Chapter 4

(Decrease-and-Conquer)

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

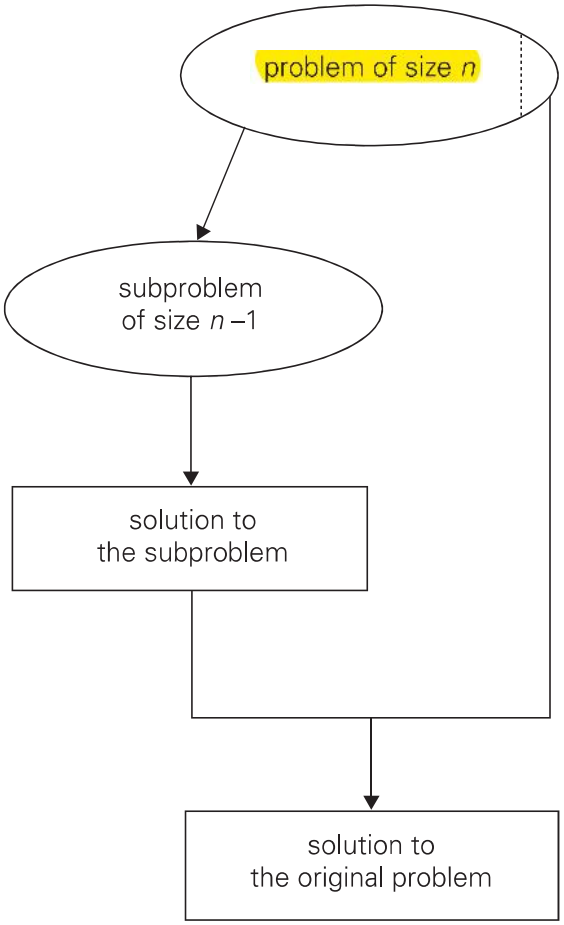


FIGURE 4.1 Decrease-(by one)-and-conquer technique.

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

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:

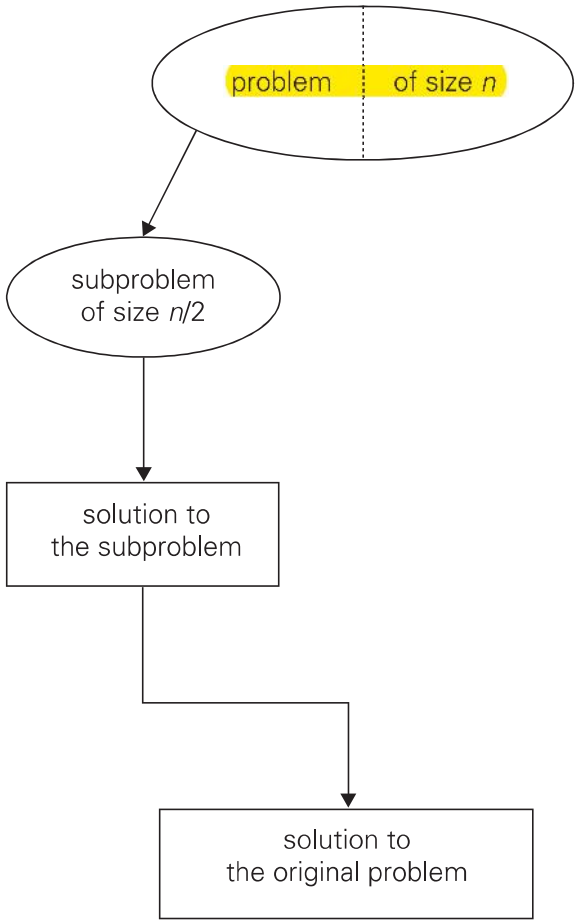


FIGURE 4.2 Decrease-(by half)-and-conquer technique.

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

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 $n-1$: $A[0] \le \cdots \le A[n-2]$. 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 $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \ge 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$

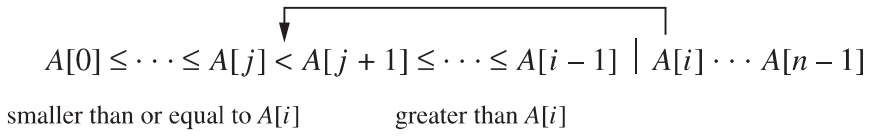


FIGURE 4.3 Iteration of insertion sort: $A[i]$ is inserted in its proper position among the preceding elements previously sorted.

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 ith 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.,

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

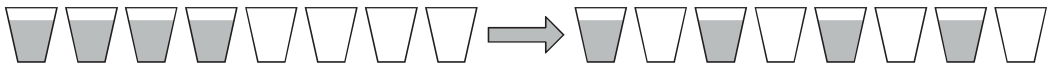
This twice-as-fast average-case performance coupled with an excellent efficiency on almost-sorted arrays makes insertion sort stand out among its principal competitors among elementary sorting algorithms, selection sort and bubble sort. In addition, its extension named ***shellsort***, after its inventor 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.

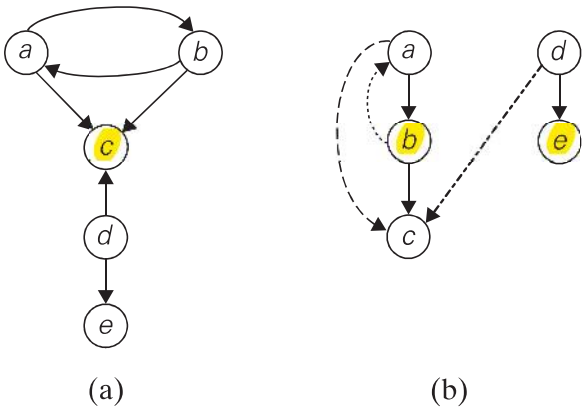


FIGURE 4.5 (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at a.

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

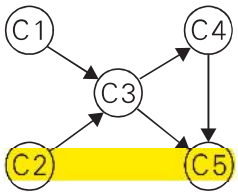


FIGURE 4.6 Digraph representing the prerequisite structure of five courses.

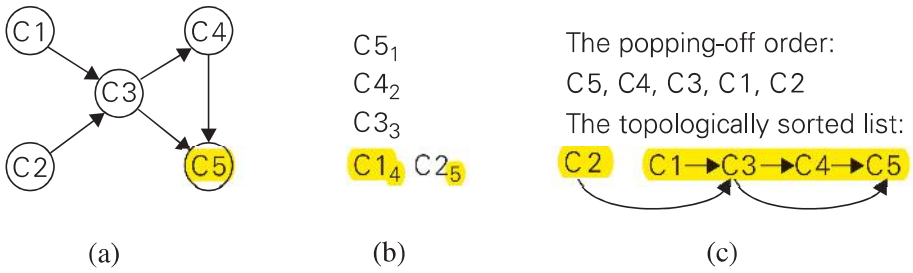


FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

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.

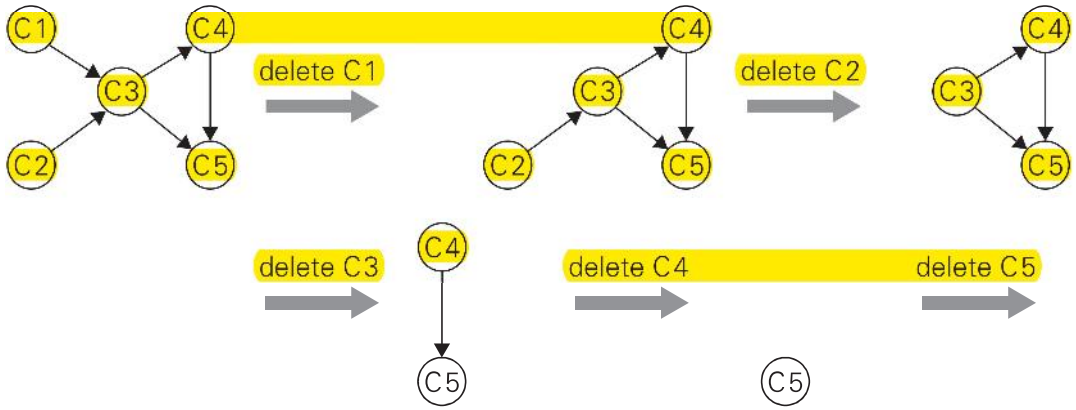


FIGURE 4.8 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

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.

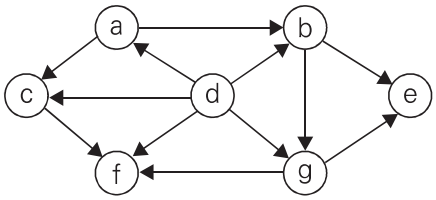
Note that the solution obtained by the source-removal algorithm is different from the one obtained by the DFS-based algorithm. Both of them are correct, of course; the topological sorting problem may have several alternative solutions.

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.

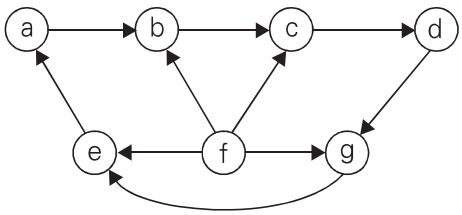
As to applications of topological sorting in computer science, they include instruction scheduling in program compilation, cell evaluation ordering in spreadsheet formulas, and resolving symbol dependencies in linkers.

Exercises 4.2

1. Apply the DFS-based algorithm to solve the topological sorting problem for the following digraphs:



(a)



(b)

2. **a.** Prove that the topological sorting problem has a solution if and only if it is a dag.
   b. For a digraph with n vertices, what is the largest number of distinct solutions the topological sorting problem can have?
3. **a.** What is the time efficiency of the DFS-based algorithm for topological sorting?
   b. How can one modify the DFS-based algorithm to avoid reversing the vertex ordering generated by DFS?
4. Can one use the order in which vertices are pushed onto the DFS stack (instead of the order they are popped off it) to solve the topological sorting problem?
5. Apply the source-removal algorithm to the digraphs of Problem 1 above.
6. **a.** Prove that a nonempty dag must have at least one source.
   b. How would you find a source (or determine that such a vertex does not exist) in a digraph represented by its adjacency matrix? What is the time efficiency of this operation?
   c. How would you find a source (or determine that such a vertex does not exist) in a digraph represented by its adjacency lists? What is the time efficiency of this operation?
7. Can you implement the source-removal algorithm for a digraph represented by its adjacency lists so that its running time is in $O(|V| + |E|)$?
8. Implement the two topological sorting algorithms in the language of your choice. Run an experiment to compare their running times.
9. A digraph is called ***strongly connected*** if for any pair of two distinct vertices u and v there exists a directed path from u to v and a directed path from v to u. In general, a digraph's vertices can be partitioned into disjoint maximal subsets of vertices that are mutually accessible via directed paths; these subsets are called ***strongly connected components*** of the digraph. There are two DFS-

4.4 Decrease-by-a-Constant-Factor Algorithms

You may recall from the introduction to this chapter that decrease-by-a-constant-factor is the second major variety of decrease-and-conquer. As an example of an algorithm based on this technique, we mentioned there exponentiation by squaring defined by formula (4.2). In this section, you will find a few other examples of such algorithms.. The most important and well-known of them is binary search. Decrease-by-a-constant-factor algorithms usually run in logarithmic time, and, being very efficient, do not happen often; a reduction by a factor other than two is especially rare.

Binary Search

Binary search is a remarkably efficient algorithm for searching in a sorted array. It works by comparing a search key K with the array's middle element $A[m]$. If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if $K < A[m]$, and for the second half if $K > A[m]$:

$$\underbrace{A[0] \ldots A[m-1]}_{\text{search here if } K<A[m]} \quad \begin{matrix} K \\ \updownarrow \\ A[m] \end{matrix} \quad \underbrace{A[m+1] \ldots A[n-1]}_{\text{search here if } K>A[m]}.$$

As an example, let us apply binary search to searching for $K = 70$ in the array

3	14	27	31	39	42	55	70	74	81	85	93	98

The iterations of the algorithm are given in the following table:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	l						m						r
iteration 2								l		m			r
iteration 3								l,m	r				

Though binary search is clearly based on a recursive idea, it can be easily implemented as a nonrecursive algorithm, too. Here is pseudocode of this nonrecursive version.

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

```
//Implements nonrecursive binary search
//Input: An array A[0..n − 1] sorted in ascending order and
//       a search key K
//Output: An index of the array's element that is equal to K
//       or −1 if there is no such element
l ← 0;    r ← n − 1
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ← m − 1
    else l ← m + 1
return −1
```

The standard way to analyze the efficiency of binary search is to count the number of times the search key is compared with an element of the array. Moreover, for the sake of simplicity, we will count the so-called three-way comparisons. This assumes that after one comparison of K with $A[m]$, the algorithm can determine whether K is smaller, equal to, or larger than $A[m]$.

How many such comparisons does the algorithm make on an array of n elements? The answer obviously depends not only on n but also on the specifics of a particular instance of the problem. Let us find the number of key comparisons in the worst case $C_{worst}(n)$. The worst-case inputs include all arrays that do not contain a given search key, as well as some successful searches. Since after one comparison the algorithm faces the same situation but for an array half the size, we get the following recurrence relation for $C_{worst}(n)$:

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 1. \tag{4.3}$$

(Stop and convince yourself that $n/2$ must be, indeed, rounded down and that the initial condition must be written as specified.)

We already encountered recurrence (4.3), with a different initial condition, in Section 2.4 (see recurrence (2.4) and its solution there for $n = 2^k$). For the initial condition $C_{worst}(1) = 1$, we obtain

$$C_{worst}(2^k) = k + 1 = \log_2 n + 1. \tag{4.4}$$

Further, similarly to the case of recurrence (2.4) (Problem 7 in Exercises 2.4), the solution given by formula (4.4) for $n = 2^k$ can be tweaked to get a solution valid for an arbitrary positive integer n:

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n + 1) \rceil. \tag{4.5}$$

Formula (4.5) deserves attention. First, it implies that the worst-case time efficiency of binary search is in $\Theta(\log n)$. Second, it is the answer we should have

fully expected: since the algorithm simply reduces the size of the remaining array by about half on each iteration, the number of such iterations needed to reduce the initial size n to the final size 1 has to be about $\log_2 n$. Third, to reiterate the point made in Section 2.1, the logarithmic function grows so slowly that its values remain small even for very large values of n. In particular, according to formula (4.5), it will take no more than $\lceil \log_2(10^3 + 1) \rceil = 10$ three-way comparisons to find an element of a given value (or establish that there is no such element) in any sorted array of one thousand elements, and it will take no more than $\lceil \log_2(10^6 + 1) \rceil = 20$ comparisons to do it for any sorted array of size one million!

What can we say about the average-case efficiency of binary search? A sophisticated analysis shows that the average number of key comparisons made by binary search is only slightly smaller than that in the worst case:

$$C_{avg}(n) \approx \log_2 n.$$

(More accurate formulas for the average number of comparisons in a successful and an unsuccessful search are $C_{avg}^{yes}(n) \approx \log_2 n - 1$ and $C_{avg}^{no}(n) \approx \log_2(n + 1)$, respectively.)

Though binary search is an optimal searching algorithm if we restrict our operations only to comparisons between keys (see Section 11.2), there are searching algorithms (see interpolation search in Section 4.5 and hashing in Section 7.3) with a better average-case time efficiency, and one of them (hashing) does not even require the array to be sorted! These algorithms do require some special calculations in addition to key comparisons, however. Finally, the idea behind binary search has several applications beyond searching (see, e.g., [Ben00]). In addition, it can be applied to solving nonlinear equations in one unknown; we discuss this continuous analogue of binary search, called the method of bisection, in Section 12.4.

Fake-Coin Problem

Of several versions of the fake-coin identification problem, we consider here the one that best illustrates the decrease-by-a-constant-factor strategy. Among n identical-looking coins, one is fake. With a balance scale, we can compare any two sets of coins. That is, by tipping to the left, to the right, or staying even, the balance scale will tell whether the sets weigh the same or which of the sets is heavier than the other but not by how much. The problem is to design an efficient algorithm for detecting the fake coin. An easier version of the problem—the one we discuss here—assumes that the fake coin is known to be, say, lighter than the genuine one.[1]

The most natural idea for solving this problem is to divide n coins into two piles of $\lfloor n/2 \rfloor$ coins each, leaving one extra coin aside if n is odd, and put the two

1. A much more challenging version assumes no additional information about the relative weights of the fake and genuine coins or even the presence of the fake coin among n given coins. We pursue this more difficult version in the exercises for Section 11.2.