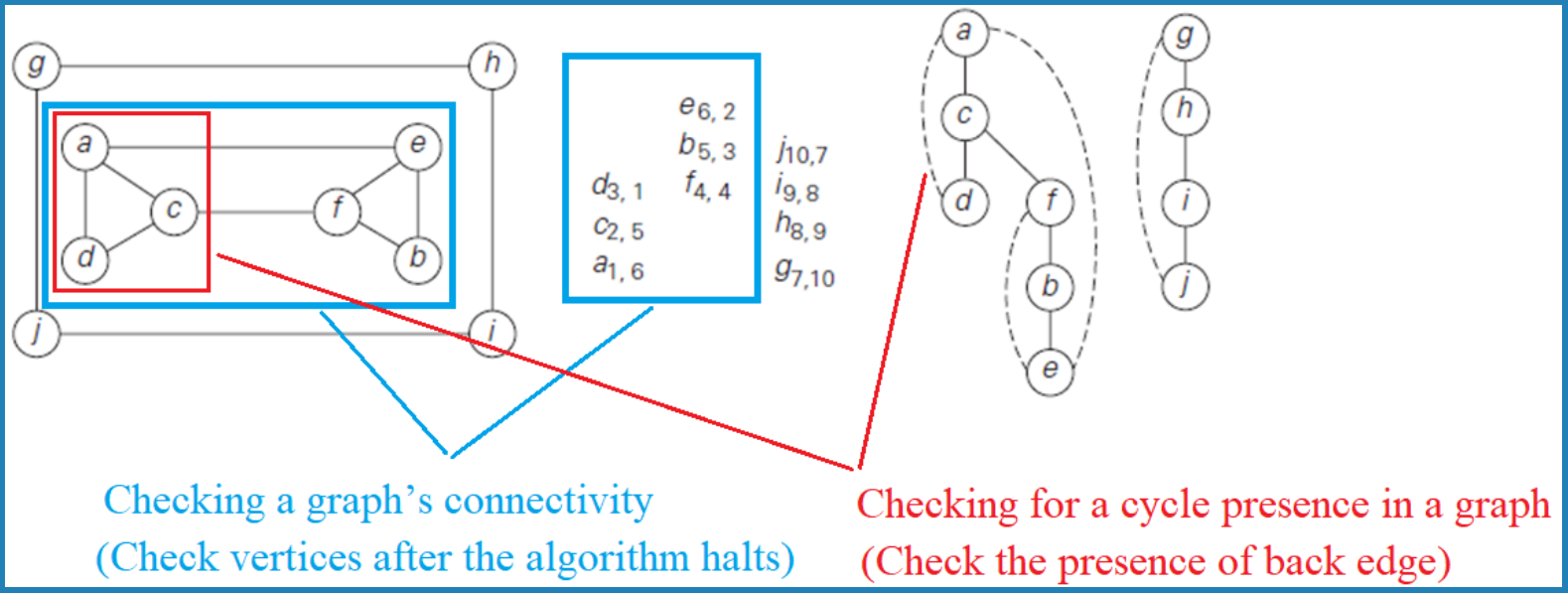
# Depth-First Search



- The DFS pseudocode can be applied using two representations.
  - Adjacency matrix, with traversal time in $\theta(|V|^2)$
  - Adjacency list, with traversal time in $\theta(|V| + |E|)$

# Depth-First Search

- **Applications of DFS .**
  - **Checking a graph's connectivity**
  - **Checking for a cycle presence in a graph**



# Breadth-First Search

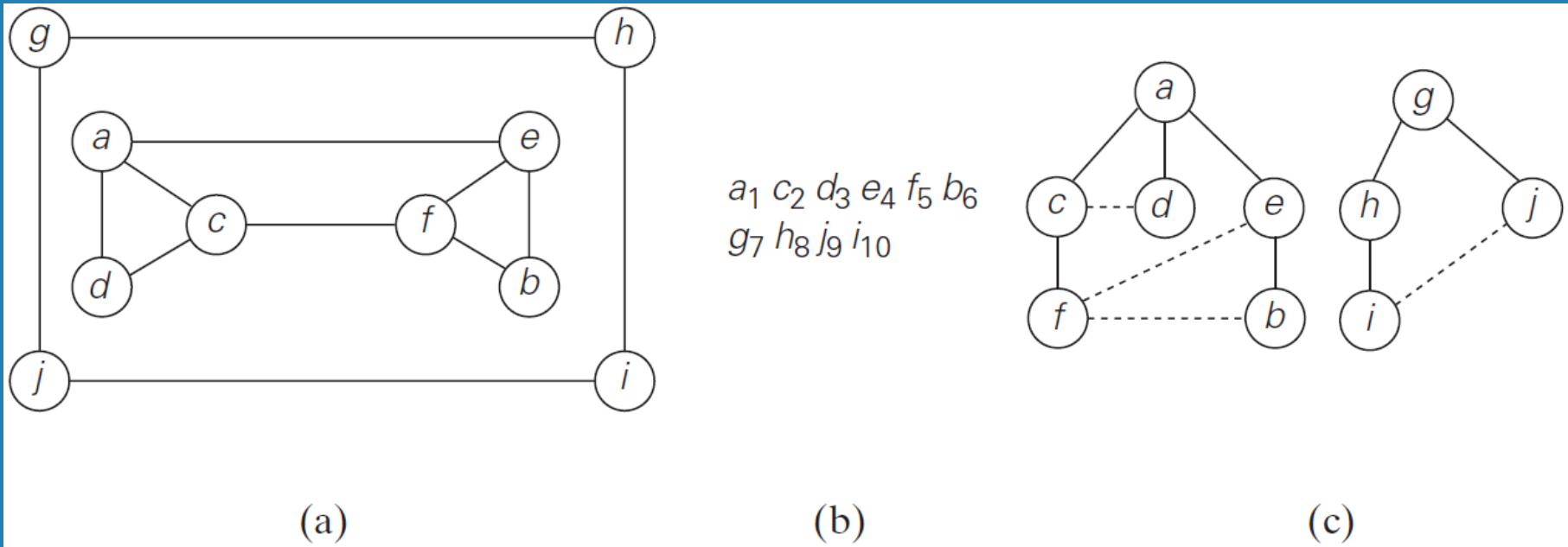
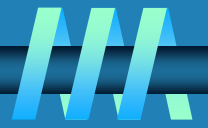



**FIGURE 3.11** Example of a BFS traversal. (a) Graph. (b) Traversal queue, with the numbers indicating the order in which the vertices are visited, i.e., added to (and removed from) the queue. (c) BFS forest with the tree and cross edges shown with solid and dotted lines, respectively.

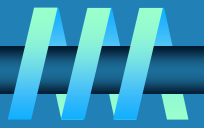
# Breadth-First Search



**ALGORITHM** *BFS*$(G)$

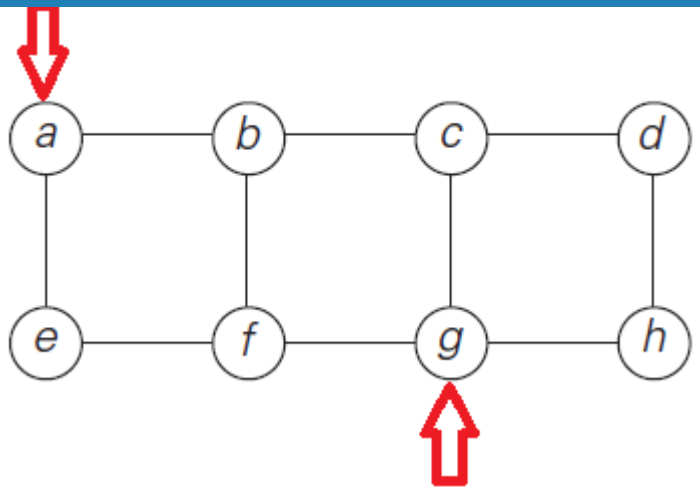
```
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//        in the order they are visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        bfs(v)

bfs(v)
count ← count + 1;  mark v with count and initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the front vertex do
        if w is marked with 0
            count ← count + 1;  mark w with count
            add w to the queue
    remove the front vertex from the queue
```

# Breadth-First Search

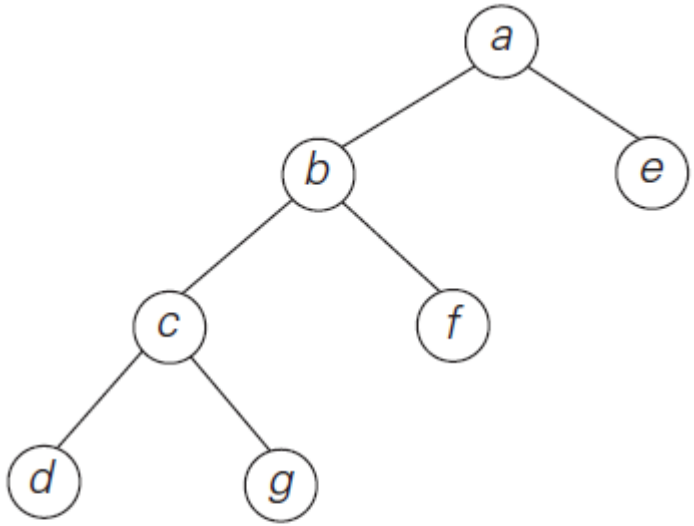


- BFS has the same efficiency as DFS.
- Unlike the DFS which yields two orderings of vertices, BFS yields a single ordering.
- Applications of DFS .
  - Checking a graph's connectivity
  - Checking for a cycle presence in a graph
  - Finding a path with the fewest number of edges between two given vertices

# Breadth-First Search




(a)




(b)

**FIGURE 3.12** Illustration of the BFS-based algorithm for finding a minimum-edge path. (a) Graph. (b) Part of its BFS tree that identifies the minimum-edge path from $a$ to $g$.

# Breadth-First Search

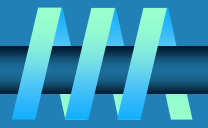


**TABLE 3.1** Main facts about depth-first search (DFS) and breadth-first search (BFS)

| | DFS | BFS |
|---|---|---|
| Data structure | a stack | a queue |
| Number of vertex orderings | two orderings | one ordering |
| Edge types (undirected graphs) | tree and back edges | tree and cross edges |
| Efficiency for adjacency matrix | $\Theta(\|V^2\|)$ | $\Theta(\|V^2\|)$ |
| Efficiency for adjacency lists | $\Theta(\|V\| + \|E\|)$ | $\Theta(\|V\| + \|E\|)$ |