```
1     // Dynamic programming algorithm to solve change-making problem.
2     // As a result, the coinsUsed array is filled with the
3     // minimum number of coins needed for change from 0 -> maxChange
4     // and lastCoin contains one of the coins needed to make the change.
5     public static void makeChange( int [ ] coins, int differentCoins,
6                                    int maxChange, int [ ] coinsUsed, int [ ] lastCoin )
7     {
8         coinsUsed[ 0 ] = 0; lastCoin[ 0 ] = 1;
9
10        for( int cents = 1; cents <= maxChange; cents++ )
11        {
12            int minCoins = cents;
13            int newCoin  = 1;
14
15            for( int j = 0; j < differentCoins; j++ )
16            {
17                if( coins[ j ] > cents )   // Cannot use coin j
18                    continue;
19                if( coinsUsed[ cents - coins[ j ] ] + 1 < minCoins )
20                {
21                    minCoins = coinsUsed[ cents - coins[ j ] ] + 1;
22                    newCoin  = coins[ j ];
23                }
24            }
25
26            coinsUsed[ cents ] = minCoins;
27            lastCoin[ cents ]  = newCoin;
28        }
29    }
```

figure 7.25

A dynamic programming algorithm for solving the change-making problem by computing optimal change for all amounts from 0 to `maxChange` and maintaining information to construct the actual coin sequence

7.7 backtracking

A *backtracking algorithm* uses recursion to try all the possibilities.

In this section we set out the last application of recursion. We show how to write a routine to have the computer select an optimal move in the game Tic-Tac-Toe. The class `Best`, shown in Figure 7.26, is used to store the optimal move that is returned by the move selection algorithm. The skeleton for a `TicTacToe` class is shown in Figure 7.27. The class has a data object `board` that represents the current game position.[6] A host of trivial methods are specified,

6. Tic-Tac-Toe is played on a three-by-three board. Two players alternate placing their symbols on squares. The first to get three squares in a row, column, or a long diagonal wins.

figure 7.26
Class to store an evaluated move

```
1  final class Best
2  {
3      int row;
4      int column;
5      int val;
6
7      public Best( int v )
8        { this( v, 0, 0 ); }
9
10     public Best( int v, int r, int c )
11       { val = v; row = r; column = c; }
12 }
```

including routines to clear the board, to test whether a square is occupied, to place something on a square, and to test whether a win has been achieved. The implementation details are provided in the online code.

The challenge is to decide, for any position, what the best move is. The routine used is `chooseMove`. The general strategy involves the use of a backtracking algorithm. A *backtracking algorithm* uses recursion to try all the possibilities.

The *minimax strategy* is used for Tic-Tac-Toe. It is based on the assumption of optimal play by both sides.

The basis for making this decision is `positionValue`, which is shown in Figure 7.28. The method `positionValue` returns `HUMAN_WIN`, `DRAW`, `COMPUTER_WIN`, or `UNCLEAR`, depending on what the board represents.

The strategy used is the *minimax strategy,* which is based on the assumption of optimal play by both players. The value of a position is a `COMPUTER_WIN` if optimal play implies that the computer can force a win. If the computer can force a draw but not a win, the value is `DRAW`; if the human player can force a win, the value is `HUMAN_WIN`. We want the computer to win, so we have `HUMAN_WIN < DRAW < COMPUTER_WIN`.

For the computer, the value of the position is the maximum of all the values of the positions that can result from making a move. Suppose that one move leads to a winning position, two moves lead to a drawing position, and six moves lead to a losing position. Then the starting position is a winning position because the computer can force the win. Moreover, the move that leads to the winning position is the move to make. For the human player we use the minimum instead of the maximum.

This approach suggests a recursive algorithm to determine the value of a position. Keeping track of the best move is a matter of bookkeeping once the basic algorithm to find the value of the position has been written. If the position is a terminal position (i.e., we can see right away that Tic-Tac-Toe has been achieved or the board is full without Tic-Tac-Toe), the position's value is immediate. Otherwise, we recursively try all moves, computing the value of

```
1  class TicTacToe
2  {
3      public static final int HUMAN        = 0;
4      public static final int COMPUTER     = 1;
5      public static final int EMPTY        = 2;
6
7      public static final int HUMAN_WIN    = 0;
8      public static final int DRAW         = 1;
9      public static final int UNCLEAR      = 2;
10     public static final int COMPUTER_WIN = 3;
11
12         // Constructor
13     public TicTacToe( )
14       { clearBoard( ); }
15
16         // Find optimal move
17     public Best chooseMove( int side )
18       { /* Implementation in Figure 7.29 */  }
19
20         // Compute static value of current position (win, draw, etc.)
21     private int positionValue( )
22       {  /* Implementation in Figure 7.28 */ }
23
24         // Play move, including checking legality
25     public boolean playMove( int side, int row, int column )
26       { /* Implementation in online code */ }
27
28         // Make board empty
29     public void clearBoard( )
30       { /* Implementation in online code */ }
31
32         // Return true if board is full
33     public boolean boardIsFull( )
34       { /* Implementation in online code */ }
35
36         // Return true if board shows a win
37     public boolean isAWin( int side )
38       { /* Implementation in online code */ }
39
40         // Play a move, possibly clearing a square
41     private void place( int row, int column, int piece )
42       { board[ row ][ column ] = piece; }
43
44         // Test if a square is empty
45     private boolean squareIsEmpty( int row, int column )
46       { return board[ row ][ column ] == EMPTY; }
47
48     private int [ ] [ ] board = new int[ 3 ][ 3 ];
49 }
```

figure 7.27

Skeleton for class TicTacToe

figure 7.28
Supporting routine for evaluating positions

```
1     // Compute static value of current position (win, draw, etc.)
2     private int positionValue( )
3     {
4         return isAWin( COMPUTER ) ? COMPUTER_WIN :
5                isAWin( HUMAN )    ? HUMAN_WIN :
6                boardIsFull( )     ? DRAW          : UNCLEAR;
7     }
```

each resulting position, and choose the maximum value. The recursive call then requires that the human player evaluate the value of the position. For the human player the value is the minimum of all the possible next moves because the human player is trying to force the computer to lose. Thus the recursive method `chooseMove`, shown in Figure 7.29 takes a parameter `side`, which indicates whose turn it is to move.

Lines 12 and 13 handle the base case of the recursion. If we have an immediate answer, we can return. Otherwise, we set some values at lines 15 to 22, depending on which side is moving. The code in lines 28 to 38 is executed once for each available move. We try the move at line 28, recursively evaluate the move at line 29 (saving the value), and then undo the move at line 30. Lines 33 and 34 test to determine whether this move is the best seen so far. If so, we adjust `value` at line 36 and record the move at line 37. At line 41 we return the value of the position in a `Best` object.

Alpha–beta pruning is an improvement to the minimax algorithm.

Although the routine shown in Figure 7.29 optimally solves Tic-Tac-Toe, it performs a lot of searching. Specifically, to choose the first move on an empty board, it makes 549,946 recursive calls (this number is obtained by running the program). By using some algorithmic tricks, we can compute the same information with fewer searches. One such technique is known as *alpha–beta pruning,* which is an improvement to the minimax algorithm. We describe this technique in detail in Chapter 10. Application of alpha–beta pruning reduces the number of recursive calls to only 18,297.

summary

In this chapter we examined recursion and showed that it is a powerful problem-solving tool. Following are its fundamental rules, which you should never forget.

1. *Base cases:* Always have at least one case that can be solved without using recursion.
2. *Make progress:* Any recursive call must progress toward the base case.

```
1     // Find optimal move
2     public Best chooseMove( int side )
3     {
4         int opp;              // The other side
5         Best reply;           // Opponent's best reply
6         int dc;               // Placeholder
7         int simpleEval;       // Result of an immediate evaluation
8         int bestRow = 0;
9         int bestColumn = 0;
10        int value;
11
12        if( ( simpleEval = positionValue( ) ) != UNCLEAR )
13            return new Best( simpleEval );
14
15        if( side == COMPUTER )
16        {
17            opp = HUMAN; value = HUMAN_WIN;
18        }
19        else
20        {
21            opp = COMPUTER; value = COMPUTER_WIN;
22        }
23
24        for( int row = 0; row < 3; row++ )
25            for( int column = 0; column < 3; column++ )
26                if( squareIsEmpty( row, column ) )
27                {
28                    place( row, column, side );
29                    reply = chooseMove( opp );
30                    place( row, column, EMPTY );
31
32                      // Update if side gets better position
33                    if( side == COMPUTER && reply.val > value
34                        || side == HUMAN && reply.val < value )
35                    {
36                        value = reply.val;
37                        bestRow = row; bestColumn = column;
38                    }
39                }
40
41        return new Best( value, bestRow, bestColumn );
42    }
```

figure 7.29

A recursive routine for finding an optimal Tic-Tac-Toe move

3. *"You gotta believe":* Always assume that the recursive call works.
4. *Compound interest rule:* Never duplicate work by solving the same instance of a problem in separate recursive calls.

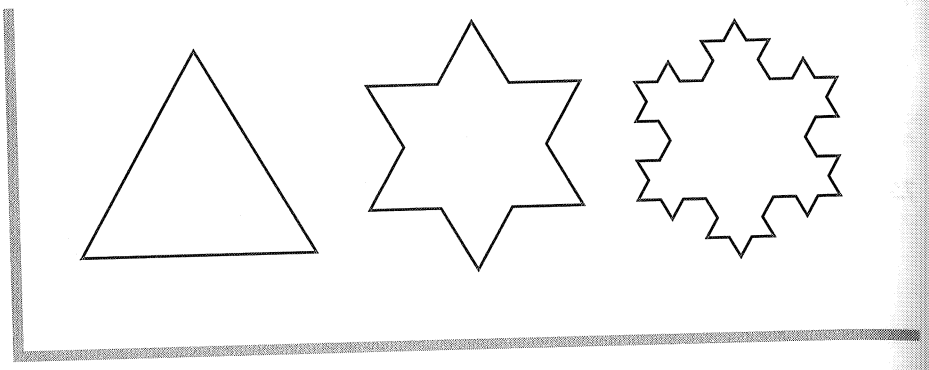

figure 7.30
The first three iterations of a Koch star.

The first three iterations of this procedure are shown in Figure 7.30. Write a Java program to draw a Koch star.

7.36 The method `printReverse` takes a `Scanner` as a parameter, prints each line in the `Scanner` stream, and closes the `Scanner` when it is done. However, the lines are to be output in reverse order of their occurrence. In other words, the last line is output first, and the first line is output last. Implement `printReverse` without using any Collections API or user-written containers. Do so by using recursion (in which you output the first line AFTER recursively outputting the subsequent lines in reverse).

7.37 Function `findMaxAndMin`, defined below is intended to return (in an array of length 2) the maximum and minimum item (if `arr.length` is 1, the maximum and minimum are the same):

```
// Precondition: arr.length >=1
// Postcondition: the 0th item in the return value is the maximum
//                the 1st item in the return value is the minimum
public static double [ ] findMaxAndMin( double [ ] arr )
```

Write an appropriate `private static` recursive routine to implement the `public static` driver `findMaxAndMin` declared above. Your recursive routine must split a problem into roughly two halves, but should never split into two odd-sized problems (in other words, a problem of size 10 is to be split into 4 and 6, rather than 5 and 5).

7.38 a. Design a recursive algorithm to find the longest increasing sequence of numbers in a rectangular grid. For example, if the grid contains

97	47	56	36
35	57	41	13
89	36	98	75
25	45	26	17

then the longest increasing sequence of numbers is the sequence of length eight consisting of 17, 26, 36, 41, 47, 56,

57, 97. Note that there are no duplicates in the increasing sequence.

b. Design an algorithm that solves the same problem but allows for nondecreasing sequences; thus there may be duplicates in the increasing sequence.

7.39 Use dynamic programming to solve the longest increasing sequence problem in Exercise 7.38 (a). *Hint*: Find the best sequence emanating from each grid element, and to do so, consider the grid elements in decreasing sorted order (so that the grid element containing 98 is considered first).

PROGRAMMING PROJECTS

7.40 The binomial coefficients $C(N, k)$ can be defined recursively as $C(N, 0) = 1$, $C(N, N) = 1$ and, for $0 < k < N$, $C(N, k) = C(N - 1, k) + C(N - 1, k - 1)$. Write a method and give an analysis of the running time to compute the binomial coefficients

a. Recursively

b. By using dynamic programming

7.41 Write routine `getAllWords` that takes as parameter a word and returns a `Set` containing all the substrings of the word. The substrings do not need to be real words, nor contiguous, but the letters in the substrings must retain the same order as in the word. For instance, if the word is `cabb`, words that set returned by `getAllWords` would be ["", "b", "bb", "a", "ab", "abb", "c", "cb", "cbb", "ca", "cab", "cabb"].

7.42 Suppose a data file contains lines that consist of either a single integer, or the name of a file that contains more lines. Note that a data file may reference several other files and the referenced files may themselves contain some additional file names, and so forth. Write a method that reads a specified file and returns the sum of all the integers in the file and any files that are referenced. You may assume that no file is referenced more than once.

7.43 Repeat Exercise 7.42, but add code that detects if a file is referenced more than once. When such a situation is detected, the additional references are ignored.

7.44 Method `reverse` shown below returns the reverse of a `String`.

```
String reverse( String str )
```

a. Implement `reverse` recursively. Do not worry about the inefficiency of string concatenation.

b. Implement `reverse` by having `reverse` be the driver for a private recursive routine. `reverse` will create a `StringBuffer` and pass it to the recursive routine.