Solutions for Exercises Unit 2

This is a non-graded exercise which should be posted to your learning journal.

In many textbooks, the first examples of recursion are the mathematical functions factorial and fibonacci. These functions are defined for non-negative integers using the following recursive formulas:

factorial(0) = 1
factorial(N) = N*factorial(N-1) for N > 0

fibonacci(0) = 1
fibonacci(1) = 1
fibonacci(N) = fibonacci(N-1) + fibonacci(N-2) for N > 1

Write recursive functions to compute factorial(N) and fibonacci(N) for a given non-negative integer N, and write a main() routine to test your functions.

(In fact, *factorial* and *fibonacci* are really not very good examples of recursion, since the most natural way to compute them is to use simple for loops. Furthermore, *fibonacci* is a particularly bad example, since the natural recursive approach to computing this function is extremely inefficient.)

Discussion

The recursive definitions of the two functions can be translated rather easily into Java code. Note that for factorial(N), the base case is N=0, while fibonacci(N) has two base cases, N=0 and N=1. The obvious recursive function for computing fibonacci(N) is:

```
static int fibonacci( int N ) {
   if ( N == 0 || N == 1 ) {
         // Base cases; the answer is 1.
      return 1;
   }
   else {
         // Recursive case; the answer is obtained by applying the function
         // recursively to N-1 and to N-2, and adding the two answers.
      return fibonacci(N-1) + fibonacci(N-2);
   }
}
```

and for factorial(N) is

```
static int factorial( int N ) {
  if ( N == 0) {
        // Base case; the answer is 1.
     return 1;
  }
  else {
       // Recursive case; the answer is obtained by applying the function
       // recursively to N-1 and multiplying the answer by N.
     return factorial(N-1) * N;
  }
}
```

Adding a main() routine that calls these methods for several values of N would be an acceptable solution to the exercise. However, when testing the program, you will quickly notice some problems. First of all, the recursive algorithm for fibonacci(N) is so inefficient that it takes an unreasonably long time for it to run even for values of N as small as 40. (In fact, the recursive evaluation of fibonacci(N) calls both fibonacci(N-1) and fibonacci(N-2) and so takes almost twice as long as the evaluation of fibonacci(N-1); with the computation time for fibonacci(N) almost doubling every time N goes up by 1, the computation time quickly becomes unreasonable.) Although the recursive version of fibonacci is included in the solution below, the program refuses to use this function if N is greater than 40.

Another problem occurs because of the limited size of values of type int. The value of factorial(N) grows very quickly as N increases -- so

quickly that by the time N reaches 13, the value of factorial(N) is already too large to be expressed as a 32-bit integer! So, the factorial function as defined above only gives the correct answer for numbers 0 through 12. Although fibonacci(N) does not grow nearly so fast as factorial(N), it is still true that by the time N reaches 46, fibonacci(N) is outside the range of 32-bit integers.

The problem of the limited size of values of type int was discussed in section 8.1.3 of the textbook. One approach to dealing with the problem was presented in Exercise 8.2: The class java.math.BigInteger represents integer values that can be arbitrarily large (within the limits of the computer's memory). In the solution to the exercise, BigInteger values were used to compute factorial(N) and to compute fibonacci(N) non-recursively. This allows the program to work even for fairly large values of N, say up to a few thousand. Since it's still fun to see a computer working with very large numbers, here is an applet version of the program for you to try:

You can read the solution below to see how it uses the BigInteger class and how it computes fibonacci(N) non-recursively.

The Solution

```
import java.math.BigInteger;

/**
 * Computes factorial(N) and fibonacci(N) for integers N entered by
 * the user, as a demonstration of recursion.
 */
public class FibonacciAndFactorial {


   /**
    * Main routine reads integers N from the user and prints the
    * values of factorial(N) and fibonacci(N), stopping when the
    * user inputs a zero.
    */
   public static void main( String[] args ) {
      while (true) {
         System.out.print("\n\nEnter a postive integer, or 0 to end:  ");
         int N = TextIO.getlnInt();
         if (N == 0)
            break;
         else if (N < 0) {
            System.out.println("Negative numbers are not allowed.");
            continue;
         }
         BigInteger NasBigInteger = BigInteger.valueOf(N);
         System.out.println("\n  factorial(" + N + ") is " + factorial(NasBigInteger));
         if (N > 40) {
            System.out.println("\n  N is too big to compute fibonacci(N) recursively");
         }
         else {
            System.out.println("\n  fibonacci(" + N + ") is " +
                  fibonacci(N) + "   (recursively)");
         }
         System.out.println("\n  fibonacci(" + N + ") is " +
                  fibonacci_nonrecursive(N) + "   (non-recursively)");
      }
   }


   /**
    * Compute fibonacci(N) using recursion.  Because this is so inefficient,
    * even for fairly small values of N, N should be less than or equal to 40.
    * Also, N must be greater than or equal to zero, or an infinite recursion
    * will occur.
    */
   static int fibonacci( int N ) {
      assert N >= 0 : "fibonacci( n) is only defined for non-negative n";
      assert N <= 40 : "n is to large to compute fibonacci(N) recursively";
      if ( N == 0 || N == 1 ) {
            // Base cases; the answer is 1.
```

```
            return 1;
        }
        else {
              // Recursive case; the answer is obtained by applying the function
              // recursively to N-1 and to N-2, and adding the two answers.
            return fibonacci(N-1) + fibonacci(N-2);
        }
    }


    /**
     * Compute fibonacci( N) using a for loop.  The answer is returned as
     * a BigInteger and can be very large even for fairly small values
     * of N.  N must be greater than or equal to zero.
     */
    static BigInteger fibonacci_nonrecursive( int N ) {
      assert N >= 0 : "fibonacci( n) is only defined for non-negative n";
      if (N == 0 || N == 1) {
            // fibonacci(0) = fibonacci(1) = 1;
        return BigInteger.ONE;
      }
      else {
        BigInteger f0 = BigInteger.ONE;  // In the loop, this is fibonacci(i-2)
        BigInteger f1 = BigInteger.ONE;  // In the loop, this is fibonacci(i-1)
        for (int i = 2; i <= N; i++) {
          BigInteger fi = f0.add(f1);  // Computes fibonacci(i)
          f0 = f1;  // Update to account for i++
          f1 = fi;  // Update to account for i++
        }
        return f1;  // Final value of f1 is fibonacci( N)
      }
    }


    /**
     * Compute factorial (N) using recursion.  The computation is done using
     * BigIntegers and can be very large even for fairly small values of N.
     * N must be greater than or equal to zero.
     */
    static BigInteger factorial( BigInteger N ) {
      assert N.signum() >= 0 : "factorial( n) is only defined for non-negative n";
      if ( N.equals(BigInteger.ZERO) ) {
            // Base case; the answer is 1.
        return new BigInteger("1");
      }
      else {
            // Recursive case; the answer is obtained by applying the function
            // recursively to N-1 and multiplying the answer by N.
        BigInteger factorialOfNMinus1 = factorial(N.subtract(BigInteger.ONE));
        return N.multiply(factorialOfNMinus1);
      }
    }

}
```

Last modified: Wednesday, 13 May 2020, 12:53 PM