nected by a path to the starting vertex, checking a graph's connectivity can be done as follows. Start a DFS traversal at an arbitrary vertex and check, after the algorithm halts, whether all the vertices of the graph will have been visited. If they have, the graph is connected; otherwise, it is not connected. More generally, we can use DFS for identifying connected components of a graph (how?).

As for checking for a cycle presence in a graph, we can take advantage of the graph's representation in the form of a DFS forest. If the latter does not have back edges, the graph is clearly acyclic. If there is a back edge from some vertex $u$ to its ancestor $v$ (e.g., the back edge from $d$ to $a$ in Figure 3.10c), the graph has a cycle that comprises the path from $v$ to $u$ via a sequence of tree edges in the DFS forest followed by the back edge from $u$ to $v$.

You will find a few other applications of DFS later in the book, although more sophisticated applications, such as finding articulation points of a graph, are not included. (A vertex of a connected graph is said to be its ***articulation point*** if its removal with all edges incident to it breaks the graph into disjoint pieces.)

## Breadth-First Search

If depth-first search is a traversal for the brave (the algorithm goes as far from "home" as it can), breadth-first search is a traversal for the cautious. It proceeds in a concentric manner by visiting first all the vertices that are adjacent to a starting vertex, then all unvisited vertices two edges apart from it, and so on, until all the vertices in the same connected component as the starting vertex are visited. If there still remain unvisited vertices, the algorithm has to be restarted at an arbitrary vertex of another connected component of the graph.

It is convenient to use a queue (note the difference from depth-first search!) to trace the operation of breadth-first search. The queue is initialized with the traversal's starting vertex, which is marked as visited. On each iteration, the algorithm identifies all unvisited vertices that are adjacent to the front vertex, marks them as visited, and adds them to the queue; after that, the front vertex is removed from the queue.

Similarly to a DFS traversal, it is useful to accompany a BFS traversal by constructing the so-called ***breadth-first search forest***. The traversal's starting vertex serves as the root of the first tree in such a forest. Whenever a new unvisited vertex is reached for the first time, the vertex is attached as a child to the vertex it is being reached from with an edge called a ***tree edge***. If an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree) is encountered, the edge is noted as a ***cross edge***. Figure 3.11 provides an example of a breadth-first search traversal, with the traversal queue and corresponding breadth-first search forest shown.

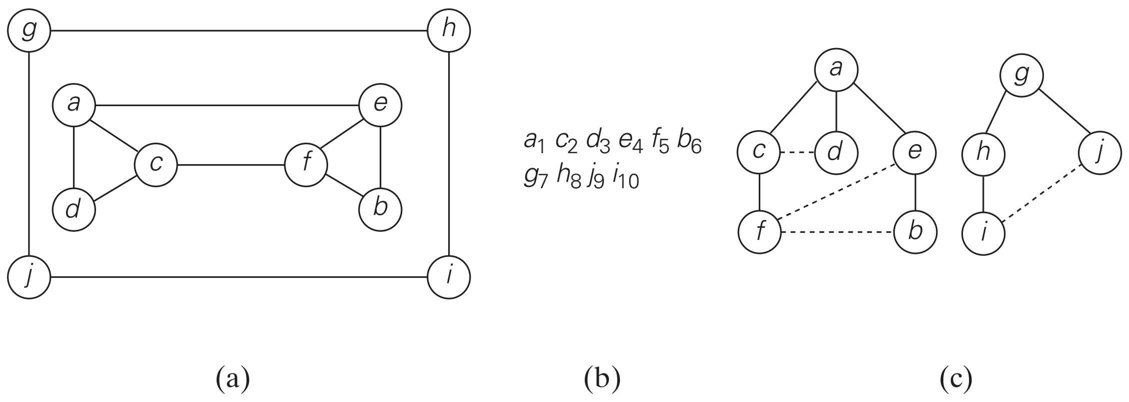

g
h
a
e
c
f
d
b
j
i
$a_1$ $c_2$ $d_3$ $e_4$ $f_5$ $b_6$
$g_7$ $h_8$ $j_9$ $i_{10}$
(a)
(b)
(c)

**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.

Here is pseudocode of the breadth-first search.

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

```
//Implements a breadth-first search traversal of a given graph
//Input: Graph $G = \langle V, E \rangle$
//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"
$count \leftarrow 0$
for each vertex $v$ in $V$ do
    if $v$ is marked with 0
        $bfs(v)$

$bfs(v)$
//visits all the unvisited vertices connected to vertex $v$
//by a path and numbers them in the order they are visited
//via global variable $count$
$count \leftarrow count + 1$;   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
            $count \leftarrow count + 1$;   mark $w$ with $count$
            add $w$ to the queue
    remove the front vertex from the queue
```

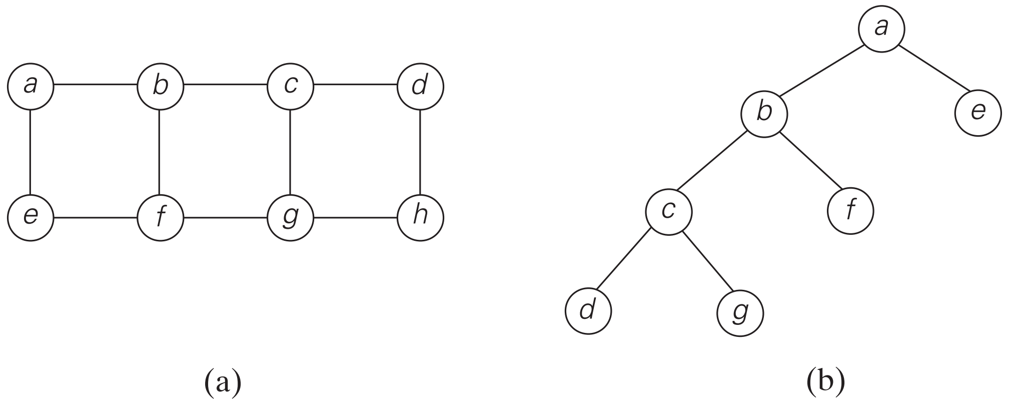


**FIGURE 3.12** Illustration of the BFS-based algorithm for finding a minimum-edge path. (a) Graph. (b) Part of its BFS tree that identifies the minimum-edge path from $a$ to $g$.

Breadth-first search has the same efficiency as depth-first search: it is in $\Theta(|V|^2)$ for the adjacency matrix representation and in $\Theta(|V| + |E|)$ for the adjacency list representation. Unlike depth-first search, it yields a single ordering of vertices because the queue is a FIFO (first-in first-out) structure and hence the order in which vertices are added to the queue is the same order in which they are removed from it. As to the structure of a BFS forest of an undirected graph, it can also have two kinds of edges: tree edges and cross edges. Tree edges are the ones used to reach previously unvisited vertices. Cross edges connect vertices to those visited before, but, unlike back edges in a DFS tree, they connect vertices either on the same or adjacent levels of a BFS tree.

BFS can be used to check connectivity and acyclicity of a graph, essentially in the same manner as DFS can. It is not applicable, however, for several less straightforward applications such as finding articulation points. On the other hand, it can be helpful in some situations where DFS cannot. For example, BFS can be used for finding a path with the fewest number of edges between two given vertices. To do this, we start a BFS traversal at one of the two vertices and stop it as soon as the other vertex is reached. The simple path from the root of the BFS tree to the second vertex is the path sought. For example, path $a - b - c - g$ in the graph in Figure 3.12 has the fewest number of edges among all the paths between vertices $a$ and $g$. Although the correctness of this application appears to stem immediately from the way BFS operates, a mathematical proof of its validity is not quite elementary (see, e.g., [Cor09, Section 22.2]).

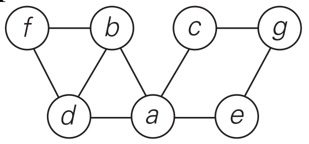
Table 3.1 summarizes the main facts about depth-first search and breadth-first search.

**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 |
| Applications | connectivity, acyclicity, articulation points | connectivity, acyclicity, minimum-edge paths |
| Efficiency for adjacency matrix | $\Theta(\lvert V^2\rvert)$ | $\Theta(\lvert V^2\rvert)$ |
| Efficiency for adjacency lists | $\Theta(\lvert V\rvert + \lvert E\rvert)$ | $\Theta(\lvert V\rvert + \lvert E\rvert)$ |

## Exercises 3.5

1. Consider the following graph.



   **a.** Write down the adjacency matrix and adjacency lists specifying this graph. (Assume that the matrix rows and columns and vertices in the adjacency lists follow in the alphabetical order of the vertex labels.)

   **b.** Starting at vertex $a$ and resolving ties by the vertex alphabetical order, traverse the graph by depth-first search and construct the corresponding depth-first search tree. Give the order in which the vertices were reached for the first time (pushed onto the traversal stack) and the order in which the vertices became dead ends (popped off the stack).

2. If we define sparse graphs as graphs for which $|E| \in O(|V|)$, which implementation of DFS will have a better time efficiency for such graphs, the one that uses the adjacency matrix or the one that uses the adjacency lists?

3. Let $G$ be a graph with $n$ vertices and $m$ edges.

   **a.** True or false: All its DFS forests (for traversals starting at different vertices) will have the same number of trees?

   **b.** True or false: All its DFS forests will have the same number of tree edges and the same number of back edges?

4. Traverse the graph of Problem 1 by breadth-first search and construct the corresponding breadth-first search tree. Start the traversal at vertex $a$ and resolve ties by the vertex alphabetical order.

**5.** Prove that a cross edge in a BFS tree of an undirected graph can connect vertices only on either the same level or on two adjacent levels of a BFS tree.

**6. a.** Explain how one can check a graph's acyclicity by using breadth-first search.

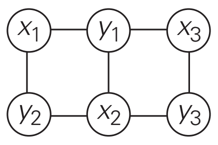
**b.** Does either of the two traversals—DFS or BFS—always find a cycle faster than the other? If you answer yes, indicate which of them is better and explain why it is the case; if you answer no, give two examples supporting your answer.

**7.** Explain how one can identify connected components of a graph by using

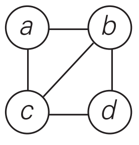
**a.** a depth-first search.

**b.** a breadth-first search.

**8.** A graph is said to be ***bipartite*** if all its vertices can be partitioned into two disjoint subsets $X$ and $Y$ so that every edge connects a vertex in $X$ with a vertex in $Y$. (One can also say that a graph is bipartite if its vertices can be colored in two colors so that every edge has its vertices colored in different colors; such graphs are also called ***2-colorable***.) For example, graph (i) is bipartite while graph (ii) is not.



(i)



(ii)

**a.** Design a DFS-based algorithm for checking whether a graph is bipartite.

**b.** Design a BFS-based algorithm for checking whether a graph is bipartite.

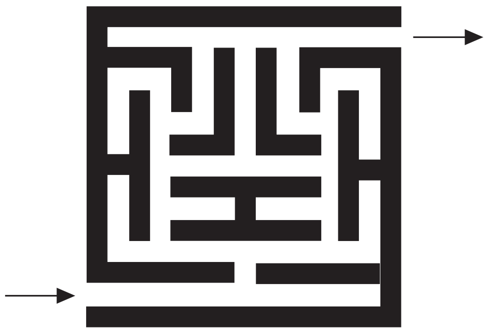
**9.** Write a program that, for a given graph, outputs:

**a.** vertices of each connected component

**b.** its cycle or a message that the graph is acyclic

**10.** One can model a maze by having a vertex for a starting point, a finishing point, dead ends, and all the points in the maze where more than one path can be taken, and then connecting the vertices according to the paths in the maze.

**a.** Construct such a graph for the following maze.



**b.** Which traversal—DFS or BFS—would you use if you found yourself in a maze and why?

**11.** *Three Jugs* Siméon Denis Poisson (1781–1840), a famous French mathematician and physicist, is said to have become interested in mathematics after encountering some version of the following old puzzle. Given an 8-pint jug full of water and two empty jugs of 5- and 3-pint capacity, get exactly 4 pints of water in one of the jugs by completely filling up and/or emptying jugs into others. Solve this puzzle by using breadth-first search.

## SUMMARY

- *Brute force* is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.
- The principal strengths of the brute-force approach are wide applicability and simplicity; its principal weakness is the subpar efficiency of most brute-force algorithms.
- A first application of the brute-force approach often results in an algorithm that can be improved with a modest amount of effort.
- The following noted algorithms can be considered as examples of the brute-force approach:
  - definition-based algorithm for matrix multiplication
  - *selection sort*
  - *sequential search*
  - straightforward string-matching algorithm
- *Exhaustive search* is a brute-force approach to combinatorial problems. It suggests generating each and every combinatorial object of the problem, selecting those of them that satisfy all the constraints, and then finding a desired object.
- The *traveling salesman problem,* the *knapsack problem,* and the *assignment problem* are typical examples of problems that can be solved, at least theoretically, by exhaustive-search algorithms.
- Exhaustive search is impractical for all but very small instances of problems it can be applied to.
- *Depth-first search* (*DFS*) and *breadth-first search* (*BFS*) are two principal graph-traversal algorithms. By representing a graph in a form of a depth-first or breadth-first search forest, they help in the investigation of many important properties of the graph. Both algorithms have the same time efficiency: $\Theta(|V|^2)$ for the adjacency matrix representation and $\Theta(|V| + |E|)$ for the adjacency list representation.

# 4 Decrease-and-Conquer

*Plutarch says that Sertorius, in order to teach his soldiers that perseverance and wit are better than brute force, had two horses brought before them, and set two men to pull out their tails. One of the men was a burly Hercules, who tugged and tugged, but all to no purpose; the other was a sharp, weasel-faced tailor, who plucked one hair at a time, amidst roars of laughter, and soon left the tail quite bare.*

—E. Cobham Brewer, *Dictionary of Phrase and Fable*, 1898

The ***decrease-and-conquer*** technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance. Once such a relationship is established, it can be exploited either top down or bottom up. The former leads naturally to a recursive implementation, although, as one can see from several examples in this chapter, an ultimate implementation may well be nonrecursive. The bottom-up variation is usually implemented iteratively, starting with a solution to the smallest instance of the problem; it is called sometimes the ***incremental approach***.

There are three major variations of decrease-and-conquer:

- decrease by a constant
- decrease by a constant factor
- variable size decrease

In the ***decrease-by-a-constant*** variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one (Figure 4.1), although other constant size reductions do happen occasionally.

Consider, as an example, the exponentiation problem of computing $a^n$ where $a \neq 0$ and $n$ is a nonnegative integer. The relationship between a solution to an instance of size $n$ and an instance of size $n-1$ is obtained by the obvious formula $a^n = a^{n-1} \cdot a$. So the function $f(n) = a^n$ can be computed either "top down" by using its recursive definition

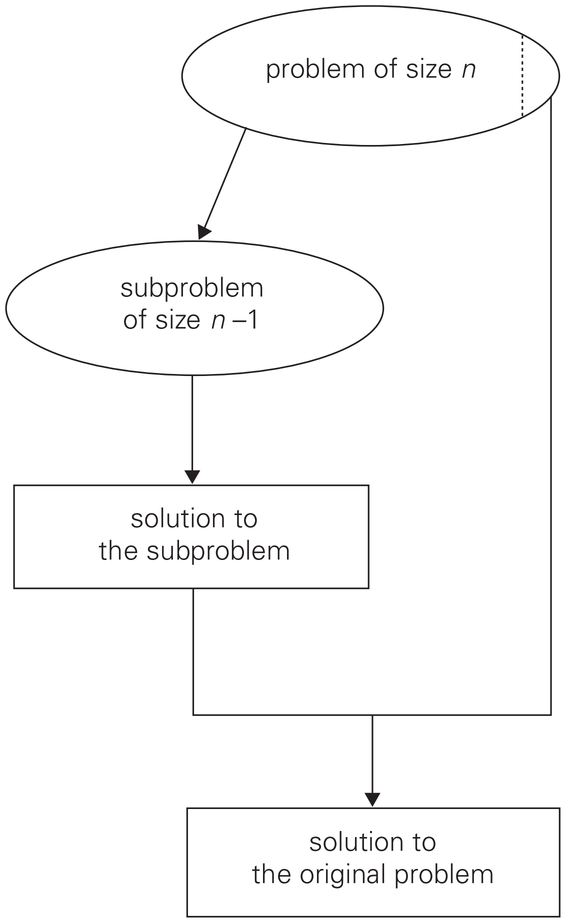


**FIGURE 4.1** Decrease-(by one)-and-conquer technique.

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0, \end{cases} \tag{4.1}$$

or "bottom up" by multiplying 1 by $a$ $n$ times. (Yes, it is the same as the brute-force algorithm, but we have come to it by a different thought process.) More interesting examples of decrease-by-one algorithms appear in Sections 4.1–4.3.

The ***decrease-by-a-constant-factor*** technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two. (Can you give an example of such an algorithm?) The decrease-by-half idea is illustrated in Figure 4.2.

For an example, let us revisit the exponentiation problem. If the instance of size $n$ is to compute $a^n$, the instance of half its size is to compute $a^{n/2}$, with the obvious relationship between the two: $a^n = (a^{n/2})^2$. But since we consider here instances with integer exponents only, the former does not work for odd $n$. If $n$ is odd, we have to compute $a^{n-1}$ by using the rule for even-valued exponents and then multiply the result by $a$. To summarize, we have the following formula:

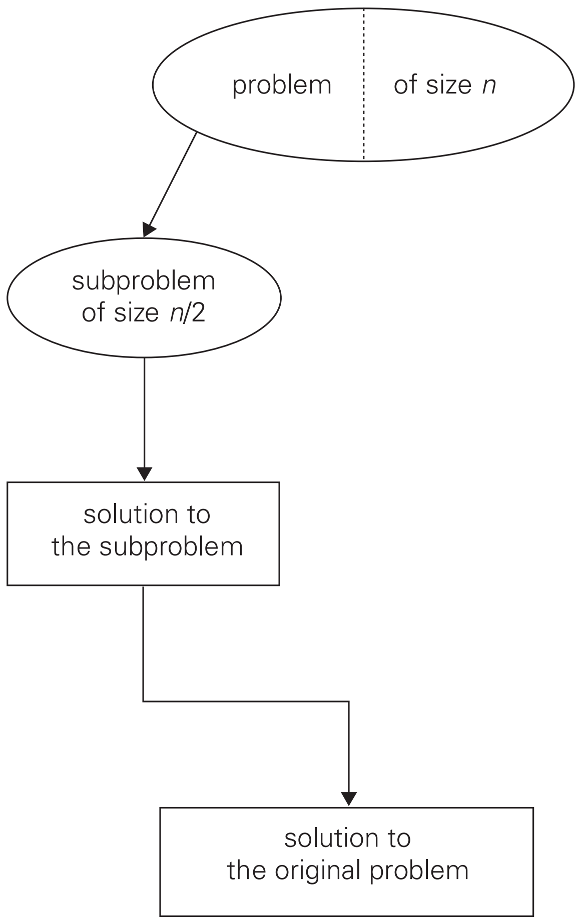


**FIGURE 4.2** Decrease-(by half)-and-conquer technique.

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases} \tag{4.2}$$

If we compute $a^n$ recursively according to formula (4.2) and measure the algorithm's efficiency by the number of multiplications, we should expect the algorithm to be in $\Theta(\log n)$ because, on each iteration, the size is reduced by about a half at the expense of one or two multiplications.

A few other examples of decrease-by-a-constant-factor algorithms are given in Section 4.4 and its exercises. Such algorithms are so efficient, however, that there are few examples of this kind.

Finally, in the ***variable-size-decrease*** variety of decrease-and-conquer, the size-reduction pattern varies from one iteration of an algorithm to another. Euclid's algorithm for computing the greatest common divisor provides a good example of such a situation. Recall that this algorithm is based on the formula

$$\gcd(m, n) = \gcd(n, m \bmod n).$$

Though the value of the second argument is always smaller on the right-hand side than on the left-hand side, it decreases neither by a constant nor by a constant factor. A few other examples of such algorithms appear in Section 4.5.

## 4.1 Insertion Sort

In this section, we consider an application of the decrease-by-one technique to sorting an array $A[0..n-1]$. Following the technique's idea, we assume that the smaller problem of sorting the array $A[0..n-2]$ has already been solved to give us a sorted array of size $n-1$: $A[0] \leq \cdots \leq A[n-2]$. How can we take advantage of this solution to the smaller problem to get a solution to the original problem by taking into account the element $A[n-1]$? Obviously, all we need is to find an appropriate position for $A[n-1]$ among the sorted elements and insert it there. This is usually done by scanning the sorted subarray from right to left until the first element smaller than or equal to $A[n-1]$ is encountered to insert $A[n-1]$ right after that element. The resulting algorithm is called ***straight insertion sort*** or simply ***insertion sort***.

Though insertion sort is clearly based on a recursive idea, it is more efficient to implement this algorithm bottom up, i.e., iteratively. As shown in Figure 4.3, starting with $A[1]$ and ending with $A[n-1]$, $A[i]$ is inserted in its appropriate place among the first $i$ elements of the array that have been already sorted (but, unlike selection sort, are generally not in their final positions).

Here is pseudocode of this algorithm.

```
ALGORITHM InsertionSort(A[0..n − 1])
    //Sorts a given array by insertion sort
    //Input: An array A[0..n − 1] of n orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    for i ← 1 to n − 1 do
        v ← A[i]
        j ← i − 1
        while j ≥ 0 and A[j] > v do
            A[j + 1] ← A[j]
            j ← j − 1
        A[j + 1] ← v
```

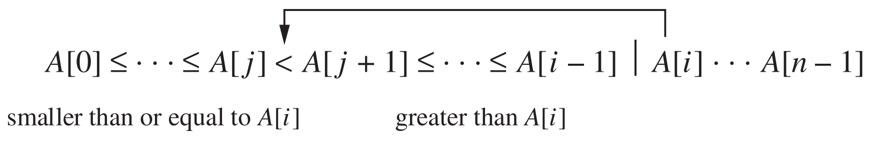

$A[0] \leq \cdots \leq A[j] < A[j+1] \leq \cdots \leq A[i-1]$ | $A[i] \cdots A[n-1]$
smaller than or equal to $A[i]$
greater than $A[i]$

**FIGURE 4.3** Iteration of insertion sort: $A[i]$ is inserted in its proper position among the preceding elements previously sorted.

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

**FIGURE 4.4** Example of sorting with insertion sort. A vertical bar separates the sorted part of the array from the remaining elements; the element being inserted is in bold.

The operation of the algorithm is illustrated in Figure 4.4.

The basic operation of the algorithm is the key comparison $A[j] > v$. (Why not $j \geq 0$? Because it is almost certainly faster than the former in an actual computer implementation. Moreover, it is not germane to the algorithm: a better implementation with a sentinel—see Problem 8 in this section's exercises—eliminates it altogether.)

The number of key comparisons in this algorithm obviously depends on the nature of the input. In the worst case, $A[j] > v$ is executed the largest number of times, i.e., for every $j = i - 1, \ldots, 0$. Since $v = A[i]$, it happens if and only if $A[j] > A[i]$ for $j = i - 1, \ldots, 0$. (Note that we are using the fact that on the $i$th iteration of insertion sort all the elements preceding $A[i]$ are the first $i$ elements in the input, albeit in the sorted order.) Thus, for the worst-case input, we get $A[0] > A[1]$ (for $i = 1$), $A[1] > A[2]$ (for $i = 2$), $\ldots$, $A[n-2] > A[n-1]$ (for $i = n - 1$). In other words, the worst-case input is an array of strictly decreasing values. The number of key comparisons for such an input is

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Thus, in the worst case, insertion sort makes exactly the same number of comparisons as selection sort (see Section 3.1).

In the best case, the comparison $A[j] > v$ is executed only once on every iteration of the outer loop. It happens if and only if $A[i-1] \leq A[i]$ for every $i = 1, \ldots, n - 1$, i.e., if the input array is already sorted in nondecreasing order. (Though it "makes sense" that the best case of an algorithm happens when the problem is already solved, it is not always the case, as you are going to see in our discussion of quicksort in Chapter 5.) Thus, for sorted arrays, the number of key comparisons is

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

This very good performance in the best case of sorted arrays is not very useful by itself, because we cannot expect such convenient inputs. However, almost-sorted files do arise in a variety of applications, and insertion sort preserves its excellent performance on such inputs.

A rigorous analysis of the algorithm's average-case efficiency is based on investigating the number of element pairs that are out of order (see Problem 11 in this section's exercises). It shows that on randomly ordered arrays, insertion sort makes on average half as many comparisons as on decreasing arrays, i.e.,

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2).$$

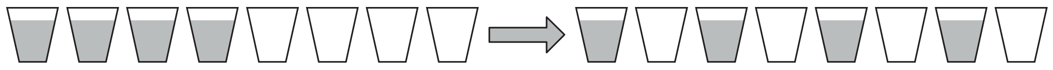
This twice-as-fast average-case performance coupled with an excellent efficiency on almost-sorted arrays makes insertion sort stand out among its principal competitors among elementary sorting algorithms, selection sort and bubble sort. In addition, its extension named ***shellsort***, after its inventor D. L. Shell [She59], gives us an even better algorithm for sorting moderately large files (see Problem 12 in this section's exercises).

## Exercises 4.1

1. *Ferrying soldiers* A detachment of $n$ soldiers must cross a wide and deep river with no bridge in sight. They notice two 12-year-old boys playing in a rowboat by the shore. The boat is so tiny, however, that it can only hold two boys or one soldier. How can the soldiers get across the river and leave the boys in joint possession of the boat? How many times need the boat pass from shore to shore?

2. *Alternating glasses*

   **a.** There are $2n$ glasses standing next to each other in a row, the first $n$ of them filled with a soda drink and the remaining $n$ glasses empty. Make the glasses alternate in a filled-empty-filled-empty pattern in the minimum number of glass moves. [Gar78]



   **b.** Solve the same problem if $2n$ glasses—$n$ with a drink and $n$ empty—are initially in a random order.

3. *Marking cells* Design an algorithm for the following task. For any even $n$, mark $n$ cells on an infinite sheet of graph paper so that each marked cell has an odd number of marked neighbors. Two cells are considered neighbors if they are next to each other either horizontally or vertically but not diagonally. The marked cells must form a contiguous region, i.e., a region in which there is a path between any pair of marked cells that goes through a sequence of marked neighbors. [Kor05]

4. Design a decrease-by-one algorithm for generating the power set of a set of $n$ elements. (The power set of a set $S$ is the set of all the subsets of $S$, including the empty set and $S$ itself.)

5. Consider the following algorithm to check connectivity of a graph defined by its adjacency matrix.

```
ALGORITHM  Connected(A[0..n − 1, 0..n − 1])
    //Input: Adjacency matrix A[0..n − 1, 0..n − 1]) of an undirected graph G
    //Output: 1 (true) if G is connected and 0 (false) if it is not
    if n = 1 return 1   //one-vertex graph is connected by definition
    else
        if not Connected(A[0..n − 2, 0..n − 2]) return 0
        else for j ← 0 to n − 2 do
                if A[n − 1, j] return 1
            return 0
```

Does this algorithm work correctly for every undirected graph with $n > 0$ vertices? If you answer yes, indicate the algorithm's efficiency class in the worst case; if you answer no, explain why.

6. *Team ordering* You have the results of a completed round-robin tournament in which $n$ teams played each other once. Each game ended either with a victory for one of the teams or with a tie. Design an algorithm that lists the teams in a sequence so that every team did not lose the game with the team listed immediately after it. What is the time efficiency class of your algorithm?

7. Apply insertion sort to sort the list $E$, $X$, $A$, $M$, $P$, $L$, $E$ in alphabetical order.

8. **a.** What sentinel should be put before the first element of an array being sorted in order to avoid checking the in-bound condition $j \geq 0$ on each iteration of the inner loop of insertion sort?

   **b.** Is the sentinel version in the same efficiency class as the original version?

9. Is it possible to implement insertion sort for sorting linked lists? Will it have the same $O(n^2)$ time efficiency as the array version?

10. Compare the text's implementation of insertion sort with the following version.

```
ALGORITHM  InsertSort2(A[0..n − 1])
    for i ← 1 to n − 1 do
        j ← i − 1
        while j ≥ 0 and A[j] > A[j + 1] do
            swap(A[j], A[j + 1])
            j ← j − 1
```

What is the time efficiency of this algorithm? How is it compared to that of the version given in Section 4.1?

**11.** Let $A[0..n-1]$ be an array of $n$ sortable elements. (For simplicity, you may assume that all the elements are distinct.) A pair $(A[i], A[j])$ is called an ***inversion*** if $i < j$ and $A[i] > A[j]$.

**a.** What arrays of size $n$ have the largest number of inversions and what is this number? Answer the same questions for the smallest number of inversions.

**b.** Show that the average-case number of key comparisons in insertion sort is given by the formula

$$C_{avg}(n) \approx \frac{n^2}{4}.$$

**12.** Shellsort (more accurately Shell's sort) is an important sorting algorithm that works by applying insertion sort to each of several interleaving sublists of a given list. On each pass through the list, the sublists in question are formed by stepping through the list with an increment $h_i$ taken from some predefined decreasing sequence of step sizes, $h_1 > \cdots > h_i > \cdots > 1$, which must end with 1. (The algorithm works for any such sequence, though some sequences are known to yield a better efficiency than others. For example, the sequence 1, 4, 13, 40, 121, . . . , used, of course, in reverse, is known to be among the best for this purpose.)

**a.** Apply shellsort to the list

$$S, H, E, L, L, S, O, R, T, I, S, U, S, E, F, U, L$$

**b.** Is shellsort a stable sorting algorithm?

**c.** Implement shellsort, straight insertion sort, selection sort, and bubble sort in the language of your choice and compare their performance on random arrays of sizes $10^n$ for $n = 2, 3, 4, 5$, and 6 as well as on increasing and decreasing arrays of these sizes.

## 4.2 Topological Sorting

In this section, we discuss an important problem for directed graphs, with a variety of applications involving prerequisite-restricted tasks. Before we pose this problem, though, let us review a few basic facts about directed graphs themselves. A ***directed graph***, or ***digraph*** for short, is a graph with directions specified for all its edges (Figure 4.5a is an example). The adjacency matrix and adjacency lists are still two principal means of representing a digraph. There are only two notable differences between undirected and directed graphs in representing them: (1) the adjacency matrix of a directed graph does not have to be symmetric; (2) an edge in a directed graph has just one (not two) corresponding nodes in the digraph's adjacency lists.

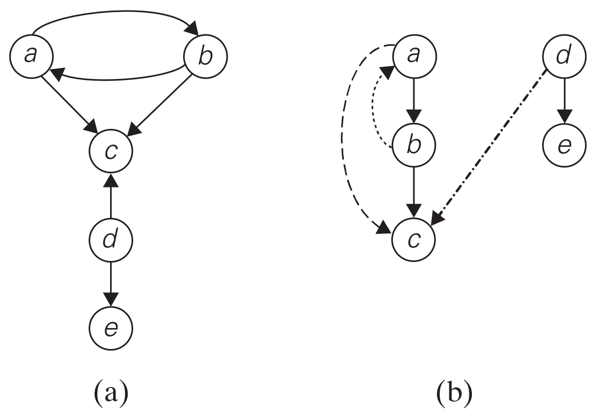


**FIGURE 4.5** (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at $a$.

Depth-first search and breadth-first search are principal traversal algorithms for traversing digraphs as well, but the structure of corresponding forests can be more complex than for undirected graphs. Thus, even for the simple example of Figure 4.5a, the depth-first search forest (Figure 4.5b) exhibits all four types of edges possible in a DFS forest of a directed graph: ***tree edges*** ($ab$, $bc$, $de$), ***back edges*** ($ba$) from vertices to their ancestors, ***forward edges*** ($ac$) from vertices to their descendants in the tree other than their children, and ***cross edges*** ($dc$), which are none of the aforementioned types.

Note that a back edge in a DFS forest of a directed graph can connect a vertex to its parent. Whether or not it is the case, the presence of a back edge indicates that the digraph has a directed cycle. A ***directed cycle*** in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor. For example, $a$, $b$, $a$ is a directed cycle in the digraph in Figure 4.5a. Conversely, if a DFS forest of a digraph has no back edges, the digraph is a ***dag***, an acronym for ***directed acyclic graph***.

Edge directions lead to new questions about digraphs that are either meaningless or trivial for undirected graphs. In this section, we discuss one such question. As a motivating example, consider a set of five required courses {C1, C2, C3, C4, C5} a part-time student has to take in some degree program. The courses can be taken in any order as long as the following course prerequisites are met: C1 and C2 have no prerequisites, C3 requires C1 and C2, C4 requires C3, and C5 requires C3 and C4. The student can take only one course per term. In which order should the student take the courses?

The situation can be modeled by a digraph in which vertices represent courses and directed edges indicate prerequisite requirements (Figure 4.6). In terms of this digraph, the question is whether we can list its vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends. (Can you find such an ordering of this digraph's vertices?) This problem is called ***topological sorting***. It can be posed for an

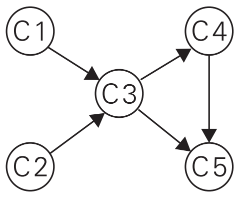


**FIGURE 4.6** Digraph representing the prerequisite structure of five courses.

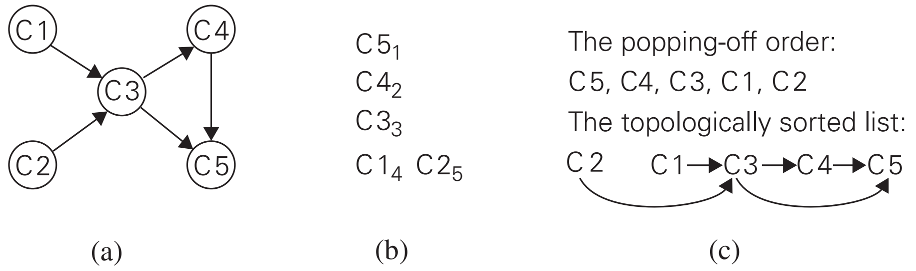


**FIGURE 4.7** (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

arbitrary digraph, but it is easy to see that the problem cannot have a solution if a digraph has a directed cycle. Thus, for topological sorting to be possible, a digraph in question must be a dag. It turns out that being a dag is not only necessary but also sufficient for topological sorting to be possible; i.e., if a digraph has no directed cycles, the topological sorting problem for it has a solution. Moreover, there are two efficient algorithms that both verify whether a digraph is a dag and, if it is, produce an ordering of vertices that solves the topological sorting problem.

The first algorithm is a simple application of depth-first search: perform a DFS traversal and note the order in which vertices become dead-ends (i.e., popped off the traversal stack). Reversing this order yields a solution to the topological sorting problem, provided, of course, no back edge has been encountered during the traversal. If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.

Why does the algorithm work? When a vertex $v$ is popped off a DFS stack, no vertex $u$ with an edge from $u$ to $v$ can be among the vertices popped off before $v$. (Otherwise, $(u, v)$ would have been a back edge.) Hence, any such vertex $u$ will be listed after $v$ in the popped-off order list, and before $v$ in the reversed list.

Figure 4.7 illustrates an application of this algorithm to the digraph in Figure 4.6. Note that in Figure 4.7c, we have drawn the edges of the digraph, and they all point from left to right as the problem's statement requires. It is a convenient way to check visually the correctness of a solution to an instance of the topological sorting problem.

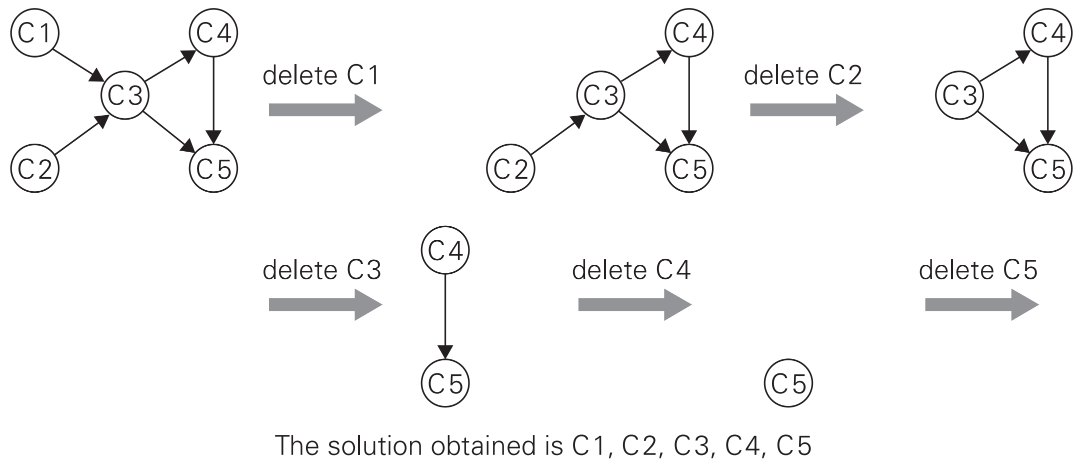


**FIGURE 4.8** Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

The second algorithm is based on a direct implementation of the decrease-(by one)-and-conquer technique: repeatedly, identify in a remaining digraph a ***source***, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it. (If there are several sources, break the tie arbitrarily. If there are none, stop because the problem cannot be solved—see Problem 6a in this section's exercises.) The order in which the vertices are deleted yields a solution to the topological sorting problem. The application of this algorithm to the same digraph representing the five courses is given in Figure 4.8.

Note that the solution obtained by the source-removal algorithm is different from the one obtained by the DFS-based algorithm. Both of them are correct, of course; the topological sorting problem may have several alternative solutions.

The tiny size of the example we used might create a wrong impression about the topological sorting problem. But imagine a large project—e.g., in construction, research, or software development—that involves a multitude of interrelated tasks with known prerequisites. The first thing to do in such a situation is to make sure that the set of given prerequisites is not contradictory. The convenient way of doing this is to solve the topological sorting problem for the project's digraph. Only then can one start thinking about scheduling tasks to, say, minimize the total completion time of the project. This would require, of course, other algorithms that you can find in general books on operations research or in special ones on CPM (Critical Path Method) and PERT (Program Evaluation and Review Technique) methodologies.

As to applications of topological sorting in computer science, they include instruction scheduling in program compilation, cell evaluation ordering in spreadsheet formulas, and resolving symbol dependencies in linkers.