Recursion is a powerful programming technique that is often difficult for students to understand. The challenge is explaining recursion in a way that is already natural to the student. Many thought processes are recursive in nature and can be used to ease the students into thinking recursively. One danger with teaching recursion is an attempt to solve all problems recursively. There are problems that are elegantly solved using recursion and there are problems that are not. The student should be encouraged to develop and understanding of when to apply recursion.

**Exercises**

12-6 Write a recursive definition of a valid Java identifier (see chapter 2).

An identifier is a letter, underscore, or dollar. An identifier is also an identifier followed by a letter, digit, underscore, or dollar sign.

12-7 Write a recursive definition of \(x^y\) (x to the power y), where x and y are integers and \(y > 0\).

\[N(y) = N(y-1) \times x; \quad N(0) = 1;\]

12-8 Write a recursive definition of \(i \times j\) (integer multiplication), where \(i > 0\). Define it in terms of integer addition.

\[M(i) = M(i-1) + j; \quad M(0) = 0;\]

12-9 Write a recursive definition of the Fibonacci numbers. The Fibonacci numbers are a sequence of integers, each of which is the sum of the previous two numbers. The first two numbers in the sequence are 0 and 1. Explain why you would not normally use recursion to solve this problem.

\[F(1) = 0; \quad F(2) = 1; \quad F(N) = F(N-1) + F(N-2);\]

12-10 Trace the execution of the Palindromes program when it is run against the string "wxyzzyxw". Produce a diagram similar to the one shown in Figure 12.3. Explain the processing of each step.

Not Provided.

12-11 Repeat Problem 12-10 using the string "abcba". What is the primary difference between the string used in Problem 12-10 and this one? Explain how the program handles that difference.

12-12 Repeat Problem 12-10 using the string "abcxycba".

Not Provided.

12-13 Write an iterative version of the Palindromes program. Compare and contrast it with the recursive version presented in this chapter.

Not Provided.
12-14 Write a recursive method to determine and print the value of \( N! \) (N factorial) using the definition given in this chapter. Explain why you would not normally use recursion to solve this problem.

*Not Provided.*

12-15 Write a recursive method to reverse a *String*. Explain why you would not normally use recursion to solve this problem.

*Not Provided.*

### Programming Projects

12-10 Redesign and implement the recursive program that calculates the sum of the integers between 1 and \( N \) to match the recursive definition below. Trace your solution using an \( N \) of 5.

\[
\sum_{i=1}^{N} i = \sum_{i=1}^{\frac{N}{2}} i + \sum_{i=\frac{N}{2}+1}^{N} i
\]

```java
public class Formula {
    public static void main (String[] argv) {
        // Your code here
    }
}
```
```java
int high = 5;
Sigma e = new Sigma();

System.out.println
("The sum all numbers from 1 and " + high + " is: " + e.sum ());

} // method main

} // class Formula
```

```java
// Class Sigma performs the summation for the formula.
// Constructor:
//    public Sigma()
//    public Sigma (int upper)
// Methods:
//    public int sum()
//    private int sum (int lower, int upper)

class Sigma {

    // Starting upper bound
    private int initial_upper;

    // Initialization of and upper bound
    public Sigma (int upper) {
        initial_upper = upper;
    } // constructor Sigma

    // Default constructor
    public Sigma() {
        initial_upper = 5;
    } // constructor Sigma
```
public int sum() {
    return this.sum(1, initial_upper);
} // method sum

private int sum(int lower, int upper) {

    if (lower < upper) {

        int middle = (lower + upper - 1) / 2;

        // Recursive call
        return sum(lower, middle) +
        sum(middle + 1, upper);
    } else {
        return lower;
    }
} // method sum

} // class Sigma

12-11 Design and implement a program that implements Euclid’s algorithm for finding the greatest common divisor of two positive integers. The greatest common divisor is the largest integer that divides both values without producing a remainder. In a class called Divisor_Calc, define a method called gcd that accepts two integers, num1 and num2. The algorithm is defined as follows:

• gcd (num1, num2) is num2 if num2 <= num1 and num2 divides num1
• gcd (num1, num2) is gcd (num2, num1) if num1 < num2
• gcd (num1, num2) is gcd (num2, num1%num2) otherwise

// Euclid.java In Text Application
// Authors: Lewis and Loftus
// Classes: Euclid, Divisor_Calc

Java Software Solutions: Foundations of Program Design 394
// // Class Euclid is a test driver for the Divisor_Calc class. // // Methods: // // public static void main (String[] argv) // // public class Euclid {

public static void main (String[] argv) {

    int num1 = 792;
    int num2 = 348;

    Divisor_Calc euclid = new Divisor_Calc();

    System.out.println
        ("The gcd of " + num1 + " and " + num2 + " is: " +
            euclid.gcd (num1, num2));
} // method main

} // class Euclid

// // Class Divisor_Calc is used to calculate gcd using Euclid's // algorithm // // Methods: // // public int gcd (int num1, int num2) // // class Divisor_Calc {

    // Recursively solves the gcd
    public int gcd (int num1, int num2) {

if (num2 <= num1 && num1 % num2 == 0) {
    return num2;
} else {
    if (num1 < num2) {
        return gcd (num2, num1);
    } else {
        return gcd (num2, num1 % num2);
    }
}

} // method gcd

} // class Divisor_Calc

12-12 Modify the maze problem so that it prints out the path of the final solution without storing it.

//-------------------------------------------------------------------
// Class Maze_Search contains the driver of a program that
// demonstrates the use of recursion to solve a maze.
// Methods:
//
// public static void main (String[] args)
//-------------------------------------------------------------------

public class Maze_Search {
    //
// Creates a new maze, prints its original form, attempts
// to solve it, and prints out its final form.
//===----------------------------------------------------------------------===
public static void main (String[] args) {

    Maze labyrinth = new Maze();

    labyrinth.print_maze();

    if (labyrinth.solve(0, 0))
        System.out.println ("Maze solved!");
    else
        System.out.println ("No solution.");

} // method main

} // class Maze_Search

//===----------------------------------------------------------------------===

//
// Class Maze represents a maze of characters. The goal is to get
// from the top left corner to the bottom right, following a path
// of 1's.
//
// Methods:
//
//    public void print_maze ()
//    public boolean solve (int row, int column)
//    private boolean valid (int row, int column)
//
//===----------------------------------------------------------------------===

class Maze {

    int[][] grid = {{1,1,0,1,1,0,0,0,1,1,1,1},
                    {1,0,1,1,0,1,1,1,0,1,1,1},
                    {0,0,0,1,0,1,0,1,0,1,0,1},
                    {1,1,1,0,1,1,1,0,1,0,1,1},
                    {1,0,1,0,0,0,1,1,0,1,0,1},
                    {1,0,1,1,1,1,1,1,0,1,1,1},
                    {1,0,0,0,0,0,0,0,0,0,0,0},
                    {1,1,1,1,1,1,1,1,1,1,1,1}};

    //----------------------------------------------------------------------

Java Software Solutions: Foundations of Program Design
// Prints the maze grid.
//============================================
public void print_maze () {

    System.out.println();

    for (int row=0; row < grid.length; row++) {
        for (int column=0; column < grid[row].length; column++)
            System.out.print (grid[row][column]);
        System.out.println();
    }
    System.out.println();
}
}  // method print_maze

//============================================
// Attempts to recursively traverse the maze. It inserts
// special characters indicating locations that have been
// tried and that eventually become part of the solution.
//============================================
public boolean solve (int row, int column) {

    boolean done = false;

    if (valid (row, column)) {

        grid[row][column] = 3;  // cell has been tried

        if (row == grid.length-1 && column == grid[0].length-1)
            done = true;  // maze is solved
        else {
            done = solve (row+1, column);  // down
            if (!done)
                done = solve (row, column+1);  // right
            if (!done)
                done = solve (row-1, column);  // up
            if (!done)
                done = solve (row, column-1);  // left
        }
    }

    if (done)  // part of the final path
        System.out.println ("Path: " + row + ", " + column);
}

return done;

}  // method solve

//===========================================================
// Determines if a specific location is valid.
//===========================================================
private boolean valid (int row, int column) {

    boolean result = false;

    // check if cell is in the bounds of the matrix
    if (row >= 0 && row < grid.length &&
        column >= 0 && column < grid[0].length)
        // check if cell is not blocked and not previously tried
        if (grid[row][column] == 1)
            result = true;

    return result;

}  // method valid
}  // class Maze

12-13 Design and implement a recursive program which solves the Non-Attacking Queens problem. Determine how eight queens can be positioned on an eight-by-eight chessboard so that none of them are in the same row, column, or diagonal as any other queen. There are no other chess pieces on the board.

//**************************************************************************
//  Eight_Queens.java          In Text          Application
//  Authors:  Lewis and Loftus
//  Classes:  Eight_Queens
//          Board
//**************************************************************************

// Class Eight_Queens contains the driver of a program that
demonstrates the use of recursion to solve the eight queens
problem.
public class Eight_Queens {
    public static void main (String[] args) {
        Board chess_board = new Board();
        boolean success = false;

        for (int column = 1; (column <= Board.board_size) && ! success; column++) {
            success = Board.place_piece (1, column, 1);
        }

        if (!success) {
            System.out.println ("No configuration possible!");
        }
    } // method main
} // eight_queens
class Board {

    public static int board_size, queens;
    private static boolean board[][];

    //==================================================================
    // Creates a Board of a particular size with a similar
    // number of queens.
    //==================================================================
    public Board (int size) {
        board_size = queens = size;
        board = new boolean [board_size + 1][board_size + 1];
    } // constructor Board

    //==================================================================
    // Creates an 8x8 Board with 8 queens.
    //==================================================================
    public Board () {
        board_size = queens = 8;
        board = new boolean [board_size + 1][board_size + 1];
    } // constructor Board

    //==================================================================
    // Puts a numbered queen in row r and column c.
    //==================================================================
    public static boolean place_piece (int r, int c, int num) {
        boolean success = false;
        if (num == queens) {
            System.out.println("Queen positioned at " + r + ", " + c);
            success = true;
        } else {
            board[r][c] = true;
            for (int row = num + 1; row < board_size+1 && !success; row++) {
                for (int column = 1; column < board_size+1 && !success; column++) {
                    if (!threatened (row, column) && !board[row][column]) {
                        success = place_piece (row, column, num + 1);
                    }
                }
            }
        }
        return success;
    }
}
if (!success) {
    board[r][c] = false;
} else {
    System.out.println("Queen positioned at " + r + ", " + c);
}

return success;
} // method place_piece

//===========================================================
// Returns true if the queen at row r and column c is
// threatened by any other queen.
//===========================================================
private static boolean threatened (int r, int c) {
    return check_for_piece (-1, -1, r, c) ||
    check_for_piece (-1, 1, r, c) ||
    check_for_piece (1, -1, r, c) ||
    check_for_piece (1, 1, r, c) ||
    check_for_piece (0, -1, r, c) ||
    check_for_piece (-1, 0, r, c) ||
    check_for_piece (0, 1, r, c) ||
    check_for_piece (1, 0, r, c);
} // method threatened

//===========================================================
// Returns true if a queen is threatened in a particular
// direction.
//===========================================================
private static boolean check_for_piece (int x_dir, int y_dir, int r, int c) {
    if (c + x_dir < board_size + 1 && c + x_dir > 0 &&
        r + y_dir < board_size + 1 && r + y_dir > 0) {
        if (board[r + y_dir][c + x_dir]) return true;
        else {
            return check_for_piece (x_dir, y_dir, r + y_dir, c + x_dir);
        }
    } else {
        return false;
    }
12-14 In the language of an alien race, all words take the form of Blurs. A Blurb is a Whoozit followed by one or more Whatzits. A Whoozit is the character x followed by zero or more ys. A Whatzit is a q followed by either a z or a d, followed by a Whoozit. Design and implement a recursive program that generates random Blurs in this alien language.

*Not Provided.*

12-15 Design and implement a recursive program to determine if a string is a valid Blurb as defined in Problem 12-14.

*Not Provided.*