This file contains the exercises, hints, and solutions for Chapter 8 of the book "Introduction to the Design and Analysis of Algorithms," 3rd edition, by A. Levitin. The problems that might be challenging for at least some students are marked by ▷; those that might be difficult for a majority of students are marked by ►.

Exercises 8.1

1. What does dynamic programming have in common with divide-and-conquer? What is a principal difference between them?

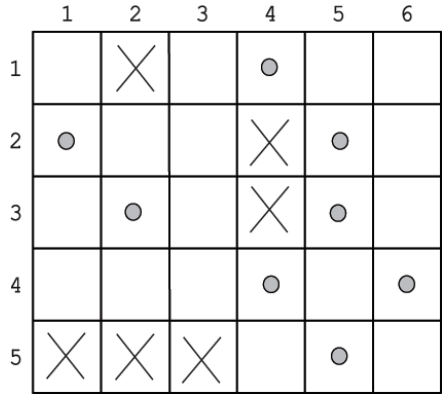
2. Solve the instance 5, 1, 2, 10, 6 of the coin-row problem.

3. a. Show that the time efficiency of solving the coin-row problem by straightforward application of recurrence (8.3) is exponential.

   b. Show that the time efficiency of solving the coin-row problem by exhaustive search is at least exponential.

4. Apply the dynamic programming algorithm to find all the solutions to the change-making problem for the denominations 1, 3, 5 and the amount $n = 9$.

5. How would you modify the dynamic programming algorithm for the coin-collecting problem if some cells on the board are inaccessible for the robot? Apply your algorithm to the board below, where the inaccessible cells are shown by X's. How many optimal paths are there for this board?



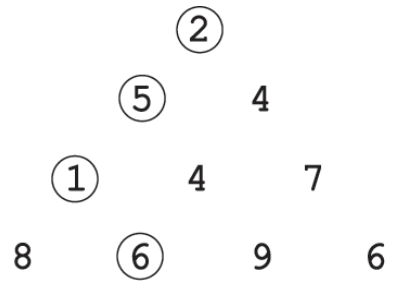
6. ▷ *Rod-cutting problem* Design a dynamic programming algorithm for the following problem. Find the maximum total sale price that can be obtained by cutting a rod of n units long into integer-length pieces if the sale price of a piece i units long is p_i for $i = 1, 2, ..., n$. What are the time and space efficiencies of your algorithm?

7. *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. The length of a path is measured by the number of squares it passes through, including the first and the last squares. Solve the problem

   (a) by a dynamic programming algorithm.

   (b) by using elementary combinatorics.

8. *Minimum-sum descent* Some positive integers are arranged in an equilateral triangle with n numbers in its base like the one shown in the figure below for $n = 4$. The problem is to find the smallest sum in a descent from the triangle apex to its base through a sequence of adjacent numbers (shown in the figure by the circles). Design a dynamic programming algorithm for this problem and indicate its time efficiency.



9. *Binomial coefficient* Design an efficient algorithm for computing the binomial coefficient $C(n, k)$ that uses no multiplications. What are the time and space efficiencies of your algorithm?

10. *Longest path in a dag* a. Design an efficient algorithm for finding the length of a longest path in a dag. (This problem is important both as a prototype of many other dynamic programming applications and in its own right because it determines the minimal time needed for completing a project comprising precedence-constrained tasks.)

    ▷ b. Show how to reduce the coin-row problem discussed in this section to the problem of finding a longest path in a dag.

11. ▶ *Maximum square submatrix* Given an $m \times n$ boolean matrix B, find its largest square submatrix whose elements are all zeros. Design a dynamic programming algorithm and indicate its time efficiency. (The algorithm may be useful for, say, finding the largest free square area on a computer screen or for selecting a construction site.)

12. ▶ *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 it winning a game is 0.4.

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

Hints to Exercises 8.1

1. Compare the definitions of the two techniques.

2. Use the table generated by the dynamic programming algorithm in solving the problem's instance in Example 1 in the section.

3. a. The analysis is similar to that of the top-down recursive computation of the nth Fibonacci number in Section 2.5.

   b. Set up and solve a recurrence for the number of candidate solutions that need to be processed by the exhaustive search algorithm.

4. Apply the dynamic programming algorithm to the instance given as it is done in Example 2 of the section. Note that there are two optimal coin combinations here.

5. Adjust formula (8.5) for inadmissible cells and their immediate neighbors.

6. The problem is similar to the change-making problem discussed in the section.

7. a. Relate the number of the rook's shortest paths to the square in the ith row and the jth column of the chessboard to the numbers of the shortest paths to the adjacent squares.

   b. Consider one shortest path as 14 consecutive moves to adjacent squares.

8. One can solve the problem in quadratic time.

9. Use a well-known formula from elementary combinatorics relating $C(n, k)$ to smaller binomial coefficients.

10. a. Topologically sort the dag's vertices first.

    b. Create a dag with $n + 1$ vertices: one vertex to start and the others to represent the coins given.

11. Let $F(i, j)$ be the order of the largest all-zero submatrix of a given matrix with its low right corner at (i, j) Set up a recurrence relating $F(i, j)$ to $F(i - 1, j)$, $F(i, j - 1)$, and $F(i - 1, j - 1)$.

12. a. In the situation where teams A and B need i and j games, respectively, to win the series, consider the result of team A winning the game and the result of team A losing the game.

    b. Set up a table with five rows ($0 \leq i \leq 4$) and five columns ($0 \leq j \leq 4$) and fill it by using the recurrence derived in part (a).

    c. Your pseudocode should be guided by the recurrence set up in part

(a). The efficiency answers follow immediately from the table's size and the time spent on computing each of its entries.

Solutions to Exercises 8.1

1. Both techniques solve a problem by dividing it into several subproblems. But smaller subproblems in divide-and-conquer do not overlap; therefore their solutions are not stored for reuse. Smaller subproblems in dynamic programming do overlap; therefore their solutions are stored for reuse.

2. The application of the dynamic programming algorithm to the input 5, 1, 2, 10, 6, 2 in section 8.1 yielded the following table:

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	17

   Using the data in the first six columns, we conclude that the largest amount of money that can be obtained for the input 5, 1, 2, 10, 6 is $F(5) = 15$, which is obtained by taking coins $c_4 = 10$ and $c_1 = 5$.

3. a. The time efficiency analysis of the algorithm in question is identical to that of the top-down computation of the nth Fibonacci number in Section 2.5: see recurrence (2.11) for the number of additions made by both algorithms. Hence, the time efficiency class is $\Theta(\phi^n)$ where $\phi = (1+\sqrt{5})/2$.

   b. If an exhaustive search algorithm generates all the subsets of the coin row given before checking which of them don't include adjacent coins, the number of the subsets will be equal 2^n, which answers the question. But even the number of subsets with no adjacent coins is exponential as well. Indeed, let $S(n)$ be the number of such subsets including the empty one. They can be divided into the subsets that contain the first coin and the subsets that do not contain it. The number of the subsets of the first kind is equal to $S(n-2)$; the number of the subsets of the second kind is equal to $S(n-1)$. Hence we have the recurrence

$$S(n) = S(n-2) + S(n-1) \quad \text{for } n > 2, \quad S(1) = 2, \quad S(2) = 3.$$

   It's easy to see that the terms of the sequence $S(n)$, defined by these formulas, are nothing else but the Fibonacci numbers running two terms "ahead" of their canonical counterparts defined the initial conditions $F(0) = 0$ and $F(1) = 1$. Hence $S(n) = F(n+2) \in \Theta(\phi^{n+2}) = \Theta(\phi^n)$, which answers the question posed by the exercise.

4. The application of the dynamic programming algorithm to the instance

given yields the following table

$F[0] = 0$

n	0	1	2	3	4	5	6	7	8	9
F	0									

$F[1] = \min\{F[1-1]\} + 1 = 1$

n	0	1	2	3	4	5	6	7	8	9
F	0	1								

$F[2] = \min\{F[2-1]\} + 1 = 2$

n	0	1	2	3	4	5	6	7	8	9
F	0	1	2							

$F[3] = \min\{F[3-1],\ F[3-3]\} + 1 = 1$

n	0	1	2	3	4	5	6	7	8	9
F	0	1	2	1						

$F[4] = \min\{F[4-1],\ F[4-3]\} + 1 = 2$

n	0	1	2	3	4	5	6	7	8	9
F	0	1	2	1	2					

$F[5] = \min\{F[5-1],\ F[5-3],\ F[5-5]\} + 1 = 2$

n	0	1	2	3	4	5	6	7	8	9
F	0	1	2	1	2	1				

$F[6] = \min\{F[6-1],\ F[6-3],\ F[6-5]\} + 1 = 2$

n	0	1	2	3	4	5	6	7	8	9
F	0	1	2	1	2	1	2			

$F[7] = \min\{F[7-1],\ F[7-3],\ F[7-5]\} + 1 = 3$

n	0	1	2	3	4	5	6	7	8	9
F	0	1	2	1	2	1	2	3		

$F[8] = \min\{F[8-1],\ F[8-3],\ F[8-5]\} + 1 = 2$

n	0	1	2	3	4	5	6	7	8	9
F	0	1	2	1	2	1	2	3	2	

$F[9] = \min\{F[9-1],\ F[9-3],\ F[9-5]\} + 1 = 3$

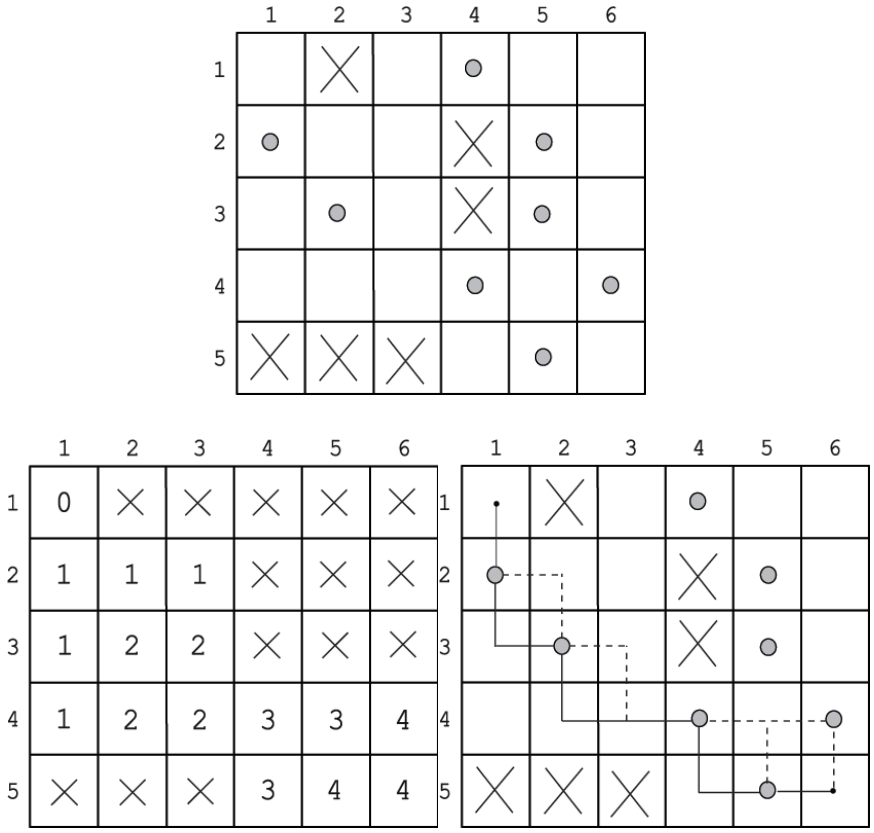
n	0	1	2	3	4	5	6	7	8	9
F	0	1	2	1	2	1	2	3	2	**3**

Application of Algorithm *MinCoinChange* to amount $n = 9$ and coin denominations 1, 3, and 5

The minimum number of coins obtained is $F(9) = 3$. There are two optimal coin sets: $\{1, 3, 5\}$ and $\{3, 3, 3\}$.

5. Formula (8.5) used for computing the largest number of coins that can be brought to a cell needs to be adjusted as follows. If a cell is inadmissible or has no admissible neighbors above or to the left of it, it is marked as inadmissible (if it hasn't been already marked as such) and no value is computed for it. If a cell has just one admissible neighbor above or to the left of it, only this value is used in formula (8.5). Otherwise, the algorithm proceeds filling the table exactly the same way as it's done in the section. The results of its application to the board given are shown below. The largest number of coins that can be brought by the robot to the lower

right corner is 4; there are 12 different paths for the robot to do this.



Application of the dynamic programming algorithm to the board shown in the top figure.

.

6. Let $F(n)$ be the maximum price for a given rod of length n. We have the following recurrence for its values:

$$\begin{aligned} F(n) &= \max_{1 \le j \le n} \{p_j + F(n-j)\} \quad \text{for } n > 0, \\ F(0) &= 0. \end{aligned}$$

Using this recurrence, we can fill a one-dimensional array with $n+1$ consecutive values of F. The last value, $F(n)$, will be the maximum possible price in question. Since computing each value of $F(i)$ requires i additions (and i integer subtractions), the total number of additions is equal to $1 + 2 + \cdots + n = n(n+1)/2$, making the algorithm's time efficiency quadratic. The space efficiency of the algorithm is obviously linear since it uses an additional array of $n+1$ elements.

Actual cuts yielding the maximum price can be obtained by backtracing (see the examples in Section 8.1 and the solution to Problem 5 in these exercises).

Note: The rod-cutting problem is discussed in Cormen et al. *Introduction to Algorithms, 3rd edition,* Section 15.1.

7. a. With no loss of generality, we can assume that the rook is initially located in the lower left corner of a chessboard, whose rows and columns are numbered from 1 to 8 bottom up and left to right, respectively. Let $P(i, j)$ be the number of the rook's shortest paths from square (1,1) to square (i, j) in the ith row and the jth column, where $1 \leq i, j \leq 8$. Any such path will be composed of vertical and horizontal moves directed toward the goal. Obviously, $P(i, 1) = P(1, j) = 1$ for any $1 \leq i, j \leq 8$. In general, any shortest path to square (i, j) is reached either from its left neighbor, i.e., square $(i, j - 1)$, or from its neighbors below, i.e., square $(i - 1, j)$. Hence we have the following recurrence

$$\begin{aligned} P(i, j) &= P(i, j - 1) + P(i - 1, j) \text{ for } 1 < i, j \leq 8, \\ P(i, 1) &= P(1, j) = 1 \text{ for } 1 \leq i, j \leq 8. \end{aligned}$$

Using this recurrence, we can compute the values of $P(i, j)$ for each square $(i.j)$ of the board. This can be done either row by row, or column by column, or diagonal by diagonal. (One can also take advantage of the board's symmetry to make the computations only for the squares either on and above or on and below the board's main diagonal.) The results are given in the diagram below:

1	8	36	120	330	792	1716	3432
1	7	28	84	210	462	924	1716
1	6	21	56	126	252	462	792
1	5	15	35	70	126	210	330
1	4	10	20	35	56	84	120
1	3	6	10	15	21	28	36
1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1

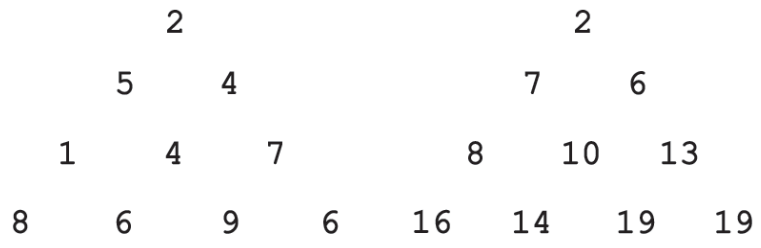
b. Any shortest path from square (1,1) to square (8,8) can be thought of as 14 consecutive moves to adjacent squares, seven of which being up while the other seven being to the right. For example, the shortest path composed of the vertical move from (1,1) to (8,1) followed by the horizontal move from (8,1) to (8,8) corresponds to the following sequence of 14 one-square moves:

$$(u, u, u, u, u, u, u, r, r, r, r, r, r, r),$$

where u and r stand for a move up and to the right, respectively. Hence, the total number of distinct shortest paths is equal to the number of different ways to choose seven u-positions among the total of 14 possible positions, which is equal to $C(14, 7)$.

Note: The problem is discussed in Martin Gardner's *aha!Insight*, Scientific American/W.H.Freeman, p. 10.

8. Using the standard dynamic programming technique, compute the minimum sum along a descending path from the apex to each number in the triangle. Start with the apex, for which this sum is obviously equal to the number itself. Then compute the sums moving top down and, say, left to right across the triangle's rows as follows. For any number that is either the first or the last in its row, add the sum previously computed for the adjacent number in the preceding row and the number itself; for any number that is neither the first nor the last in its row, add the smaller of the previously computed sums for the two adjacent numbers in the preceding row and the number itself. When all such sums are computed for the numbers at the base of the triangle, find the smallest among them.
The figure below illustrates the algorithm for the triangle given in the problem's statement.



(a) (b)

Application of the dynamic programming algorithm for the minimum-sum descent problem: (a) input triangle; (b) triangle of minimum sums along descending paths with 14 being the smallest

The time efficiency of the algorithm is obviously quadratic: it spends constant time to find a minimum sum on a path to each of $n(n+1)/2$ numbers in the triangle and then needs a linear time to find the smallest among n such sums for the base of the triangle.

Note: The problem is analogous to Problem 18 on the Project Euler website at *projecteuler.net* (accessed February 14, 2011) .

9. The recurrence underlying the algorithm in question is

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

Here is pseudocode of the dynamic programming algorithm based on these formulas.

Algorithm *Binomial*(n,k)

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

The algorithm computes all the binomial coefficients $C(i,j)$ for $0 \le i \le n$ and $0 \le j \le \min(i,k)$, which fill the triangular table with $k+1$ rows followed by a rectangular table with $n-k$ rows and $k+1$ columns. One addition is made to compute each binomial coefficient, except those columns 0 and k in the triangular table and those in column 0 in the rectangular table. Therefore we can compute $A(n,k)$, the total number of additions made by this algorithm in computing $C(n,k)$, as follows:

$$\begin{aligned} A(n,k) &= \sum_{i=1}^{k} \sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n} \sum_{j=1}^{k} 1 = \sum_{i=1}^{k} (i-1) + \sum_{i=k+1}^{n} k \\ &= \frac{(k-1)k}{2} + k(n-k). \end{aligned}$$

The simple algebra yields

$$\frac{(k-1)k}{2} + k(n-k) = nk - \frac{1}{2}k^2 - \frac{1}{2}k.$$

So, we can obtain an upper bound by eliminating the negative terms:

$$nk - \frac{1}{2}k^2 - \frac{1}{2}k \le nk \quad \text{for all } n, k \ge 0.$$

We can get a lower bound by considering $n \ge 2$ and $0 \le k \le n$:

$$nk - \frac{1}{2}k^2 - \frac{1}{2}k \ge nk - \frac{1}{2}nk - \frac{1}{2}k\frac{1}{2}n = \frac{1}{4}nk.$$

Hence $A(n, k) \in \Theta(nk)$, which indicates the time efficiency class of the algorithm.

The space efficiency of the above algorithm is also $\Theta(nk)$ since the computed binomial coefficients occupy their own memory cells in the rectangular table with $n + 1$ rows and $k + 1$ columns. Considering only the memory cells actually used by the algorithm still yields the same asymptotic class:

$$S(n, k) = \sum_{i=0}^{k}(i + 1) + \sum_{i=k+1}^{n}(k + 1) = \frac{(k + 1)(k + 2)}{2} + (k + 1)(n - k) \in \Theta(nk).$$

The following algorithm uses just one-dimensional table by storing a new row of binomial coefficients over its predecessor.

Algorithm *Binomial2*(n, k)
//Computes $C(n, k)$ by the dynamic programming algorithm
//with a one-dimensional table
//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**
    if $i \le k$ //in the triangular part
        $T[i] \leftarrow 1$ //the diagonal element
        $u \leftarrow i - 1$ //the rightmost element to be computed
    else $u \leftarrow k$ //in the rectangular part
    //overwrite the preceding row moving right to left
    for $j \leftarrow u$ **downto** 1 **do**
        $T[j] \leftarrow T[j - 1] + T[j]$
return $T[k]$

The space efficiency of *Binomial2* is obviously $\Theta(n)$.

10. After topological sorting of the digraph's vertices, the following formula for the length of the longest path to vertex u is all but obvious:

$$d_u = \max_{(v,u)\in E}\{d_v + w(v, u)\}$$

a. **Algorithm** *DagLongestPath*(G)

```
//Finds the length of a longest path in a dag
//Input: A weighted dag G = ⟨V, E⟩
//Output: The length of its longest path dmax
topologically sort the vertices of G
for every vertex v do
    d_v ← 0 //the length of the longest path to v
for every vertex v taken in topological order do
    for every vertex u such that (v, u) ∈ E do
        d_u ← max{d_v + w(v, u), d_u}
dmax ← 0
for every vertex v do
    dmax ← max{d_v, dmax}
return dmax
```

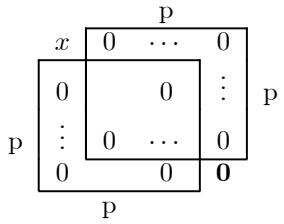
b. The dag in question will have $n + 1$ vertices, placed for convenience in a row mimicking the coin row: vertex 0 for the start and the other n vertices for the coins in the order given. The start vertex will be connected to each of the other vertices; each of the coin-representing vertices will have an outgoing edge to each of the coin-representing vertices after its immediate successor. The weight of an edge will be the value of the coin represented by the vertex the edge points to.

11. We will assume that the rows and columns of a given matrix B are numbered from 1 to m and from 1 to n, respectively. Let $F(i, j)$ be the order of the largest all-zero submatrix of a given matrix B with its low right corner at (i, j). If $b_{ij} = 1$, $F(i, j) = 0$ according to the definition of $F(i, j)$. The nontrivial case is that of $b_{ij} = 0$. In this case, one can prove that

$$\begin{aligned} F(i,j) &= \min\{F(i-1,j),\ F(i,j-1),\ F(i-1,j-1)\} + 1 \quad \text{for } 1 \le i \le m,\ 1 \le j \le n \\ F(0,j) &= 0 \quad \text{for } 0 \le j \le n \quad \text{and} \quad F(i,0) = 0 \quad \text{for } 0 \le i \le m. \end{aligned}$$

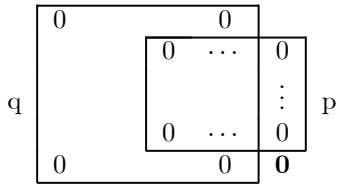
Indeed, if $b_{ij} = 0$ and at least one of $b_{i-1,j}$, $b_{i,j-1}$, and $b_{i-1,j-1}$ is 1, then $F(i, j) = 1$ and $\min\{F(i-1, j), F(i, j-1), F(i-1, j-1)\} + 1 = 1$.

Consider now the case of $b_{ij} = 0$ and $F(i-1, j) = F(i, j-1) = p > 0$, depicted below with b_{ij} shown in bold.



If $x = 0$, $F(i-1, j-1) \ge p$ and $\min\{F(i-1, j), F(i, j-1), F(i-1, j-1)\}+1 = p+1 = F(i, j)$. If $x = 1$, $F(i-1, j-1) = p-1$ and $\min\{F(i-1, j), F(i, j-1), F(i-1, j-1)\} + 1 = (p-1) + 1 = p = F(i, j)$.

Consider the case of $b_{ij} = 0$, $F(i-1, j) = p$ and $F(i, j-1) = q$, where $p \neq q$. Without loss of generality, we assume that $p < q$.



Then $F(i-1, j-1) \geq p$ and $\min\{F(i-1, j),\, F(i, j-1),\, F(i-1, j-1)\} + 1 = p + 1 = F(i, j)$.

Using the above recurrence, one can use it to fill the $m \times n$ table of $F(i, j)$ values row by row top to bottom and each row left to write and then find the largest value in this table. Here is an example.

$B =$

	1	2	3	4	5	6
1	0	0	1	0	0	0
2	1	1	0	0	0	0
3	0	0	1	0	0	0
4	1	0	0	1	0	0

$F =$

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	1	0	1	1	1
2	0	0	0	1	1	2	2
3	0	1	1	0	1	2	3
4	0	0	1	1	0	1	2

12. a. 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. If team A wins the game, which happens with probability p, A will need $i - 1$ more wins to win the series while B will still need j wins. If team A looses the game, which happens with probability $q = 1 - p$, A will still need i wins while B will need $j - 1$ wins to win the series. This leads to the recurrence

$$P(i, j) = pP(i-1, j) + qP(i, j-1) \quad \text{for } i, j > 0.$$

The initial conditions follow immediately from the definition of $P(i, j)$:

$$P(0, j) = 1 \quad \text{for } j > 0, \quad P(i, 0) = 0 \quad \text{for } i > 0.$$

b. Here is the dynamic programming table in question, with its entries rounded-off to two decimal places. (It can be filled either row-by-row, or column-by-column, or diagonal-by-diagonal.)

$i \setminus j$	0	1	2	3	4
0		1	1	1	1
1	0	0.40	0.64	0.78	0.87
2	0	0.16	0.35	0.52	0.66
3	0	0.06	0.18	0.32	0.46
4	0	0.03	0.09	0.18	0.29

Thus, $P[4, 4] \approx 0.29$.

c. **Algorithm** *WorldSeries*(n, p)
//Computes the odds of winning a series of n games
//Input: A number of wins n needed to win the series
// and probability p of one particular team winning a game
//Output: The probability of this team winning the series
$q \leftarrow 1 - p$
for $j \leftarrow 1$ **to** n **do**
  $P[0, j] \leftarrow 1.0$
for $i \leftarrow 1$ **to** n **do**
  $P[i, 0] \leftarrow 0.0$
  for $j \leftarrow 1$ **to** n **do**
    $P[i, j] \leftarrow p * P[i - 1, j] + q * P[i, j - 1]$
return $P[n, n]$

Both the time efficiency and the space efficiency are in $\Theta(n^2)$ because each entry of the $n + 1$-by-$n + 1$ table (except $P[0, 0]$, which is not computed) is computed in $\Theta(1)$ time.

Exercises 8.2

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?

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

   b. Write 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.

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 $O(n)$.

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

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

5. Design a dynamic programming algorithm for the version of the knapsack problem in which there are unlimited quantities of copies for each of the n item kinds given. Indicate the time efficiency of the algorithm.

6. 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.

7. 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).

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

9. ▷ 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

Hints to Exercises 8.2

1. a. Use formulas (8.6)–(8.7) to fill in the appropriate table, as is done for another instance of the problem in the section.

   b.–c. What would the equality of the two terms in

   $$\max\{F(i-1,j),\ v_i + F(i-1,j-w_i)\}$$

   mean?

2. a. Write pseudocode to fill the table in Fig. 8.4 (say, row by row) by using formulas (8.6)–(8.7).

   b. An algorithm for identifying an optimal subset is outlined in the section via an example.

3. How many values does the algorithm compute? How long does it take to compute one value? How many table cells need to be traversed to identify the composition of an optimal subset?

4. Use the definition of $F(i,j)$ to check whether it is always true that

   a. $F(i,j-1) \le F(i,j)$ for $1 \le j \le W$.

   b. $F(i-1,j) \le F(i,j)$ for $1 \le i \le n$.

5. The problem is similar to one of the problems discussed in Section 8.1.

6. Trace the calls of the function *MemoryKnapsack*(i,j) on the instance in question. (An application to another instance can be found in the section.)

7. The algorithm applies formula (8.6) to fill *some* of the table's cells. Why can we still assert that its efficiencies are in $\Theta(nW)$?

8. One of the reasons deals with the time efficiency; the other deals with the space efficiency.

9. n/a

Solutions to Exercises 8.2

1. a.

		capacity j						
	i	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$, $v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2$, $v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1$, $v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4$, $v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5$, $v_5 = 50$	5	0	15	20	35	40	55	65

The maximal value of a feasible subset is $F[5, 6] = 65$. The optimal subset is {item 3, item 5}.

b.-c. The instance has a unique optimal subset in view of the following general property: An instance of the knapsack problem has a unique optimal solution if and only if the algorithm for obtaining an optimal subset, which retraces backward the computation of $F[n, W]$, encounters no equality between $F[i-1, j]$ and $v_i + F[i-1, j-w_i]$ during its operation.

2. a. **Algorithm** *DPKnapsack*$(w[1..n], v[1..n], W)$
//Solves the knapsack problem by dynamic programming (bottom up)
//Input: Arrays $w[1..n]$ and $v[1..n]$ of weights and values of n items,
// knapsack capacity W
//Output: The table $F[0..n, 0..W]$ that contains the value of an optimal
// subset in $F[n, W]$ and from which the items of an optimal
// subset can be found
for $i \leftarrow 0$ **to** n **do** $F[i, 0] \leftarrow 0$
for $j \leftarrow 1$ **to** W **do** $F[0, j] \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** W **do**
    if $j - w[i] \geq 0$
      $F[i, j] \leftarrow \max\{F[i-1, j],\ v[i] + F[i-1, j-w[i]]\}$
    else $F[i, j] \leftarrow F[i-1, j]$
return $F[0..n, 0..W]$

b. **Algorithm** *OptimalKnapsack*$(w[1..n], v[1..n], F[0..n, 0..W])$
//Finds the items composing an optimal solution to the knapsack problem
//Input: Arrays $w[1..n]$ and $v[1..n]$ of weights and values of n items,
// and table $F[0..n, 0..W]$ generated by
// the dynamic programming algorithm
//Output: List $L[1..k]$ of the items composing an optimal solution
$k \leftarrow 0$ //size of the list of items in an optimal solution
$j \leftarrow W$ //unused capacity

for $i \leftarrow n$ **downto** 1 **do**
    if $F[i, j] > F[i-1, j]$
        $k \leftarrow k + 1$; $L[k] \leftarrow i$ //include item i
        $j \leftarrow j - w[i]$
return L

Note: In fact, we can also stop the algorithm as soon as j, the unused capacity of the knapsack, becomes 0.

3. The algorithm fills a table with $n + 1$ rows and $W + 1$ columns, spending $\Theta(1)$ time to fill one cell (either by applying (8.6) or (8.7). Hence, its time efficiency and its space efficiency are in $\Theta(nW)$.

   In order to identify the composition of an optimal subset, the algorithm repeatedly compares values at no more than two cells in a previous row. Hence, its time efficiency class is in $O(n)$.

4. Both assertions are true:

   a. $F(i, j-1) \leq F(i, j)$ for $1 \leq j \leq W$ is true because it simply means that the maximal value of a subset that fits into a knapsack of capacity $j - 1$ cannot exceed the maximal value of a subset that fits into a knapsack of capacity j.

   b. $F(i-1, j) \leq F(i, j)$ for $1 \leq i \leq n$ is true because it simply means that the maximal value of a subset of the first $i - 1$ items that fits into a knapsack of capacity j cannot exceed the maximal value of a subset of the first i items that fits into a knapsack of the same capacity j.

5. Here, the recurrence underlying the dynamic-programming algorithm is

$$\begin{aligned} F(W) &= \max_{j:\, W \geq w_j} \{F(W - w_j)\} + v_j, \\ F(W) &= 0 \text{ if } W < w_j \text{ for all } 1 \leq j \leq n. \end{aligned}$$

   Using this recurrence, the algorithm can fill the one-row table table in the same way such a table is filled for the change-making problem discussed in Section 8.1.

6. In the table below, the cells marked by a minus are the ones for which no entry is computed for the instance in question; the only nontrivial entry that is retrieved without recomputation is $(2, 1)$.

		capacity j						
	i	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$, $v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2$, $v_2 = 20$	2	0	0	20	-	-	45	45
$w_3 = 1$, $v_3 = 15$	3	0	15	20	-	-	-	60
$w_4 = 4$, $v_4 = 40$	4	0	15	-	-	-	-	60
$w_5 = 5$, $v_5 = 50$	5	0	-	-	-	-	-	65

7. Since some of the cells of a table with $n + 1$ rows and $W + 1$ columns are filled in constant time, both the time and space efficiencies are in $O(nW)$. But all the entries of the table need to be initialized (unless virtual initialization of Problem 7 in Exercises 7.1 is used); this puts them in $\Omega(nW)$. Hence, both efficiencies are in $\Theta(nW)$.

8. For the problem of computing a binomial coefficient, we know in advance which cells of the table need to be computed. Therefore unnecessary computations can be avoided by the bottom-up dynamic programming algorithm as well. Also, using the memory function method requires $\Theta(nk)$ space, whereas the bottom-up algorithm needs only $\Theta(n)$ because the next row of the table can be written over its immediate predecessor.

9. n/a

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 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 $n = 1, 2, ..., 5$.

   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. Write 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.

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

$$A_1 \cdot A_2 \cdot \ldots \cdot A_n$$

whose dimensions are $d_0 \times d_1$, $d_1 \times d_2, ..., d_{n-1} \times d_n$, 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.

Hints to Exercises 8.3

1. Continue applying formula (8.8) as prescribed by the algorithm.

2. a. The algorithm's time efficiency can be investigated by following the standard plan of analyzing the time efficiency of a nonrecursive algorithm.

   b. How much space do the two tables generated by the algorithm use?

3. $k = R[1, n]$ indicates that the root of an optimal tree is the kth key in the list of ordered keys $a_1, ..., a_n$. The roots of its left and right subtrees are specified by $R[1, k-1]$ and $R[k+1, n]$, respectively.

4. Use a space-for-time trade-off.

5. If the assertion were true, would we not have a simpler algorithm for constructing an optimal binary search tree?

6. The structure of the tree should simply minimize the average depth of its nodes. Do not forget to indicate a way to distribute the keys among the nodes of the tree.

7. a. Since there is a one-to-one correspondence between binary search trees for a given set of n orderable keys and the total number of binary trees with n nodes (why?), you can count the latter. Consider all the possibilities of partitioning the nodes between the left and right subtrees.

   b. Compute the values in question using the two formulas.

   c. Use the formula for the nth Catalan number and Stirling's formula for $n!$.

8. Change the bounds of the innermost loop of algorithm *OptimalBST* by exploiting the monotonicity of the root table mentioned at the end of the section.

9. Assume that $a_1, ..., a_n$ are distinct keys ordered from the smallest to the largest, $p_1, ..., p_n$ are the probabilities of searching for them, and $q_0, q_1, ..., q_n$ are probabilities of unsuccessful searches for keys in intervals $(-\infty, a_1)$, $(a_1, a_2), ..., (a_n, \infty)$, respectively; $(p_1 + \cdots + p_n) + (q_0 + \cdots + q_n) = 1$. Set up a recurrence relation similar to recurrence (8.8) for the expected number of key comparisons that takes into account both successful and unsuccessful searches.

10. See the memory function solution for the knapsack problem in Section 8.2.

11. a. It is easier to find a general formula for the number of multiplications needed for computing $(A_1 \cdot A_2) \cdot A_3$ and $A_1 \cdot (A_2 \cdot A_3)$ for matrices $A_1$with dimensions $d_0 \times d_1$, A_2 with dimensions $d_1 \times d_2$, and A_3 with dimensions

$d_2 \times d_3$ and then choose some specific values for the dimensions to get a required example.

b. You can get the answer by following the approach used for counting binary trees.

c. The recurrence relation for the optimal number of multiplications in computing $A_i \cdot \ldots \cdot A_j$ is very similar to the recurrence relation for the optimal number of comparisons in searching in a binary tree composed of keys $a_i, ..., a_j$.

Solutions to Exercises 8.3

1. The instance of the problem in question is defined by the data

key	A	B	C	D
probability	0.1	0.2	0.4	0.3

The table entries for the dynamic programming algorithm are computed as follows:

$$C[1,2] = \min \begin{array}{ll} k=1: & C[1,0]+C[2,2]+\sum_{s=1}^{2} p_s = 0+0.2+0.3 = 0.5 \\ k=2: & C[1,1]+C[3,2]+\sum_{s=1}^{2} p_s = 0.1+0+0.3 = \mathbf{0.4} \end{array} = 0.4$$

$$C[2,3] = \min \begin{array}{ll} k=2: & C[2,1]+C[3,3]+\sum_{s=2}^{3} p_s = 0+0.4+0.6 = 1.0 \\ k=3: & C[2,2]+C[4,3]+\sum_{s=2}^{3} p_s = 0.2+0+0.6 = \mathbf{0.8} \end{array} = 0.8$$

$$C[3,4] = \min \begin{array}{ll} k=3: & C[3,2]+C[4,4]+\sum_{s=3}^{4} p_s = 0+0.3+0.7 = \mathbf{1.0} \\ k=4: & C[3,3]+C[5,4]+\sum_{s=3}^{4} p_s = 0.4+0+0.7 = 1.1 \end{array} = 1.0$$

$$C[1,3] = \min \begin{array}{ll} k=1: & C[1,0]+C[2,3]+\sum_{s=1}^{3} p_s = 0+0.8+0.7 = 1.5 \\ k=2: & C[1,1]+C[3,3]+\sum_{s=1}^{3} p_s = 0.1+0.4+0.7 = 1.2 \\ k=3: & C[1,2]+C[4,3]+\sum_{s=1}^{3} p_s = 0.4+0+0.7 = \mathbf{1.1} \end{array} = 1.1$$

$$C[2,4] = \min \begin{array}{ll} k=2: & C[2,1]+C[3,4]+\sum_{s=2}^{4} p_s = 0+1.0+0.9 = 1.9 \\ k=3: & C[2,2]+C[4,4]+\sum_{s=2}^{4} p_s = 0.2+0.3+0.9 = \mathbf{1.4} \\ k=4: & C[2,3]+C[5,4]+\sum_{s=2}^{4} p_s = 0.8+0+0.9 = 1.7 \end{array} = 1.1$$

$$C[1,4] = \min \begin{array}{ll} k=1: & C[1,0]+C[2,4]+\sum_{s=1}^{s=4} p_s = 0+1.4+1.0 = 2.4 \\ k=2: & C[1,1]+C[3,4]+\sum_{s=1}^{4} p_s = 0.1+1.0+1.0 = 2.1 \\ k=3: & C[1,2]+C[4,4]+\sum_{s=1}^{4} p_s = 0.4+0.3+1.0 = \mathbf{1.7} \\ k=4: & C[1,3]+C[5,4]+\sum_{s=1}^{4} p_s = 1.1+0+1.0 = 2.1 \end{array} = 1.7$$

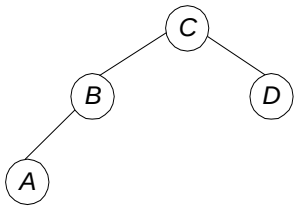
the main table

	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

the root table

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

the optimal tree



2. a. The number of times the innermost loop is executed is given by the sum

$$\begin{aligned}
\sum_{d=1}^{n-1}\sum_{i=1}^{n-d}\sum_{k=i}^{i+d}1 &= \sum_{d=1}^{n-1}\sum_{i=1}^{n-d}(i+d-i+1) = \sum_{d=1}^{n-1}\sum_{i=1}^{n-d}(d+1) \\
&= \sum_{d=1}^{n-1}(d+1)(n-d) = \sum_{d=1}^{n-1}(dn+n-d^2-d) \\
&= \sum_{d=1}^{n-1}nd + \sum_{d=1}^{n-1}n - \sum_{d=1}^{n-1}d^2 - \sum_{d=1}^{n-1}d \\
&= n\frac{(n-1)n}{2} + n(n-1) - \frac{(n-1)n(2n-1)}{6} - \frac{(n-1)n}{2} \\
&= \frac{1}{2}n^3 - \frac{2}{6}n^3 + O(n^2) \in \Theta(n^3).
\end{aligned}$$

b. The algorithm generates the $(n+1)$-by-$(n+1)$ table C and the n-by-n table R and fills about one half of each. Hence, the algorithm's space efficiency is in $\Theta(n^2)$.

3. Call *OptimalTree*$(1, n)$ below:

Algorithm *OptimalTree*(i, j)
//Input: Indices i and j of the first and last keys of a sorted list of keys
//composing the tree and table $R[1..n, 1..n]$ obtained by dynamic
//programming
//Output: Indices of nodes of an optimal binary search tree in preorder
if $i \le j$
    $k \leftarrow R[i, j]$
    print(k)
    OptimalTree$(i, k-1)$
    OptimalTree$(k+1, j)$

4. Precompute $S_k = \sum_{s=1}^{k} p_s$ for $k = 1, 2, ..., n$ and set $S_0 = 0$. Then $\sum_{s=i}^{j} p_s$ can be found as $S_j - S_{i-1}$ for any $1 \le i \le j \le n$.

5. False. Here is a simple counterexample: $A(0.3)$, $B(0.3)$, $C(0.4)$. (The numbers in the parentheses indicate the search probabilities.) The average number of comparisons in a binary search tree with C in its root is 0.3·2 + 0.3·3 + 0.4·1 = 1.9, while the average number of comparisons in the binary search tree with B in its root is 0.3·1 + 0.3·2 + 0.4·2 = 1.7.

6. The binary search tree in question should have a maximal number of nodes on each of its levels except the last one. (For simplicity, we can put all the nodes of the last level in their leftmost positions possible to make it complete.) The keys of a given sorted list can be distributed among the nodes of the binary tree by performing its in-order traversal.

   Let p/n be the probability of searching for each key, where $0 \le p \le 1$. The complete binary tree with 2^k nodes will have 2^i nodes on level i for $i = 0, ..., k-1$ and one node on level k. Hence, the average number of comparisons in a successful search will be given by the following formula:

$$\begin{aligned} C(2^k) &= \sum_{i=0}^{k-1} (p/2^k)(i+1)2^i + (p/2^k)(k+1) \\ &= (p/2^k)\frac{1}{2}\sum_{i=0}^{k-1}(i+1)2^{i+1} + (p/2^k)(k+1) \\ &= (p/2^k)\frac{1}{2}\sum_{j=1}^{k} j2^j + (p/2^k)(k+1) \\ &= (p/2^k)\frac{1}{2}[(k-1)2^{k+1}+2] + (p/2^k)(k+1) \\ &= (p/2^k)[(k-1)2^k + k + 2]. \end{aligned}$$

7. a. Let $b(n)$ be the number of distinct binary trees with n nodes. If the left subtree of a binary tree with n nodes has k nodes ($0 \le k \le n-1$), the right subtree must have $n-1-k$ nodes. The number of such trees is therefore $b(k)b(n-1-k)$. Hence,

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

   b. Substituting the first five values of n into the formula above and into the formula for the nth Catalan number yields the following values:

n	0	1	2	3	4	5
$b(n) = c(n)$	1	1	2	5	14	42

c.

$$\begin{aligned} b(n) &= c(n) = \binom{2n}{n}\frac{1}{n+1} = \frac{(2n)!}{(n!)^2}\frac{1}{n+1} \\ &\approx \frac{\sqrt{2\pi 2n}(2n/e)^{2n}}{[\sqrt{2\pi n}(n/e)^n]^2}\frac{1}{n+1} = \frac{\sqrt{4\pi n}(2n/e)^{2n}}{2\pi n(n/e)^{2n}}\frac{1}{n+1} \\ &\approx \frac{1}{\sqrt{\pi n}}\left(\frac{2n/e}{n/e}\right)^{2n}\frac{1}{n+1} = \frac{1}{\sqrt{\pi n}}2^{2n}\frac{1}{n+1} \in \Theta(4^n n^{-3/2}). \end{aligned}$$

This implies that finding an optimal binary search tree by exhaustive search is feasible only for very small values of n and is, in general, vastly inferior to the dynamic programming algorithm.

8. The dynamic programming algorithm finds the root a_{kmin} of an optimal binary search tree for keys $a_i, ..., a_j$ by minimizing $\{C[i,k-1]+C[k+1,j]\}$ for $i \le k \le j$. As pointed out at the end of Section 8.3 (see also [KnuIII], p. 456, Exercise 27), $R[i,j-1] \le kmin \le R[i+1,j]$. This observation allows us to change the bounds of the algorithm's innermost loop to the lower bound of $R[i,j-1]$ and the upper bound of $R[i+1,j]$, respectively. The number of times the innermost loop of the modified algorithm will be executed can be estimated as suggested by Knuth (see [KnuIII], p.439):

$$\begin{aligned} \sum_{d=1}^{n-1}\sum_{i=1}^{n-d}\sum_{k=R[i,j-1]}^{R[i+1,j]} 1 &= \sum_{d=1}^{n-1}\sum_{i=1}^{n-d}\sum_{k=R[i,i+d-1]}^{R[i+1,i+d]} 1 \\ &= \sum_{d=1}^{n-1}\sum_{i=1}^{n-d}(R[i+1,i+d]-R[i,i+d-1]+1) \\ &= \sum_{d=1}^{n-1}(\sum_{i=1}^{n-d}(R[i+1,i+d]-R[i,i+d-1])+\sum_{i=1}^{n-d}1). \end{aligned}$$

By "telescoping" the first sum, we can see that all its terms except the two get cancelled to yield

$\sum_{i=1}^{n-d}(R[i+1,i+d]-R[i,i+d-1]) =$
$= (R[2,1+d]-R[1,1+d-1])$
$+(R[3,2+d]-R[2,2+d-1])$
$+ \ldots$
$+(R[n-d+1,n-d+d]-R[n-d,n-d+d-1])$
$= R[n-d+1,n]-R[1,d].$

Since the second sum $\sum_{i=1}^{n-d}1$ is equal to $n-d$, we obtain

$$\sum_{i=1}^{n-d}(R[i+1,i+d]-R[i,i+d-1])+\sum_{i=1}^{n-d}1 = R[n-d+1,n]-R[1,d]+n-d < 2n.$$

Hence,

$$\sum_{d=1}^{n-1}\sum_{i=1}^{n-d}\sum_{k=R[i,j-1]}^{R[i+1,j]} 1 = \sum_{d=1}^{n-1}(R[n-d+1,n]-R[1,d]+n-d) < \sum_{d=1}^{n-1} 2n < 2n^2.$$

On the other hand, since $R[i+1,i+d]-R[i,i+d-1] \geq 0$,

$$\begin{aligned}\sum_{d=1}^{n-1}\sum_{i=1}^{n-d}\sum_{k=R[i,j-1]}^{R[i+1,j]} 1 &= \sum_{d=1}^{n-1}\sum_{i=1}^{n-d}(R[i+1,i+d]-R[i,i+d-1]+1)\\ &\geq \sum_{d=1}^{n-1}\sum_{i=1}^{n-d} 1 = \sum_{d=1}^{n-1}(n-d) = \frac{(n-1)n}{2} \geq \frac{1}{4}n^2 \text{ for } n \geq 2.\end{aligned}$$

Therefore, the time efficiency of the modified algorithm is in $\Theta(n^2)$.

9. Let $a_1, ...a_n$ be a sorted list of n distinct keys, p_i be a known probability of searching for key a_i for $i = 1, 2, ..., n$, and q_i be a known probability of searching (unsuccessfully) for a key between a_i and a_{i+1} for $i = 0, 1, ..., n$ (with q_0 being a probability of searching for a key smaller than a_1 and q_n being a probability of searching for a key greater than a_n). It's convenient to associate unsuccessful searches with external nodes of a binary search tree (see Section 5.3). Repeating the derivation of equation (8.11) for such a tree yields the following recurrence for the expected number of key comparisons

$$C[i,j] = \min_{i \leq k \leq j}\{C[i,k-1]+C[k+1,j]\}+\sum_{s=i}^{j} p_s + \sum_{s=i-1}^{j} q_s \quad \text{for } 1 \leq i < j \leq n$$

with the initial condition

$$C[i,i] = p_i + q_{i-1} + q_i \quad \text{for } i = 1, ..., n.$$

In all other respects, the algorithm remains the same.

10. **Algorithm** *MFOptimalBST*(i, j)
//Returns the number of comparisons in a successful search in a *BST*
//Input: Indices i, j indicating the range of keys in a sorted list of n keys
//and an array $P[1..n]$ of search probabilities used as a global variable
//Uses a global table $C[1..n+1, 0..n]$ initialized with a negative number
//Output: The average number of comparisons in the optimal BST
if $j = i - 1$ **return** 0
if $j = i$ **return** $P[i]$
if $C[i,j] < 0$
    $minval \leftarrow \infty$

for $k \leftarrow i$ **to** j **do**
$minval \leftarrow \min\{MFOptimalBST(i, k-1) + MFOptimalBST(k+1, j), minval\}$
$sum \leftarrow 0$; **for** $s \leftarrow i$ **to** j **do** $sum \leftarrow sum + P[s]$
$C[i, j] \leftarrow minval + sum$
return $C[i, j]$

Note: The first two lines of this pseudocode can be eliminated by an appropriate initialization of table C.

11. a. Multiplying two matrices of dimensions $\alpha \times \beta$ and $\beta \times \gamma$ by the definition-based algorithm requires $\alpha\beta\gamma$ multiplications. (There are $\alpha\gamma$ elements in the product, each requiring β multiplications to be computed.) If the dimensions of A_1, A_2, and A_3 are $d_0 \times d_1$, $d_1 \times d_2$, and $d_2 \times d_3$, respectively, then $(A_1 \cdot A_2) \cdot A_3$ will require

$$d_0 d_1 d_2 + d_0 d_2 d_3 = d_0 d_2 (d_1 + d_3)$$

multiplications, while $A_1 \cdot (A_2 \cdot A_3)$ will need

$$d_1 d_2 d_3 + d_0 d_1 d_3 = d_1 d_3 (d_0 + d_2)$$

multiplications. Here is a simple choice of specific values to make, say, the first of them be 1000 times larger than the second:

$$d_0 = d_2 = 10^3, \quad d_1 = d_3 = 1.$$

b. Let $m(n)$ be the number of different ways to compute a chain product of n matrices $A_1 \cdot ... \cdot A_n$. Any parenthesization of the chain will lead to multiplying, as the last operation, some product of the first k matrices $(A_1 \cdot ... \cdot A_k)$ and the last $n-k$ matrices $(A_{k+1} \cdot ... \cdot A_n)$. There are $m(k)$ ways to do the former, and there are $m(n-k)$ ways to do the latter. Hence, we have the following recurrence for the total number of ways to parenthesize the matrix chain of n matrices:

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

Since parenthesizing a chain of n matrices for multiplication is very similar to constructing a binary tree of n nodes, it should come as no surprise that the above recurrence is very similar to the recurrence

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

for the number of binary trees mentioned in Section 8.3. Nor is it surprising that their solutions are very similar, too: namely,

$$m(n) = b(n-1) \quad \text{for } n \geq 1,$$

where $b(n)$ is the number of binary trees with n nodes. Let us prove this assertion by mathematical induction. The basis checks immediately: $m(1) = b(0) = 1$. For the general case, let us assume that $m(k) = b(k-1)$ for all positive integers not exceeding some positive integer n (we're using the strong version of mathematical induction); we'll show that the equality holds for $n+1$ as well. Indeed,

$$\begin{aligned}
m(n+1) &= \sum_{k=1}^{n} m(k)m(n+1-k) \\
&= [\text{using the induction's assumption}] \sum_{k=1}^{n} b(k-1)b(n-k) \\
&= [\text{substituting } l = k-1] \qquad \sum_{l=0}^{n-1} b(l)b(n-1-l) \\
&= [\text{see the recurrence for } b(n)] \qquad b(n).
\end{aligned}$$

c. Let $M[i,j]$ be the optimal (smallest) number of multiplications needed for computing $A_i \cdot \ldots \cdot A_j$. If k is an index of the last matrix in the first factor of the last matrix product, then

$$\begin{aligned}
M[i,j] &= \max_{1 \leq k \leq j-1} \{M[i,k] + M[k+1,j] + d_{i-1}d_k d_j\} \quad \text{for } 1 \leq i < j \leq n, \\
M[i,i] &= 0.
\end{aligned}$$

This recurrence, which is quite similar to the one for the optimal binary search tree problem, suggests filling the $n+1$-by-$n+1$ table diagonal by diagonal as in the following algorithm:

Algorithm *MatrixChainMultiplication*$(D[0..n])$
//Solves matrix chain multiplication problem by dynamic programming
//Input: An array $D[0..n]$ of dimensions of n matrices
//Output: The minimum number of multiplications needed to multiply
//a chain of n matrices of the given dimensions and table $T[1..n, 1..n]$
//for obtaining an optimal order of the multiplications
for $i \leftarrow 1$ **to** n **do** $M[i,i] \leftarrow 0$
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** $j-1$ **do**

$temp \leftarrow M[i,k] + M[k+1,j] + D[i-1] * D[k] * D[j]$
if $temp < minval$
$minval \leftarrow temp$
$kmin \leftarrow k$
$T[i,j] \leftarrow kmin$
return $M[1,n]$, T

To find an optimal order to multiply the matrix chain, call *OptimalMultiplicationOrder*$(1,n)$ below:
Algorithm *OptimalOrder*(i,j)
//Outputs an optimal order to multiply n matrices
//Input: Indices i and j of the first and last matrices in $A_i...A_j$ and
//table $T[1..n,1..n]$ generated by *MatrixChainMultiplication*
//Output: $A_i...A_j$ parenthesized for optimal multiplication
if $i = j$
print("A_i")
else
$k \leftarrow T[i,j]$
print("(")
OptimalOrder(i,k)
OptimalOrder$(k+1,j)$
print(")")

Exercises 8.4

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 following 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 if 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 two straws can be connected indirectly via other connected straws. [1994 East-Central Regionals of the ACM International Collegiate Programming Contest]

Hints to Exercises 8.4

1. Apply the algorithm to the adjacency matrix given as it is done in the section for another matrix.

2. a. The answer can be obtained either by considering how many values the algorithm computes or by following the standard plan for analyzing the efficiency of a nonrecursive algorithm (i.e., by setting up a sum to count its basic operation).

   b. What is the efficiency class of the traversal-based algorithm for sparse graphs represented by their adjacency lists?

3. Show that we can simply overwrite elements of $R^{(k-1)}$ with elements of $R^{(k)}$ without any other changes in the algorithm.

4. What happens if $R^{(k-1)}[i, k] = 0$?

5. Show first that formula (8.11) (from which the superscripts can be eliminated according to the solution to Problem 3)

   $$r_{ij} = r_{ij} \quad \textbf{or} \quad r_{ik} \textbf{ and } r_{kj}.$$

   is equivalent to

   $$\textbf{if } r_{ik} \quad r_{ij} \leftarrow r_{ij} \textbf{ or } r_{kj}.$$

6. a. What property of the transitive closure indicates a presence of a directed cycle? Is there a better algorithm for checking this?

   b. Which elements of the transitive closure of an undirected graph are equal to 1? Can you find such elements with a faster algorithm?

7. See an example of applying the algorithm to another instance in the section.

8. What elements of matrix $D^{(k-1)}$ does $d_{ij}^{(k)}$, the element in the ith row and the jth column of matrix $D^{(k)}$, depend on? Can these values be changed by the overwriting?

9. Your counterexample must contain a cycle of a negative length.

10. It will suffice to store, in a single matrix P, indices of intermediate vertices k used in updates of the distance matrices. This matrix can be initialized with all zero elements.

11. The problem can be solved by utilizing two well-known algorithms: one from computational geometry, the other dealing with graphs.

Solutions to Exercises 8.4

1. Applying Warshall's algorithm yields the following sequence of matrices (in which newly updated elements are shown in bold):

$$R^{(0)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad R^{(1)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$R^{(2)} = \begin{bmatrix} 0 & 1 & \mathbf{1} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad R^{(3)} = \begin{bmatrix} 0 & 1 & 1 & \mathbf{1} \\ 0 & 0 & 1 & \mathbf{1} \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$R^{(4)} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = T$$

2. a. For a graph with n vertices, the algorithm computes n matrices $R^{(k)}$ $(k = 1, 2, ..., n)$, each of which has n^2 elements. Hence, the total number of elements to be computed is n^3. Since computing each element takes constant time, the time efficiency of the algorithm is in $\Theta(n^3)$.

   b. Since one *DFS* or *BFS* traversal of a graph with n vertices and m edges, which is represented by its adjacency lists, takes $\Theta(n + m)$ time, doing this n times takes $n\Theta(n+m) = \Theta(n^2+nm)$ time. For sparse graphs (i.e., if $m \in O(n)$), $\Theta(n^2 + nm) = \Theta(n^2)$, which is more efficient than the $\Theta(n^3)$ time efficiency of Warshall's algorithm.

3. The algorithm computes the new value to be put in the ith row and the jth column by the formula

$$R^{(k)}[i, j] = R^{(k-1)}[i, j] \textbf{ or } (R^{(k-1)}[i, k] \textbf{ and } R^{(k-1)}[k, j]).$$

   The formula implies that all the elements in the kth row and all the elements in the kth column never change on the kth iteration, i.e., while computing elements of $R^{(k)}$ from elements of $R^{(k-1)}$. (Substitute k for i and k for j, respectively, into the formula to verify these assertions.) Hence, the new value of the element in the ith row and the jth column, $R^{(k)}[i, j]$, can be written over its old value $R^{(k-1)}[i, j]$ for every i, j.

4. If $R^{(k-1)}[i,k] = 0$,

$$R^{(k)}[i,j] = R^{(k-1)}[i,j] \textbf{ or } (R^{(k-1)}[i,k] \textbf{ and } R^{(k-1)}[k,j]) = R^{(k-1)}[i,j],$$

and hence the innermost loop need not be executed. And, since $R^{(k-1)}[i,k]$ doesn't depend on j, its comparison with 0 can be done outside the j loop. This leads to the following implementation of Warshall's algorithm:

Algorithm *Warshall2*$(A[1..n, 1..n])$
//Implements Warshall's algorithm with a more efficient innermost loop
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph in place of A
for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
        if $A[i,k]$
            for $j \leftarrow 1$ **to** n **do**
                if $A[k,j]$
                    $A[i,j] \leftarrow 1$
return A

5. First, it is easy to check that

$$r_{ij} \leftarrow r_{ij} \quad \textbf{or} \quad (r_{ik} \textbf{ and } r_{kj}).$$

is equivalent to

$$\textbf{if } r_{ik} \quad r_{ij} \leftarrow r_{ij} \textbf{ or } r_{kj}$$

Indeed, if $r_{ik} = 1$ (i.e., **true**),

$$r_{ij} \textbf{ or } (r_{ik} \textbf{ and } r_{kj}) = r_{ij} \textbf{ or } r_{kj},$$

which is exactly the value assigned to r_{ij} by the **if** statement as well. If $r_{ik} = 0$ (i.e., **false**),

$$r_{ij} \textbf{ or } (r_{ik} \textbf{ and } r_{kj}) = r_{ij},$$

i.e., the new value of r_{ij} will be the same as its previous value—exactly, the result we obtain by executing the **if** statement.

Here is the algorithm that exploits this observation and the bitwise *or* operation applied to matrix rows:

Algorithm *Warshall3*$(A[1..n, 1..n])$
//Implements Warshall's algorithm for computing the transitive closure
//with the bitwise *or* operation on matrix rows
//Input: The adjacency matrix A of a digraph
//Output: The transitive closure of the digraph in place of A
for $k \leftarrow 1$ **to** n **do**

```
    for i ← 1 to n do
        if A[i, k]
            row[i] ← row[i] bitwiseor row[k]  //rows of matrix A
return A
```

6. a. With the book's definition of the transitive closure (which considers only nontrivial paths of a digraph), a digraph has a directed cycle if and only if its transitive closure has a 1 on its main diagonal. The algorithm that finds the transitive closure by applying Warshall's algorithm and then checks the elements of its main diagonal is cubic. This is inferior to the quadratic algorithms for checking whether a digraph represented by its adjacency matrix is a dag, which were discussed in Section 4.2.

   b. No. If T is the transitive closure of an undirected graph, $T[i, j] = 1$ if and only if there is a nontrivial path from the ith vertex to the jth vertex. If $i \neq j$, this is the case if and only if the ith vertex and the jth vertex belong to the same connected component of the graph. Thus, one can find the elements outside the main diagonal of the transitive closure that are equal to 1 by a depth-first search or a breadth-first search traversal, which is faster than applying Warshall's algorithm. If $i = j$, $T[i, i] = 1$ if and only if the ith vertex is not isolated, i.e., if it has an edge to at least one other vertex of the graph. Isolated vertices, if any, can be easily identified by the graph's traversal as one-node connected components of the graph.

7. Applying Floyd's algorithm to the given weight matrix generates the following sequence of matrices:

$$D^{(0)} = \begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix} \qquad D^{(1)} = \begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \mathbf{14} \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \mathbf{5} & \infty & \mathbf{4} & 0 \end{bmatrix}$$

$$D^{(2)} = \begin{bmatrix} 0 & 2 & \mathbf{5} & 1 & 8 \\ 6 & 0 & 3 & 2 & 14 \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & \mathbf{8} & 4 & 0 \end{bmatrix} \qquad D^{(3)} = \begin{bmatrix} 0 & 2 & 5 & 1 & 8 \\ 6 & 0 & 3 & 2 & 14 \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & 8 & 4 & 0 \end{bmatrix}$$

$$D^{(4)} = \begin{bmatrix} 0 & 2 & \mathbf{3} & 1 & \mathbf{4} \\ 6 & 0 & 3 & 2 & \mathbf{5} \\ \infty & \infty & 0 & 4 & \mathbf{7} \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & \mathbf{6} & 4 & 0 \end{bmatrix} \qquad D^{(5)} = \begin{bmatrix} 0 & 2 & 3 & 1 & 4 \\ 6 & 0 & 3 & 2 & 5 \\ \mathbf{10} & \mathbf{12} & 0 & 4 & 7 \\ \mathbf{6} & \mathbf{8} & 2 & 0 & 3 \\ 3 & 5 & 6 & 4 & 0 \end{bmatrix} = D$$

8. The formula
$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$$
implies that the value of the element in the ith row and the jth column of matrix $D^{(k)}$ is computed from its own value and the values of the elements in the ith row and the kth column and in the kth row and the jth column in the preceding matrix $D^{(k-1)}$. The latter two cannot change their values when the elements of $D^{(k)}$ are computed because of the formulas

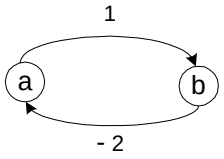
$$d_{ik}^{(k)} = \min\{d_{ik}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kk}^{(k-1)}\} = \min\{d_{ik}^{(k-1)},\ d_{ik}^{(k-1)} + 0\} = d_{ik}^{(k-1)}$$

and

$$d_{kj}^{(k)} = \min\{d_{kj}^{(k-1)},\ d_{kk}^{(k-1)} + d_{kj}^{(k-1)}\} = \min\{d_{kj}^{(k-1)},\ 0 + d_{kj}^{(k-1)}\} = d_{kj}^{(k-1)}.$$

(Note that we took advantage of the fact that the elements d_{kk} on the main diagonal remain 0's. This can be guaranteed only if the graph doesn't contain a cycle of a negative length.)

9. As a simple counterexample, one can suggest the following digraph:



Floyd's algorithm will yield:

$$D^{(0)} = \begin{bmatrix} 0 & 1 \\ -2 & 0 \end{bmatrix} \qquad D^{(1)} = \begin{bmatrix} 0 & 1 \\ -2 & -1 \end{bmatrix} \qquad D^{(2)} = \begin{bmatrix} -1 & 0 \\ -3 & -2 \end{bmatrix}$$

None of the four elements of the last matrix gives the correct value of the shortest path, which is, in fact, $-\infty$ because repeating the cycle enough times makes the length of a path arbitrarily small.

Note: Floyd's algorithm can be used for detecting negative-length cycles, but the algorithm should be stopped as soon as it generates a matrix with a negative element on its main diagonal.

10. As pointed out in the hint to this problem, Floyd's algorithm should be enhanced by recording in an n-by-n matrix index k of an intermediate vertex causing an update of the distance matrix. This is implemented in the pseudocode below:

Algorithm *FloydEnhanced*$(W[1..n, 1..n])$

```
//Input: The weight matrix W of a graph or a digraph
//Output: The distance matrix D[1..n, 1..n] and
//          the matrix of intermediate updates P[1..n, 1..n]
D ← W
for i ← 1 to n do
    for j ← 1 to n do
        P[i, j] ← 0 //initial mark
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            if D[i, k] + D[k, j] < D[i, j]
               D[i, j] ← D[i, k] + D[k, j]}
               P[i, j] ← k
```

For example, for the digraph in Fig. 8.7 whose vertices are numbered from 1 to 4, matrix P will be as follows:

$$P = \begin{bmatrix} 0 & 3 & 0 & 3 \\ 0 & 0 & 1 & 3 \\ 4 & 0 & 0 & 0 \\ 0 & 3 & 1 & 0 \end{bmatrix}.$$

The list of intermediate vertices on the shortest path from vertex i to vertex j can be then generated by the call to the following recursive algorithm, provided $D[i, j] < \infty$:

```
Algorithm ShortestPath(i, j, P[1..n, 1..n])
//The algorithm prints out the list of intermediate vertices of
//a shortest path from the ith vertex to the jth vertex
//Input: Endpoints i and j of the shortest path desired;
//         matrix P of updates generated by Floyd's algorithm
//Output: The list of intermediate vertices of the shortest path
//          from the ith vertex to the jth vertex
k ← P[i, j]
if k ≠ 0
    ShortestPath(i, k)
    print(k)
    ShortestPath(k, j)
```

11. First, for each pair of the straws, determine whether the straws intersect. (Although this can be done in $n \log n$ time by a sophisticated algorithm, the quadratic brute-force algorithm would do because of the quadratic efficiency of the subsequent step; both geometric algorithms can be found, e.g., in R. Sedgewick's "Algorithms," Addison-Wesley, 1988.) Record the obtained information in a boolean $n \times n$ matrix, which must be symmetric. Then find the transitive closure of this matrix in n^2 time by DFS or BFS (see the solution to Problem 6b in these exercises).