Chapter 8: Recursion
Lab Exercises

Computing Powers

Computing a positive integer power of a number is easily seen as a recursive process. Consider a^n:

- If n = 0, a^n is 1 (by definition)
- If n > 0, a^n is a * a^{n-1}

File `Power.java` contains a main program that reads in integers *base* and *exp* and calls method `power` to compute $base^{exp}$. Fill in the code for `power` to make it a recursive method to do the power computation. The comments provide guidance.

```
// ****************************************************************
//   Power.java
//
//   Reads in two integers and uses a recursive power method
//   to compute the first raised to the second power.
// ****************************************************************

import java.util.Scanner;

public class Power
{
    public static void main(String[] args)
    {
      int base, exp;
      int answer;

      Scanner scan = new Scanner(System.in);

      System.out.print("Welcome to the power program! ");
      System.out.println("Please use integers only.");

      //get base
      System.out.print("Enter the base you would like raised to a power: ");
      base = scan.nextInt();

      //get exponent
      System.out.print("Enter the power you would like it raised to: ");
      exp = scan.nextInt();

      answer = power(base,exp);
      System.out.println(base + " raised to the " + exp + " is " + answer);
    }

    // -------------------------------------------------
    //   Computes and returns base^exp
    // -------------------------------------------------
    public static int power(int base, int exp)
    {
      int pow;

      //if the exponent is 0, set pow to 1

      //otherwise set pow to base*base^(exp-1)

      //return pow

    }
}
```

Counting and Summing Digits

The problem of counting the digits in a positive integer or summing those digits can be solved recursively. For example, to count the number of digits think as follows:

- If the integer is less than 10 there is only one digit (the base case).
- Otherwise, the number of digits is 1 (for the units digit) plus the number of digits in the rest of the integer (what's left after the units digit is taken off). For example, the number of digits in 3278 is 1 + the number of digits in 327.

The following is the recursive algorithm implemented in Java.

```
public int numDigits (int num)
{
    if (num < 10)
      return (1);    // a number < 10  has only one digit
    else
      return (1 + numDigits (num / 10));
}
```

Note that in the recursive step, the value returned is 1 (counts the units digit) + the result of the call to determine the number of digits in *num / 10*. Recall that *num/10* is the quotient when *num* is divided by 10 so it would be all the digits except the units digit.

The file `DigitPlay.java` contains the recursive method `numDigits` (note that the method is `static`—it must be since it is called by the static method `main`). Copy this file to your directory, compile it, and run it several times to see how it works. Modify the program as follows:

1. Add a static method named `sumDigits` that finds the *sum* of the digits in a positive integer. Also add code to main to test your method. The algorithm for `sumDigits` is very similar to `numDigits`; you only have to change two lines!

2. Most identification numbers, such as the ISBN number on books or the Universal Product Code (UPC) on grocery products or the identification number on a traveller's check, have at least one digit in the number that is a *check digit*. The check digit is used to detect errors in the number. The simplest check digit scheme is to add one digit to the identification number so that the sum of all the digits, including the check digit, is evenly divisible by some particular integer. For example, American Express Traveller's checks add a check digit so that the sum of the digits in the id number is evenly divisible by 9. United Parcel Service adds a check digit to its pick up numbers so that a weighted sum of the digits (some of the digits in the number are multiplied by numbers other than 1) is divisible by 7.

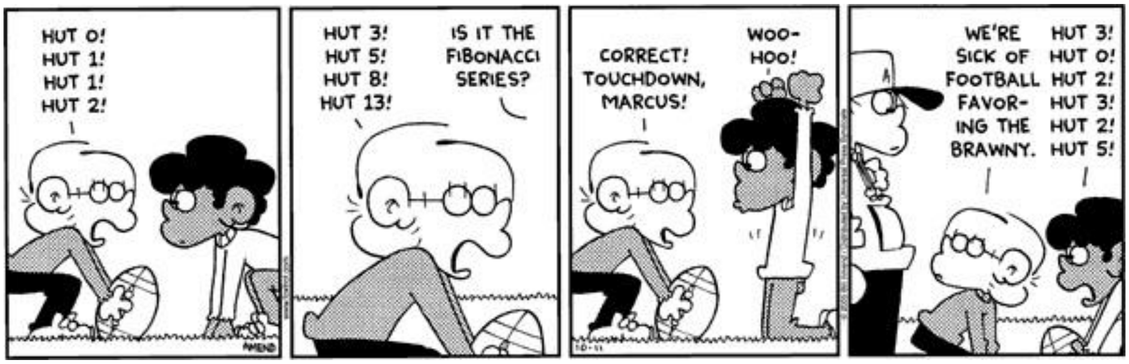
   Modify the `main` method by adding a loop to prompt the user for identification numbers to test. Test your `sumDigits` method to do the following:

- ☐ Input an identification number (a positive integer), then determine if the sum of the digits in the identification number is divisible by 7 (use your `sumDigits` method but don't change it—the only changes should be in main).
- ☐ If the sum is not divisible by 7 print a message indicating the id number is in error; otherwise print an ok message.
- ☐ Test your program on the following input, creating a loop to prompt the user for identification numbers to check:

  a) `3429072` --- error
  b) `1800237` --- ok
  c) `88231256` --- ok
  d) `3180012` --- error

```java
// ****************************************************************
//   DigitPlay.java
//
//   Finds the number of digits in a positive integer.
// ****************************************************************

import java.util.Scanner;

public class DigitPlay
{

    public static void main (String[] args)
    {
      int num;    //a number

      Scanner scan = new Scanner(System.in);

      System.out.println ();
      System.out.print ("Please enter a positive integer: ");
      num = scan.nextInt ();

      if (num <= 0)
          System.out.println ( num + " isn't positive -- start over!!");
      else
          {
            // Call numDigits to find the number of digits in the number
            // Print the number returned from numDigits
            System.out.println ("\nThe number " + num + " contains " +
                                + numDigits(num) + " digits.");
            System.out.println ();
          }

      // add loop to prompt user for identification numbers to check.
      // use the following id num’s as test data:
      //    3429072 --- error
      //    1800237 --- ok
      //    88231256 --- ok
      //    3180012 --- error

    }

    // ---------------------------------------------------------
    //  Recursively counts the digits in a positive integer
    // ---------------------------------------------------------
    public static int numDigits(int num)
    {
      if (num < 10)
          return (1);
      else
          return (1 + numDigits(num/10));
    }
}
```

Efficient Computation of Fibonacci Numbers



The *Fibonacci* sequence is a well-known mathematical sequence in which each term is the sum of the two previous terms. More specifically, if fib(n) is the nth term of the sequence, then the sequence can be defined as follows:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)   n>1
```

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	F_{19}	F_{20}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765

1. Because the Fibonacci sequence is defined recursively, it is natural to write a recursive method to determine the nth number in the sequence. File `Fib.java` contains the skeleton for a class containing a method to compute Fibonacci numbers. Save this file to your directory. Following the specification above, fill in the code for method `fib1` so that it recursively computes and returns the nth number in the sequence.

2. File `TestFib.java` contains a simple driver that asks the user for an integer and uses the `fib1` method to compute that element in the Fibonacci sequence. Save this file to your directory and use it to test your `fib1` method. First try small integers, then larger ones. You'll notice that the number doesn't have to get very big before the calculation takes a very long time. The problem is that the `fib1` method is making lots and lots of recursive calls. To see this, add a print statement at the beginning of your `fib1` method that indicates what call is being computed, e.g., "`In fib1(3)`" if the parameter is 3.

3. Now run `TestFib` again and enter 5—you should get a number of messages from your print statement. Examine these messages and figure out the sequence of calls that generated them. **(First draw the call tree on your assignment sheet.)** Since `fib(5)` is `fib(4) + fib(3)`,you should not be surprised to find calls to `fib(4)` and `fib(3)` in the printout. But why are there two calls to `fib(3)`? Because both `fib(4)` and `fib(5)` need `fib(3)`, so they both compute it—very inefficient. Run the program again with a slightly larger number and again note the repetition in the calls.

4. The fundamental source of the inefficiency is not the fact that recursive calls are being made, but that values are being recomputed. One way around this is to compute the values from the beginning of the sequence instead of from the end, saving them in an array as you go. Although this could be done recursively, it is more natural to do it iteratively. Proceed as follows:
   a. Add a method `fib2` to your `Fib` class. Like `fib1`, `fib2` should be `static` and should take an integer and return an integer.
   b. Inside `fib2`, create an array of integers the size of the value passed in.
   c. Initialize the first two elements of the array to 0 and 1, corresponding to the first two elements of the Fibonacci sequence. Then loop through the integers up to the value passed in, computing each element of the array as the sum of the two previous elements. When the array is full, its last element is the element requested. Return this value.
   d. Modify your `TestFib` class so that it calls `fib2` (first) and prints the result, then calls `fib1` and prints that result. You should get the same answers, but very different computation times.

```
// ****************************************************************
//   Fib.java
//
//   A utility class that provide methods to compute elements of the
//   Fibonacci sequence.
// ****************************************************************
public class Fib
{

    //--------------------------------------------------------------
    // Recursively computes fib(n)
    //--------------------------------------------------------------
    public static int fib1(int n)
    {
      //Fill in code -- this should look very much like the
      //mathematical specification
    }

}
```

```java
// ******************************************************************
//   TestFib.java
//
//   A simple driver that uses the Fib class to compute the
//   nth element of the Fibonacci sequence.
// ******************************************************************

import java.util.Scanner;

public class TestFib
{
    public static void main(String[] args)
    {
      int n, fib;

      Scanner scan = new Scanner(System.in);

      System.out.print("Enter an integer: ");
      n = scan.nextInt();

      fib = Fib.fib1(n);
      System.out.println("Fib(" + n + ") is " + fib);
    }
}
```