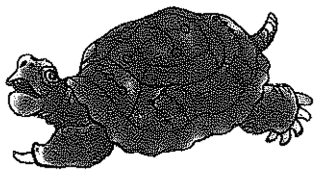
Introduction to The Design & Analysis of Algorithms

2ND EDITION



Anany Levitin

Villanova University

PEARSON Addison Wesley

Boston San Francisco New York
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

8 Dynamic Programming

An idea, like a ghost . . . must be spoken to a little before it will explain itself.
—Charles Dickens (1812–1870)

Dynamic programming is an algorithm design technique with a rather interesting history. It was invented by a prominent U.S. mathematician, Richard Bellman, in the 1950s as a general method for optimizing multistage decision processes. Thus, the word "programming" in the name of this technique stands for "planning" and does not refer to computer programming. After proving its worth as an important tool of applied mathematics, dynamic programming has eventually come to be considered, at least in computer science circles, as a general algorithm design technique that does not have to be limited to special types of optimization problems. It is from this point of view that we will consider this technique here.

Dynamic programming is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the same type. Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which we can then obtain a solution to the original problem.

This technique can be illustrated by revisiting the Fibonacci numbers discussed in Section 2.5. (If you have not read that section, you will be able to follow the discussion anyway. But it is a beautiful topic, so if you feel a temptation to read it, do succumb to it.) The Fibonacci numbers are the elements of the sequence

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34, \ldots,$$

which can be defined by the simple recurrence

$$F(n) = F(n-1) + F(n-2) \quad \text{for } n \geq 2 \qquad \textbf{(8.1)}$$

and two initial conditions

$$F(0) = 0, \ \ F(1) = 1. \qquad \textbf{(8.2)}$$

If we try to use recurrence (8.1) directly to compute the nth Fibonacci number $F(n)$, we would have to recompute the same values of this function many times (see Figure 2.6 for a specific example). Note that the problem of computing $F(n)$ is expressed in terms of its smaller and overlapping subproblems of computing $F(n-1)$ and $F(n-2)$. So we can simply fill elements of a one-dimensional array with the $n+1$ consecutive values of $F(n)$ by starting, in view of initial conditions (8.2), with 0 and 1 and using equation (8.1) as the rule for producing all the other elements. Obviously, the last element of this array will contain $F(n)$. A single-loop pseudocode for this very simple algorithm can be found in Section 2.5.

Note that we can, in fact, avoid using an extra array to accomplish this task by recording the values of just the last two elements of the Fibonacci sequence (see Problem 6 in Exercises 2.5). This phenomenon is not unusual, and we shall encounter it in a few more examples in this chapter. Thus, although a straightforward application of dynamic programming can be interpreted as a special variety of space-for-time tradeoff, a dynamic programming algorithm can sometimes be refined to avoid using extra space.

Certain algorithms compute the nth Fibonacci number without computing all the preceding elements of this sequence (see Section 2.5). It is typical of an algorithm based on the classic bottom-up dynamic programming approach, however, to solve *all* smaller subproblems of a given problem. One variation of the dynamic programming approach seeks to avoid solving unnecessary subproblems. This technique, illustrated in Section 8.4, exploits so-called memory functions and can be considered a top-down variation of dynamic programming.

Whether we use the classical bottom-up version of dynamic programming or its top-down variation, the crucial step in designing such an algorithm remains the same: namely, deriving a recurrence relating a solution to the problem's instance to solutions of its smaller (and overlapping) subinstances. The immediate availability of equation (8.1) for computing the nth Fibonacci number is one of the few exceptions to this rule.

In the sections and exercises of this chapter are a few standard examples of dynamic programming algorithms. (Some of them, in fact, were invented before or independent of the discovery of dynamic programming and only later came to be viewed as examples of this technique's applications.) Numerous other applications range from the optimal way of breaking text into lines (e.g., [Baa00]) to an optimal triangulation of a polygon (e.g., [Ski98]) to a variety of applications to sophisticated engineering problems (e.g., [Bel62], [Ber01]).

8.1 Computing a Binomial Coefficient

Computing a binomial coefficient is a standard example of applying dynamic programming to a nonoptimization problem. You may recall from your studies of elementary combinatorics that the ***binomial coefficient***, denoted $C(n, k)$ or $\binom{n}{k}$, is the number of combinations (subsets) of k elements from an n-element set $(0 \le k \le n)$. The name "binomial coefficients" comes from the participation of these numbers in the binomial formula:

$$(a+b)^n = C(n, 0)a^n + \cdots + C(n, k)a^{n-k}b^k + \cdots + C(n, n)b^n.$$

Of the numerous properties of binomial coefficients, we concentrate on two:

$$C(n, k) = C(n-1, k-1) + C(n-1, k) \quad \text{for } n > k > 0 \tag{8.3}$$

and

$$C(n, 0) = C(n, n) = 1. \tag{8.4}$$

The nature of recurrence (8.3), which expresses the problem of computing $C(n, k)$ in terms of the smaller and overlapping problems of computing $C(n-1, k-1)$ and $C(n-1, k)$, lends itself to solving by the dynamic programming technique. To do this, we record the values of the binomial coefficients in a table of $n+1$ rows and $k+1$ columns, numbered from 0 to n and from 0 to k, respectively (Figure 8.1).

To compute $C(n, k)$, we fill the table in Figure 8.1 row by row, starting with row 0 and ending with row n. Each row i $(0 \le i \le n)$ is filled left to right, starting with 1 because $C(n, 0) = 1$. Rows 0 through k also end with 1 on the table's main diagonal: $C(i, i) = 1$ for $0 \le i \le k$. We compute the other entries by formula (8.3), adding the contents of the cells in the preceding row and the previous column and in the preceding row and the same column. (If you recognize Pascal's triangle—a fascinating mathematical structure usually studied in conjunction with the notion of a combination—you are right: this is exactly what it is.) The following pseudocode implements this algorithm.

	0	1	2	...	$k-1$	k
0	1					
1	1	1				
2	1	2	1			
⋮						
k	1					1
⋮						
$n-1$	1				$C(n-1, k-1)$	$C(n-1, k)$
n	1					$C(n, k)$

FIGURE 8.1 Table for computing the binomial coefficient $C(n, k)$ by the dynamic programming algorithm

```
ALGORITHM  Binomial(n, k)
    //Computes $C(n, k)$ by the dynamic programming algorithm
    //Input: A pair of nonnegative integers $n \ge k \ge 0$
    //Output: The value of $C(n, k)$
    for $i \leftarrow 0$ to $n$ do
        for $j \leftarrow 0$ to $\min(i, k)$ do
            if $j = 0$ or $j = i$
                $C[i, j] \leftarrow 1$
            else $C[i, j] \leftarrow C[i-1, j-1] + C[i-1, j]$
    return $C[n, k]$
```

What is the time efficiency of this algorithm? Clearly, the algorithm's basic operation is addition, so let $A(n, k)$ be the total number of additions made by this algorithm in computing $C(n, k)$. Note that computing each entry by formula (8.3) requires just one addition. Also note that because the first $k + 1$ rows of the table form a triangle while the remaining $n - k$ rows form a rectangle, we have to split the sum expressing $A(n, k)$ into two parts:

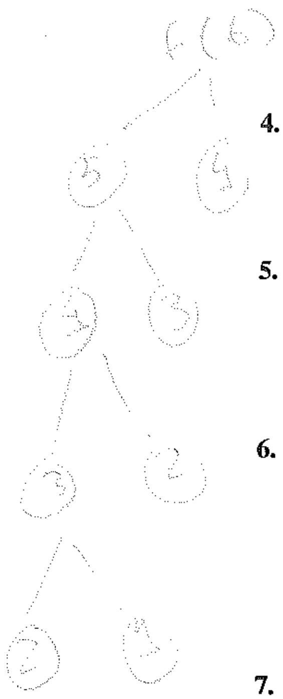
$$A(n, k) = \sum_{i=1}^{k} \sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n} \sum_{j=1}^{k} 1 = \sum_{i=1}^{k} (i - 1) + \sum_{i=k+1}^{n} k$$
$$= \frac{(k-1)k}{2} + k(n - k) \in \Theta(nk).$$

You are asked to ascertain whether this is an efficient algorithm by comparing it with the running times of a few other algorithms for this problem in the exercises. Another problem in the exercises is to analyze whether or not the extra space used by the dynamic programming algorithm is actually necessary.

Exercises 8.1

1. **a.** What does dynamic programming have in common with divide-and-conquer?
   b. What is a principal difference between the two techniques?
2. **a.** Compute $C(6, 3)$ by applying the dynamic programming algorithm.
   b. Is it also possible to compute $C(n, k)$ by filling the algorithm's dynamic programming table column by column rather than row by row?
3. Prove the following assertion made in the text while investigating the time efficiency of the dynamic programming algorithm for computing $C(n, k)$:

$$\frac{(k-1)k}{2} + k(n - k) \in \Theta(nk).$$



4. **a.** What is the space efficiency of *Binomial,* the dynamic programming algorithm for computing $C(n, k)$?

   b. Explain how the space efficiency of this algorithm can be improved. (Try to make as much of an improvement as you can.)

5. **a.** Find the order of growth of the following functions.

   i. $C(n, 1)$ ii. $C(n, 2)$ iii. $C(n, n/2)$ for even n's

   b. What major implication for computing $C(n, k)$ do the answers to the questions in part (a) have?

6. Find the exact number of additions made by the following recursive algorithm based directly on formulas (8.3) and (8.4).

```
ALGORITHM  BinomCoeff(n, k)
    if $k = 0$ or $k = n$ return 1
    else return BinomCoeff$(n - 1, k - 1)$+BinomCoeff$(n - 1, k)$
```

7. Which of the following algorithms for computing a binomial coefficient is most efficient?

   a. Use the formula

$$C(n, k) = \frac{n!}{k!(n-k)!}.$$

   b. Use the formula

$$C(n, k) = \frac{n(n-1)\ldots(n-k+1)}{k!}.$$

   c. Apply recursively the formula

$$C(n, k) = C(n-1, k-1) + C(n-1, k) \quad \text{for } n > k > 0,$$
$$C(n, 0) = C(n, n) = 1.$$

   d. Apply the dynamic programming algorithm.

8. Prove that

$$C(n, k) = C(n, n-k) \quad \text{for } n \geq k \geq 0$$

   and explain how this formula can be utilized in computing $C(n, k)$.

9. *Shortest path counting* A chess rook can move horizontally or vertically to any square in the same row or in the same column of a chessboard. Find the number of shortest paths by which a rook can move from one corner of a chessboard to the diagonally opposite corner. [Gar78], p. 10

   a. by a dynamic programming algorithm.

   b. by using elementary combinatorics.

10. *World Series odds* Consider two teams, A and B, playing a series of games until one of the teams wins n games. Assume that the probability of A winning a game is the same for each game and equal to p, and the probability of A losing a game is $q = 1 - p$. (Hence, there are no ties.) Let $P(i, j)$ be the probability of A winning the series if A needs i more games to win the series and B needs j more games to win the series.

a. Set up a recurrence relation for $P(i, j)$ that can be used by a dynamic programming algorithm.

b. Find the probability of team A winning a seven-game series if the probability of the team winning a game is 0.4.

c. Write a pseudocode of the dynamic programming algorithm for solving this problem and determine its time and space efficiencies.

8.2 Warshall's and Floyd's Algorithms

In this section, we look at two well-known algorithms: Warshall's algorithm for computing the transitive closure of a directed graph and Floyd's algorithm for the all-pairs shortest-paths problem. These algorithms are based on essentially the same idea, which we can interpret as an application of the dynamic programming technique.

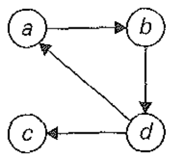
Warshall's Algorithm

Recall that the adjacency matrix $A = \{a_{ij}\}$ of a directed graph is the boolean matrix that has 1 in its ith row and jth column if and only if there is a directed edge from the ith vertex to the jth vertex. We may also be interested in a matrix containing the information about the existence of directed paths of arbitrary lengths between vertices of a given graph.

DEFINITION The ***transitive closure*** of a directed graph with n vertices can be defined as the n-by-n boolean matrix $T = \{t_{ij}\}$, in which the element in the ith row $(1 \le i \le n)$ and the jth column $(1 \le j \le n)$ is 1 if there exists a nontrivial directed path (i.e., a directed path of a positive length) from the ith vertex to the jth vertex; otherwise, t_{ij} is 0.

An example of a digraph, its adjacency matrix, and its transitive closure is given in Figure 8.2.

We can generate the transitive closure of a digraph with the help of depth-first search or breadth-first search. Performing either traversal starting at the ith vertex gives the information about the vertices reachable from the ith vertex and hence the columns that contain ones in the ith row of the transitive closure. Thus, doing such a traversal for every vertex as a starting point yields the transitive closure in its entirety.



(a)

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \left[\begin{array}{cccc} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{array}\right] \end{array}$$

(b)

$$T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \left[\begin{array}{cccc} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{array}\right] \end{array}$$

(c)

FIGURE 8.2 (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

Since this method traverses the same digraph several times, we should hope that a better algorithm can be found. Indeed, such an algorithm exists. It is called ***Warshall's algorithm*** after S. Warshall [War62]. Warshall's algorithm constructs the transitive closure of a given digraph with n vertices through a series of n-by-n boolean matrices:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots, R^{(n)}. \qquad \textbf{(8.5)}$$

Each of these matrices provides certain information about directed paths in the digraph. Specifically, the element $r_{ij}^{(k)}$ in the ith row and jth column of matrix $R^{(k)}$ ($k = 0, 1, \ldots, n$) is equal to 1 if and only if there exists a directed path (of a positive length) from the ith vertex to the jth vertex with each intermediate vertex, if any, numbered not higher than k. Thus, the series starts with $R^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $R^{(0)}$ is nothing else but the adjacency matrix of the digraph. (Recall that the adjacency matrix contains the information about one-edge paths, i.e., paths with no intermediate vertices.) $R^{(1)}$ contains the information about paths that can use the first vertex as intermediate; thus, with more freedom, so to speak, it may contain more ones than $R^{(0)}$. In general, each subsequent matrix in series (8.5) has one more vertex to use as intermediate for its paths than its predecessor and hence may, but does not have to, contain more ones. The last matrix in the series, $R^{(n)}$, reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing else but the digraph's transitive closure.

The central point of the algorithm is that we can compute all the elements of each matrix $R^{(k)}$ from its immediate predecessor $R^{(k-1)}$ in series (8.5). Let $r_{ij}^{(k)}$, the element in the ith row and jth column of matrix $R^{(k)}$, be equal to 1. This means that there exists a path from the ith vertex v_i to the jth vertex v_j with each intermediate vertex numbered not higher than k:

$$v_i, \text{ a list of intermediate vertices each numbered not higher than } k, \; v_j. \qquad \textbf{(8.6)}$$

Two situations regarding this path are possible. In the first, the list of its intermediate vertices does not contain the kth vertex. Then this path from v_i to v_j has intermediate vertices numbered not higher than $k-1$, and therefore $r_{ij}^{(k-1)}$ is equal to 1 as well. The second possibility is that path (8.6) does contain the kth vertex v_k

among the intermediate vertices. Without loss of generality, we may assume that v_k occurs only once in that list. (If it is not the case, we can create a new path from v_i to v_j with this property by simply eliminating all the vertices between the first and last occurrences of v_k in it.) With this caveat, path (8.6) can be rewritten as follows:

$$v_i, \text{ vertices numbered} \le k-1, \ v_k, \text{ vertices numbered} \le k-1, \ v_j.$$

The first part of this representation means that there exists a path from v_i to v_k with each intermediate vertex numbered not higher than $k-1$ (hence $r_{ik}^{(k-1)} = 1$), and the second part means that there exists a path from v_k to v_j with each intermediate vertex numbered not higher than $k-1$ (hence $r_{kj}^{(k-1)} = 1$).

What we have just proved is that if $r_{ij}^{(k)} = 1$, then either $r_{ij}^{(k-1)} = 1$ or both $r_{ik}^{(k-1)} = 1$ and $r_{kj}^{(k-1)} = 1$. It is easy to see that the converse of this assertion is also true. Thus, we have the following formula for generating the elements of matrix $R^{(k)}$ from the elements of matrix $R^{(k-1)}$:

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \textbf{ or } \left(r_{ik}^{(k-1)} \textbf{and } r_{kj}^{(k-1)}\right). \tag{8.7}$$

Formula (8.7) is at the heart of Warshall's algorithm. This formula implies the following rule for generating elements of matrix $R^{(k)}$ from elements of matrix $R^{(k-1)}$, which is particularly convenient for applying Warshall's algorithm by hand:

- If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$.
- If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$. (This rule is illustrated in Figure 8.3.)

As an example, the application of Warshall's algorithm to the digraph in Figure 8.2 is shown in Figure 8.4.

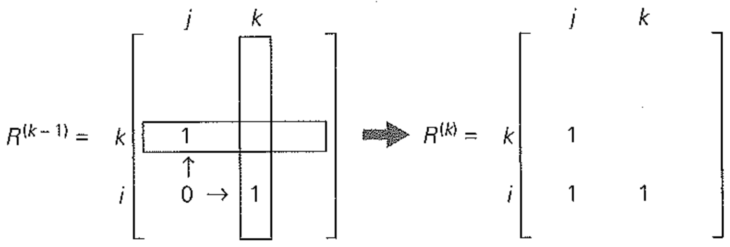
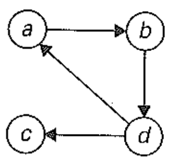


FIGURE 8.3 Rule for changing zeros in Warshall's algorithm



$$R^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

Ones reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$$R^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & \mathbf{1} & 1 & 0 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$$R^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & \mathbf{1} \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & \mathbf{1} \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$$R^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$$R^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & \mathbf{1} & 1 & \mathbf{1} & 1 \\ b & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note five new paths).

FIGURE 8.4 Application of Warshall's algorithm to the digraph shown. New ones are in bold.

Here is a pseudocode of Warshall's algorithm.

ALGORITHM *Warshall*($A[1..n, 1..n]$)

```
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ to $n$ do
    for $i \leftarrow 1$ to $n$ do
        for $j \leftarrow 1$ to $n$ do
            $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ or $(R^{(k-1)}[i, k]$ and $R^{(k-1)}[k, j])$
return $R^{(n)}$
```

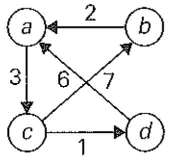
Several observations need to be made about Warshall's algorithm. First, it is remarkably succinct, is it not? Still, its time efficiency is only in $\Theta(n^3)$. In fact, for sparse graphs represented by their adjacency lists, the traversal-based algorithm mentioned at the beginning of this section has a better asymptotic efficiency than Warshall's algorithm (why?). We can speed up the above implementation of Warshall's algorithm for some inputs by restructuring its innermost loop (see Problem 4 in the exercises). Another way to make the algorithm run faster is to treat matrix rows as bit strings and employ the bitwise *or* operation available in most modern computer languages.

As to the space efficiency of Warshall's algorithm, the situation is similar to that of the two earlier examples in this chapter: computing a Fibonacci number and computing a binomial coefficient. Although we used separate matrices for recording intermediate results of the algorithm, this is, in fact, unnecessary. (Problem 3 in the exercises asks you to find a way of avoiding this wasteful use of the computer memory.) Finally, we shall see below how the underlying idea of Warshall's algorithm can be applied to the more general problem of finding lengths of shortest paths in weighted graphs.

Floyd's Algorithm for the All-Pairs Shortest-Paths Problem

Given a weighted connected graph (undirected or directed), the ***all-pairs shortest-paths problem*** asks to find the distances (the lengths of the shortest paths) from each vertex to all other vertices. It is convenient to record the lengths of shortest paths in an n-by-n matrix D called the ***distance matrix***: the element d_{ij} in the ith row and the jth column of this matrix indicates the length of the shortest path from the ith vertex to the jth vertex ($1 \le i, j \le n$). For an example, see Figure 8.5.

We can generate the distance matrix with an algorithm that is very similar to Warshall's algorithm. It is called ***Floyd's algorithm***, after its inventor R. Floyd [Flo62]. It is applicable to both undirected and directed weighted graphs provided that they do not contain a cycle of a negative length. (Of course, in the case of a directed graph, by a path or cycle we mean a directed path or a directed cycle.)



(a)

$$W = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array}$$

(b)

$$D = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 7 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

(c)

FIGURE 8.5 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of n-by-n matrices:

$$D^{(0)}, \dots, D^{(k-1)}, D^{(k)}, \dots, D^{(n)}. \tag{8.8}$$

Each of these matrices contains the lengths of shortest paths with certain constraints on the paths considered for the matrix in question. Specifically, the element $d_{ij}^{(k)}$ in the ith row and the jth column of matrix $D^{(k)}$ ($k = 0, 1, \dots, n$) is equal to the length of the shortest path among all paths from the ith vertex to the jth vertex with each intermediate vertex, if any, numbered not higher than k. In particular, the series starts with $D^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $D^{(0)}$ is nothing but the weight matrix of the graph. The last matrix in the series, $D^{(n)}$, contains the lengths of the shortest paths among all paths that can use all n vertices as intermediate and hence is nothing but the distance matrix being sought.

As in Warshall's algorithm, we can compute all the elements of each matrix $D^{(k)}$ from its immediate predecessor $D^{(k-1)}$ in series (8.8). Let $d_{ij}^{(k)}$ be the element in the ith row and the jth column of matrix $D^{(k)}$. This means that $d_{ij}^{(k)}$ is equal to the length of the shortest path among all paths from the ith vertex v_i to the jth vertex v_j with their intermediate vertices numbered not higher than k:

$$v_i, \text{ a list of intermediate vertices each numbered not higher than } k, \; v_j. \tag{8.9}$$

We can partition all such paths into two disjoint subsets: those that do not use the kth vertex v_k as intermediate and those that do. Since the paths of the first subset have their intermediate vertices numbered not higher than $k-1$, the shortest of them is, by definition of our matrices, of length $d_{ij}^{(k-1)}$.

What is the length of the shortest path in the second subset? If the graph does not contain a cycle of a negative length, we can limit our attention only to the paths in the second subset that use vertex v_k as their intermediate vertex exactly once (because visiting v_k more than once can only increase the path's length). All such paths have the following form:

$$v_i, \text{ vertices numbered} \le k-1, \; v_k, \text{ vertices numbered} \le k-1, \; v_j.$$

In other words, each of the paths is made up of a path from v_i to v_k with each intermediate vertex numbered not higher than $k-1$ and a path from v_k to v_j with each intermediate vertex numbered not higher than $k-1$. The situation is depicted symbolically in Figure 8.6.

Since the length of the shortest path from v_i to v_k among the paths that use intermediate vertices numbered not higher than $k-1$ is equal to $d_{ik}^{(k-1)}$ and the length of the shortest path from v_k to v_j among the paths that use intermediate vertices numbered not higher than $k-1$ is equal to $d_{kj}^{(k-1)}$, the length of the shortest

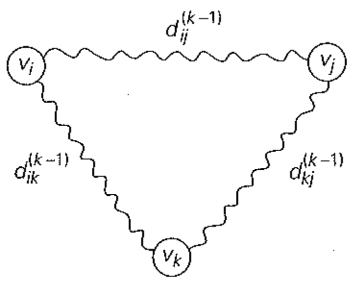


FIGURE 8.6 Underlying idea of Floyd's algorithm

path among the paths that use the kth vertex is equal to $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$. Taking into account the lengths of the shortest paths in both subsets leads to the following recurrence:

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1, \quad d_{ij}^{(0)} = w_{ij}. \qquad \textbf{(8.10)}$$

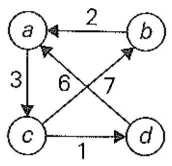
To put it another way, the element in the ith row and the jth column of the current distance matrix $D^{(k-1)}$ is replaced by the sum of the elements in the same row i and the kth column and in the same column j and the kth column if and only if the latter sum is smaller than its current value.

The application of Floyd's algorithm to the graph in Figure 8.5 is illustrated in Figure 8.7.

Here is a pseudocode of Floyd's algorithm. It takes advantage of the fact that the next matrix in sequence (8.8) can be written over its predecessor.

```
ALGORITHM  Floyd(W[1..n, 1..n])
    //Implements Floyd's algorithm for the all-pairs shortest-paths problem
    //Input: The weight matrix W of a graph with no negative-length cycle
    //Output: The distance matrix of the shortest paths' lengths
    $D \leftarrow W$ //is not necessary if W can be overwritten
    for $k \leftarrow 1$ to $n$ do
        for $i \leftarrow 1$ to $n$ do
            for $j \leftarrow 1$ to $n$ do
                $D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$
    return D
```

Obviously, the time efficiency of Floyd's algorithm is cubic—as is the time efficiency of Warshall's algorithm. In the next chapter, we examine Dijkstra's algorithm—another method for finding shortest paths.



$$D^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array}$$

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$$D^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \mathbf{5} & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \mathbf{9} & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e. just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$$D^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & 5 & \infty \\ c & \mathbf{9} & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e. a and *b* (note a new shortest path from *c* to *a*).

$$D^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \mathbf{10} & 3 & \mathbf{4} \\ b & 2 & 0 & 5 & \mathbf{6} \\ c & 9 & 7 & 0 & 1 \\ d & 6 & \mathbf{16} & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e. a, *b*, and *c* (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

$$D^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & \mathbf{7} & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e. a, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

FIGURE 8.7 Application of Floyd's algorithm to the graph shown. Updated elements are shown in bold.

We finish this section with an important general comment. It deals with a general principle that underlines dynamic programming algorithms for optimization problems. Richard Bellman called it the ***principle of optimality***. In terms somewhat different from its original formulation, it says that an optimal solution to any instance of an optimization problem is composed of optimal solutions to its subinstances. The principle of optimality holds more often than not. (To give a rather rare example, it fails for finding longest simple paths.) Although its applicability to a particular problem needs to be verified, of course, such a verification is usually not a principal difficulty in developing a dynamic programming algorithm. The challenge typically lies in figuring out what smaller subinstances need to be considered and in deriving an equation relating a solution to any instance to solutions to its smaller subinstances. We consider a few more examples in the remaining sections of this chapter and their exercises.

Exercises 8.2

1. Apply Warshall's algorithm to find the transitive closure of the digraph defined by the following adjacency matrix

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

2. **a.** Prove that the time efficiency of Warshall's algorithm is cubic.

   b. Explain why the time efficiency of Warshall's algorithm is inferior to that of the traversal-based algorithm for sparse graphs represented by their adjacency lists.

3. Explain how to implement Warshall's algorithm without using extra memory for storing elements of the algorithm's intermediate matrices.

4. Explain how to restructure the innermost loop of the algorithm *Warshall* to make it run faster at least on some inputs.

5. Rewrite the pseudocode of Warshall's algorithm assuming that the matrix rows are represented by bit strings on which the bitwise *or* operation can be performed.

6. **a.** Explain how Warshall's algorithm can be used to determine whether a given digraph is a dag (directed acyclic graph). Is it a good algorithm for this problem?

   b. Is it a good idea to apply Warshall's algorithm to find the transitive closure of an undirected graph?

7. Solve the all-pairs shortest-path problem for the digraph with the weight matrix

$$\begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

8. Prove that the next matrix in sequence (8.8) of Floyd's algorithm can be written over its predecessor.

9. Give an example of a graph or a digraph with negative weights for which Floyd's algorithm does not yield the correct result.

10. Enhance Floyd's algorithm so that shortest paths themselves, not just their lengths, can be found.

11. *Jack Straws* In the game of Jack Straws, a number of plastic or wooden "straws" are dumped on the table and players try to remove them one-by-one without disturbing the other straws. Here, we are only concerned with whether various pairs of straws are connected by a path of touching straws. Given a list of the endpoints for $n > 1$ straws (as if they were dumped on a large piece of graph paper), determine all the pairs of straws that are connected. Note that touching is connecting, but also that two straws can be connected indirectly via other connected straws [1994 East-Central Regionals of the ACM International Collegiate Programming Contest].

8.3 Optimal Binary Search Trees

A binary search tree is one of the most important data structures in computer science. One of its principal applications is to implement a dictionary, a set of elements with the operations of searching, insertion, and deletion. If probabilities of searching for elements of a set are known (e.g., from accumulated data about past searches), it is natural to pose a question about an optimal binary search tree for which the average number of comparisons in a search is the smallest possible. (For simplicity, we limit our discussion to minimizing the average number of comparisons in a successful search. The method can be extended to include unsuccessful searches as well.)

As an example, consider four keys A, B, C, and D to be searched for with probabilities 0.1, 0.2, 0.4, and 0.3, respectively. Figure 8.8 depicts two out of 14 possible binary search trees containing these keys. The average number of comparisons in a successful search in the first of these trees is $0.1 \cdot 1 + 0.2 \cdot 2 + 0.4 \cdot 3 + 0.3 \cdot 4 = 2.9$, while for the second one it is $0.1 \cdot 2 + 0.2 \cdot 1 + 0.4 \cdot 2 + 0.3 \cdot 3 = 2.1$. Neither of these two trees is, in fact, optimal. (Can you tell which binary tree is optimal?)

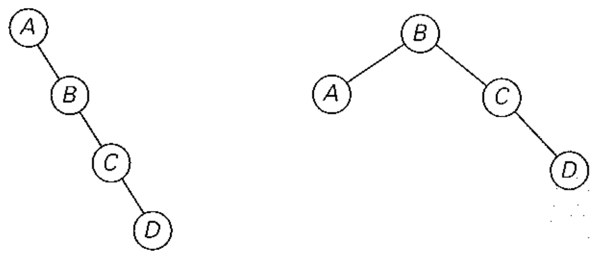


FIGURE 8.8 Two out of 14 possible binary search trees with keys A, B, C, and D

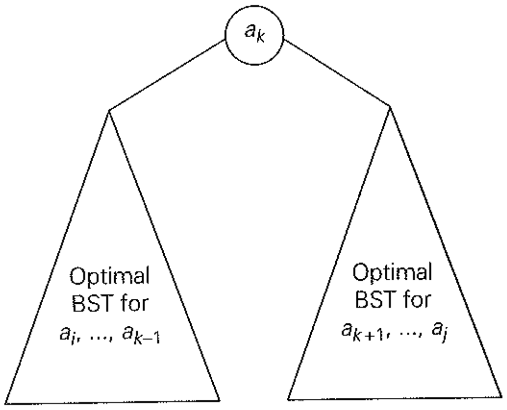

a_k
Optimal BST for $a_i, \ldots, a_{k-1}$
Optimal BST for $a_{k+1}, \ldots, a_j$

FIGURE 8.9 Binary search tree (BST) with root a_k and two optimal binary search subtrees T_i^{k-1} and T_{k+1}^j

For our tiny example, we could find the optimal tree by generating all 14 binary search trees with these keys. As a general algorithm, this exhaustive-search approach is unrealistic: the total number of binary search trees with n keys is equal to the nth ***Catalan number***

$$c(n) = \binom{2n}{n}\frac{1}{n+1} \quad \text{for } n > 0, \quad c(0) = 1,$$

which grows to infinity as fast as $4^n/n^{1.5}$ (see Problem 7 in the exercises).

So let $a_1, \ldots, a_n$ be distinct keys ordered from the smallest to the largest and let $p_1, \ldots, p_n$ be the probabilities of searching for them. Let $C[i, j]$ be the smallest average number of comparisons made in a successful search in a binary search tree T_i^j made up of keys $a_i, \ldots, a_j$, where i, j are some integer indices, $1 \le i \le j \le n$. Thus, following the classic dynamic programming approach, we will find values of $C[i, j]$ for all smaller instances of the problem, although we are interested just in $C[1, n]$. To derive a recurrence underlying the dynamic programming algorithm, we will consider all possible ways to choose a root a_k among the keys $a_i, \ldots, a_j$. For such a binary search tree (Figure 8.9), the root contains key a_k, the left subtree T_i^{k-1} contains keys $a_i, \ldots, a_{k-1}$ optimally arranged, and the right subtree T_{k+1}^j contains keys $a_{k+1}, \ldots, a_j$ also optimally arranged. (Note how we are taking advantage of the principle of optimality here.)

If we count tree levels starting with 1 (to make the comparison numbers equal the keys' levels), the following recurrence relation is obtained:

$$\begin{aligned}
C[i, j] &= \min_{i \le k \le j} \{p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \cdot (\text{level of } a_s \text{ in } T_i^{k-1} + 1) \\
&\qquad + \sum_{s=k+1}^{j} p_s \cdot (\text{level of } a_s \text{ in } T_{k+1}^{j} + 1)\} \\
&= \min_{i \le k \le j} \{p_k + \sum_{s=i}^{k-1} p_s \cdot \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=i}^{k-1} p_s \\
&\qquad + \sum_{s=k+1}^{j} p_s \cdot \text{level of } a_s \text{ in } T_{k+1}^{j} + \sum_{s=k+1}^{j} p_s\} \\
&= \min_{i \le k \le j} \{\sum_{s=i}^{k-1} p_s \cdot \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=k+1}^{j} p_s \cdot \text{level of } a_s \text{ in } T_{k+1}^{j} + \sum_{s=i}^{j} p_s\} \\
&= \min_{i \le k \le j} \{C[i, k-1] + C[k+1, j]\} + \sum_{s=i}^{j} p_s.
\end{aligned}$$

Thus, we have the recurrence

$$C[i, j] = \min_{i \le k \le j} \{C[i, k-1] + C[k+1, j]\} + \sum_{s=i}^{j} p_s \text{ for } 1 \le i \le j \le n. \quad \textbf{(8.11)}$$

We assume in formula (8.11) that $C[i, i-1] = 0$ for $1 \le i \le n+1$, which can be interpreted as the number of comparisons in the empty tree. Note that this formula implies that

$$C[i, i] = p_i \quad \text{for } 1 \le i \le n,$$

as it should be for a one-node binary search tree containing a_i.

The two-dimensional table in Figure 8.10 shows the values needed for computing $C[i, j]$ by formula (8.11): they are in row i and the columns to the left of column j and in column j and the rows below row i. The arrows point to the pairs of entries whose sums are computed in order to find the smallest one to be recorded as the value of $C[i, j]$. This suggests filling the table along its diagonals, starting with all zeros on the main diagonal and given probabilities p_i, $1 \le i \le n$, right above it and moving toward the upper right corner.

The algorithm we sketched computes $C[1, n]$—the average number of comparisons for successful searches in the optimal binary tree. If we also want to get the optimal tree itself, we need to maintain another two-dimensional table to record the value of k for which the minimum in (8.11) is achieved. The table has the same shape as the table in Figure 8.10 and is filled in the same manner, starting with entries $R[i, i] = i$ for $1 \le i \le n$. When the table is filled, its entries indicate indices of the roots of the optimal subtrees, which makes it possible to reconstruct an optimal tree for the entire set given.

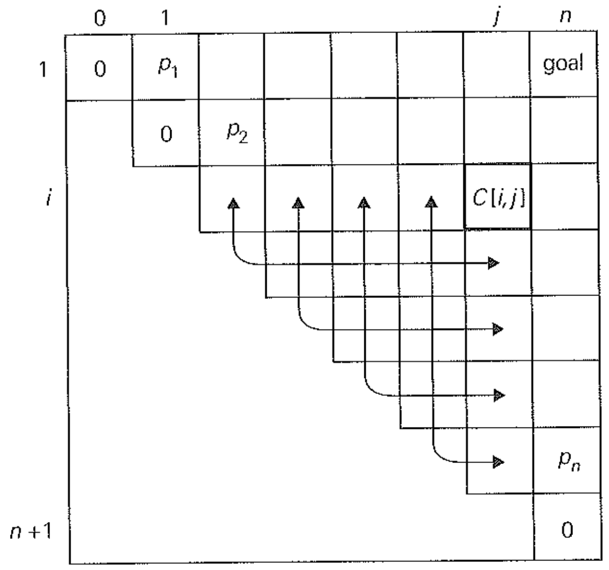


FIGURE 8.10 Table of the dynamic programming algorithm for constructing an optimal binary search tree

EXAMPLE Let us illustrate the algorithm by applying it to the four-key set we used at the beginning of this section:

key	*A*	*B*	*C*	*D*
probability	0.1	0.2	0.4	0.3

The initial tables look like this:

main table

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

Let us compute $C[1, 2]$:

$$C[1, 2] = \min \begin{matrix} k = 1\colon C[1, 0] + C[2, 2] + \sum_{s=1}^{2} p_s = 0 + 0.2 + 0.3 = 0.5 \\ k = 2\colon C[1, 1] + C[3, 2] + \sum_{s=1}^{2} p_s = 0.1 + 0 + 0.3 = 0.4 \end{matrix} = 0.4.$$

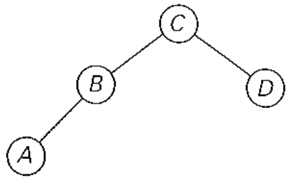


FIGURE 8.11 Optimal binary search tree for the example

Thus, out of two possible binary trees containing the first two keys, A and B, the root of the optimal tree has index 2 (i.e., it contains B), and the average number of comparisons in a successful search in this tree is 0.4.

We will ask you to finish the computations in the exercises. You should arrive at the following final tables:

main table

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					

Thus, the average number of key comparisons in the optimal tree is equal to 1.7. Since $R[1, 4] = 3$, the root of the optimal tree contains the third key, i.e., C. Its left subtree is made up of keys A and B, and its right subtree contains just key D (why?). To find the specific structure of these subtrees, we find first their roots by consulting the root table again as follows. Since $R[1, 2] = 2$, the root of the optimal tree containing A and B is B, with A being its left child (and the root of the one-node tree: $R[1, 1] = 1$). Since $R[4, 4] = 4$, the root of this one-node optimal tree is its only key D. Figure 8.11 presents the optimal tree in its entirety. ▨

Here is a pseudocode of the dynamic programming algorithm.

ALGORITHM *OptimalBST*$(P[1..n])$

```
//Finds an optimal binary search tree by dynamic programming
//Input: An array $P[1..n]$ of search probabilities for a sorted list of $n$ keys
//Output: Average number of comparisons in successful searches in the
//        optimal BST and table $R$ of subtrees' roots in the optimal BST
for $i \leftarrow 1$ to $n$ do
    $C[i, i-1] \leftarrow 0$
    $C[i, i] \leftarrow P[i]$
    $R[i, i] \leftarrow i$
```

```
$C[n+1, n] \leftarrow 0$
for $d \leftarrow 1$ to $n-1$ do //diagonal count
    for $i \leftarrow 1$ to $n-d$ do
        $j \leftarrow i+d$
        $minval \leftarrow \infty$
        for $k \leftarrow i$ to $j$ do
            if $C[i, k-1] + C[k+1, j] < minval$
                $minval \leftarrow C[i, k-1] + C[k+1, j]$; $kmin \leftarrow k$
        $R[i, j] \leftarrow kmin$
        $sum \leftarrow P[i]$; for $s \leftarrow i+1$ to $j$ do $sum \leftarrow sum + P[s]$
        $C[i, j] \leftarrow minval + sum$
return $C[1, n]$, $R$
```

The algorithm's space efficiency is clearly quadratic; the time efficiency of this version of the algorithm is cubic (why?). A more careful analysis shows that entries in the root table are always nondecreasing along each row and column. This limits values for $R[i, j]$ to the range $R[i, j-1], \ldots, R[i+1, j]$ and makes it possible to reduce the running time of the algorithm to $\Theta(n^2)$.

Exercises 8.3

1. Finish the computations started in the section's example of constructing an optimal binary search tree.
2. **a.** Why is the time efficiency of algorithm *OptimalBST* cubic?
   b. Why is the space efficiency of algorithm *OptimalBST* quadratic?
3. Write a pseudocode for a linear-time algorithm that generates the optimal binary search tree from the root table.
4. Devise a way to compute the sums $\sum_{s=i}^{j} p_s$, which are used in the dynamic programming algorithm for constructing an optimal binary search tree, in constant time (per sum).
5. True or false: The root of an optimal binary search tree always contains the key with the highest search probability?
6. How would you construct an optimal binary search tree for a set of n keys if all the keys are equally likely to be searched for? What will be the average number of comparisons in a successful search in such a tree if $n = 2^k$?
7. **a.** Show that the number of distinct binary search trees $b(n)$ that can be constructed for a set of n orderable keys satisfies the recurrence relation

$$b(n) = \sum_{k=0}^{n-1} b(k)b(n-1-k) \quad \text{for } n > 0, \quad b(0) = 1.$$

b. It is known that the solution to this recurrence is given by the Catalan numbers. Verify this assertion for $n = 1, 2, \dots, 5$.

c. Find the order of growth of $b(n)$. What implication does the answer to this question have for the exhaustive-search algorithm for constructing an optimal binary search tree?

8. Design a $\Theta(n^2)$ algorithm for finding an optimal binary search tree.

9. Generalize the optimal binary search algorithm by taking into account unsuccessful searches.

10. ***Matrix chain multiplication*** Consider the problem of minimizing the total number of multiplications made in computing the product of n matrices

$$A_1 \cdot A_2 \cdot \dots \cdot A_n$$

whose dimensions are d_0 by d_1, d_1 by d_2, ..., d_{n-1} by d_n, respectively. Assume that all intermediate products of two matrices are computed by the brute-force (definition-based) algorithm.

a. Give an example of three matrices for which the number of multiplications in $(A_1 \cdot A_2) \cdot A_3$ and $A_1 \cdot (A_2 \cdot A_3)$ differ at least by a factor 1000.

b. How many different ways are there to compute the chained product of n matrices?

c. Design a dynamic programming algorithm for finding an optimal order of multiplying n matrices.

8.4 The Knapsack Problem and Memory Functions

We start this section with designing the dynamic programming algorithm for the knapsack problem: given n items of known weights $w_1, \dots, w_n$ and values $v_1, \dots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack. (This problem was introduced in Section 3.4, where we discussed solving it by an exhaustive-search algorithm.) We assume here that all the weights and the knapsack's capacity are positive integers; the item values do not have to be integers.

To design a dynamic programming algorithm, we need to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller subinstances. Let us consider an instance defined by the first i items, $1 \le i \le n$, with weights $w_1, \dots, w_i$, values $v_1, \dots, v_i$, and knapsack capacity j, $1 \le j \le W$. Let $V[i, j]$ be the value of an optimal solution to this instance, i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j. We can divide all the subsets of the first i items that fit the knapsack of capacity j into two categories: those that do not include the ith item and those that do. Note the following:

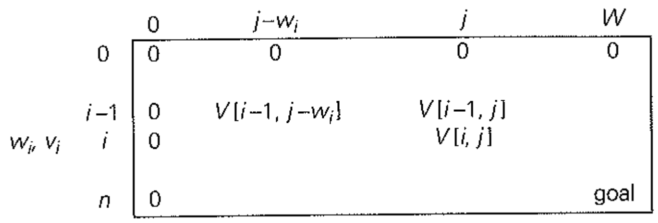


FIGURE 8.12 Table for solving the knapsack problem by dynamic programming

1. Among the subsets that do not include the ith item, the value of an optimal subset is, by definition, $V[i-1, j]$.
2. Among the subsets that do include the ith item (hence, $j - w_i \geq 0$), an optimal subset is made up of this item and an optimal subset of the first $i-1$ items that fit into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + V[i-1, j-w_i]$.

Thus, the value of an optimal solution among all feasible subsets of the first i items is the maximum of these two values. Of course, if the ith item does not fit into the knapsack, the value of an optimal subset selected from the first i items is the same as the value of an optimal subset selected from the first $i-1$ items. These observations lead to the following recurrence:

$$V[i, j] = \begin{cases} \max\{V[i-1, j],\ v_i + V[i-1, j-w_i]\} & \text{if } j - w_i \geq 0 \\ V[i-1, j] & \text{if } j - w_i < 0. \end{cases} \quad \textbf{(8.12)}$$

It is convenient to define the initial conditions as follows:

$$V[0, j] = 0 \text{ for } j \geq 0 \text{ and } V[i, 0] = 0 \text{ for } i \geq 0. \quad \textbf{(8.13)}$$

Our goal is to find $V[n, W]$, the maximal value of a subset of the n given items that fit into the knapsack of capacity W, and an optimal subset itself.

Figure 8.12 illustrates the values involved in equations (8.12) and (8.13). For $i, j > 0$, to compute the entry in the ith row and the jth column, $V[i, j]$, we compute the maximum of the entry in the previous row and the same column and the sum of v_i and the entry in the previous row and w_i columns to the left. The table can be filled either row by row or column by column.

EXAMPLE 1 Let us consider the instance given by the following data:

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4	0	10	15	25	30	37

FIGURE 8.13 Example of solving an instance of the knapsack problem by the dynamic programming algorithm

The dynamic programming table, filled by applying formulas (8.12) and (8.13), is shown in Figure 8.13.

Thus, the maximal value is $V[4, 5] = \$37$. We can find the composition of an optimal subset by tracing back the computations of this entry in the table. Since $V[4, 5] \neq V[3, 5]$, item 4 was included in an optimal solution along with an optimal subset for filling $5 - 2 = 3$ remaining units of the knapsack capacity. The latter is represented by element $V[3, 3]$. Since $V[3, 3] = V[2, 3]$, item 3 is not a part of an optimal subset. Since $V[2, 3] \neq V[1, 3]$, item 2 is a part of an optimal selection, which leaves element $V[1, 3 - 1]$ to specify its remaining composition. Similarly, since $V[1, 2] \neq V[0, 2]$, item 1 is the final part of the optimal solution {item 1, item 2, item 4}. ▨

The time efficiency and space efficiency of this algorithm are both in $\Theta(nW)$. The time needed to find the composition of an optimal solution is in $O(n + W)$. You are asked to prove these assertions in the exercises.

Memory Functions

As we discussed at the beginning of this chapter and illustrated in subsequent sections, dynamic programming deals with problems whose solutions satisfy a recurrence relation with overlapping subproblems. The direct top-down approach to finding a solution to such a recurrence leads to an algorithm that solves common subproblems more than once and hence is very inefficient (typically, exponential or worse). The classic dynamic programming approach, on the other hand, works bottom-up: it fills a table with solutions to *all* smaller subproblems, but each of them is solved only once. An unsatisfying aspect of this approach is that solutions to some of these smaller subproblems are often not necessary for getting a solution to the problem given. Since this drawback is not present in the top-down approach, it is natural to try to combine the strengths of the top-down and bottom-up approaches. The goal is to get a method that solves only subproblems which are necessary and does it only once. Such a method exists; it is based on using ***memory functions*** [Bra96].

This method solves a given problem in the top-down manner but, in addition, maintains a table of the kind that would have been used by a bottom-up dynamic programming algorithm. Initially, all the table's entries are initialized with a special "null" symbol to indicate that they have not yet been calculated. Thereafter, whenever a new value needs to be calculated, the method checks the corresponding entry in the table first: if this entry is not "null," it is simply retrieved from the table; otherwise, it is computed by the recursive call whose result is then recorded in the table.

The following algorithm implements this idea for the knapsack problem. After initializing the table, the recursive function needs to be called with $i = n$ (the number of items) and $j = W$ (the capacity of the knapsack).

ALGORITHM *MFKnapsack*(i, j)

```
//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer $i$ indicating the number of the first
//        items being considered and a nonnegative integer $j$ indicating
//        the knapsack's capacity
//Output: The value of an optimal feasible subset of the first $i$ items
//Note: Uses as global variables input arrays $Weights[1..n]$, $Values[1..n]$,
//and table $V[0..n, 0..W]$ whose entries are initialized with $-1$'s except for
//row 0 and column 0 initialized with 0's
if $V[i, j] < 0$
    if $j < Weights[i]$
        $value \leftarrow MFKnapsack(i-1, j)$
    else
        $value \leftarrow \max(MFKnapsack(i-1, j),$
                      $Values[i] + MFKnapsack(i-1, j - Weights[i]))$
    $V[i, j] \leftarrow value$
return $V[i, j]$
```

EXAMPLE 2 Let us apply the memory function method to the instance considered in Example 1. Figure 8.14 gives the results. Only 11 out of 20 nontrivial values (i.e., not those in row 0 or in column 0) have been computed. Just one nontrivial entry, $V[1, 2]$, is retrieved rather than being recomputed. For larger instances, the proportion of such entries can be significantly larger. ▩

In general, we cannot expect more than a constant-factor gain in using the memory function method for the knapsack problem because its time efficiency class is the same as that of the bottom-up algorithm (why?). A more significant improvement can be expected for dynamic programming algorithms in which a computation of one value takes more than constant time. You should also keep in mind that a memory function method may be less space-efficient than a space-efficient version of a bottom-up algorithm.

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	–	12	22	–	22
$w_3 = 3, v_3 = 20$	3	0	–	–	22	–	32
$w_4 = 2, v_4 = 15$	4	0	–	–	–	–	37

FIGURE 8.14 Example of solving an instance of the knapsack problem by the memory function algorithm

Exercises 8.4

1. **a.** Apply the bottom-up dynamic programming algorithm to the following instance of the knapsack problem:

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

capacity $W = 6$.

   b. How many different optimal subsets does the instance of part (a) have?

   c. In general, how can we use the table generated by the dynamic programming algorithm to tell whether there is more than one optimal subset for the knapsack problem's instance?

$\rightarrow j - w <$
$\rightarrow j - w >$

2. **a.** Write a pseudocode of the bottom-up dynamic programming algorithm for the knapsack problem.

   b. Write a pseudocode of the algorithm that finds the composition of an optimal subset from the table generated by the bottom-up dynamic programming algorithm for the knapsack problem.

```
for i = n to 1
  IF (V[i, j] > V[i-1, j])
  { k = k+1
      (T[k] = i
    j = j - wi
  }
```

3. For the bottom-up dynamic programming algorithm for the knapsack problem, prove that
   a. its time efficiency is in $\Theta(nW)$.
   b. its space efficiency is in $\Theta(nW)$.
   c. the time needed to find the composition of an optimal subset from a filled dynamic programming table is in $O(n + W)$.

4. **a.** True or false: A sequence of values in a row of the dynamic programming table for an instance of the knapsack problem is always nondecreasing?

b. True or false: A sequence of values in a column of the dynamic programming table for an instance of the knapsack problem is always nondecreasing?

5. Apply the memory function method to the instance of the knapsack problem given in Problem 1. Indicate the entries of the dynamic programming table that are: (i) never computed by the memory function method on this instance; (ii) retrieved without a recomputation.

6. Prove that the efficiency class of the memory function algorithm for the knapsack problem is the same as that of the bottom-up algorithm (see Problem 3).

7. Write a pseudocode of a memory function for the optimal binary search tree problem. (You may limit your function to finding the smallest number of key comparisons in a successful search.)

8. Give two reasons why the memory function approach is unattractive for the problem of computing a binomial coefficient.

9. Design a dynamic programming algorithm for the ***change-making problem***: given an amount n and unlimited quantities of coins of each of the denominations $d_1, d_2, \ldots, d_m$, find the smallest number of coins that add up to n or indicate that the problem does not have a solution.

10. Write a research report on one of the following well-known applications of dynamic programming:

    a. finding the longest common subsequence in two sequences

    b. optimal string editing

    c. minimal triangulation of a polygon

SUMMARY

- *Dynamic programming* is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the same type. Dynamic programming suggests solving each smaller subproblem once and recording the results in a table from which a solution to the original problem can be then obtained.

- Applicability of dynamic programming to an optimization problem requires the problem to satisfy the *principle of optimality*: an optimal solution to any of its instances must be made up of optimal solutions to its subinstances.

- Computing a binomial coefficient via constructing the Pascal triangle can be viewed as an application of the dynamic programming technique to a nonoptimization problem.

- *Warshall's algorithm* for finding the transitive closure and *Floyd's algorithm* for the all-pairs shortest-paths problem are based on the idea that can be interpreted as an application of the dynamic programming technique.
- Dynamic programming can be used for constructing an *optimal binary search tree* for a given set of keys and known probabilities of searching for them.
- Solving a knapsack problem by a dynamic programming algorithm exemplifies an application of this technique to difficult problems of combinatorial optimization.
- The *memory function* technique seeks to combine strengths of the top-down and bottom-up approaches to solving problems with overlapping subproblems. It does this by solving, in the top-down fashion but only once, just necessary subproblems of a given problem and recording their solutions in a table.