**Lab - Recursion**

**Introduction:**

In this lab you will gain practice with developing recursive definitions and writing recursive methods in Java. As a refresher and an example consider the factorial function that we saw in class. Recall that the factorial of a positive integer n is the product of all of the integers greater than 0 and less than or equal to n. For example:

5! = 5 \* 4 \* 3 \* 2 \* 1

It is fairly straight forward to implement an *iterative* (non-recursive) method to compute the factorial of a number. For example:

public class Factorial {

public static long iterFactorial(int n) {

long fact = 1;

for (int i=n; i>1; i--) {

fact = fact \* i;

}

return fact;

}

public static void main(String[] args) {

System.out.println("5! = " + iterFactorial(5));

}

}

As we saw in class, the factorial function can also be described recursively as shown in the following *recursive* definition:

n! = 1 if n == 1 {base case}

= n \* (n-1)! otherwise {recursive case}

To understand how this recursive definition allows the factorial of a number to be computed we can *unroll* the recursion. For example:

5! = 5 \* 4!

and 4! = 4 \* 3!

and 3! = 3 \* 2!

and 2! = 2 \* 1!

and 1! = 1

Once the base case has been reached its value (1) can be substituted into the previous recursive expression for 1!:

2! = 2 \* 1!

= 2 \* 1

= 2

Then by continuing to substitute each newly found value into the previous recursive expression we will arrive at the final answer:

3! = 3 \* 2!

= 3 \* 2

= 6

4! = 4 \* 3!

= 4 \* 6

= 24

5! = 5 \* 4!

= 5 \* 24

= 120

With the above recursive definition is now possible to add a recursive method to the Factorial class to compute the factorial function:

public class Factorial {

public static long recFactorial(int n) {

if (n == 1) {

return 1;

}

else {

return n \* recFactorial(n-1);

}

}

public static long iterFactorial(int n) {

long fact = 1;

for (int i=n; i>1; i--) {

fact = fact \* i;

}

return fact;

}

public static void main(String[] args) {

System.out.println("5! = " + iterFactorial(5));

system.out.println("6! = " + recFactorial(6));

}

}

This lab asks you to develop, explain, illustrate recursive definitions for several other functions. You will also implement and test both iterative and recursive versions of each function using Java.

**Setup:**

Before beginning the lab you will need to create a lab7 directory within your cs132 directory. All of your files for this lab must be saved in thiscs132/lab7 directory.

**Assignment:**

Below you will find descriptions of 3 functions. For each of these problems you will need to provide each of the following:

* 1. **Recursive Definition:** Develop and state a recursive definition for the function.
  2. **Unrolling the Recursion:** Give a small example of *unrolling* your recursive definition to reach its base case(s). Give one example for each base case in your recursive definition.
  3. **Illustration:** Illustrate how the base case value is substituted back into previous recursive expressions to compute the final value.
  4. **Java Class:** Write a Java class containing 3 methods:
     + a *recursive* implementation of the function as a Java method.
     + an *iterative* implementation of the function as a Java method.
     + a main method that tests your iterative and recursive methods.

**NOTE:** Parts 1-3 should be typed or neatly hand-written and will be handed in with your lab. Each of your Java classes will be similar to theFactorial class shown above. In addition, your Java classes must contain complete Java Documentation comments.

**The Functions:**

* 1. Fibonacci Numbers:

The fibonacci function, for a positive integer k, computes the kth Fibonacci number. Recall that each Fibonnaci number is the sum of the previous two and that the first two (k=1 and k=2) Fibonacci numbers are 1. So the first 12 Fibonnaci numbers are:

1 1 2 3 5 8 13 21 34 55 89 144 ...

So the operation of the fibonacci function should be as follows:

fibonacci(1) = 1

fibonacci(2) = 1

fibonacci(4) = 1 + 2 = 3

fibonacci(5) = 2 + 3 = 5

fibonacci(10) = 21 + 34 = 55

This one is really a warm-up. We developed a recursive definition for finding the kth Fibonnaci number in class. You should restate this definition for part 1. Then develop a small example for parts 2 and 3. For the iterative method in part 4 you should be able to modify the iterative solution that you wrote for the chapter 6 homework (Chapter 6, P6.7).

For part 4:

* + - * Name the class Fibonacci and save it in a file named Fibonacci.java
      * Both the iterative and recursive methods should accept a single int as an argument and return the answer as a long.
      * Name the iterative method iterFibonacci.
      * Name the recursive method recFibonacci.
  1. Summation:

The summation function, for a non-negative integer argument n, computes the sum of all of the integers from 0 up to and including n. For example:

summation(0) = 0

summation(4) = 0 + 1 + 2 + 3 + 4 = 10

summation(6) = 0 + 1 + 2 + 3 + 4 + 5 + 6 = 21

For part 4:

* + - * Name the class Summation and save it in a file named Summation.java
      * Both the iterative and recursive methods should accept an int value as an argument and return the answer as a long.
      * Name the iterative method iterSummation.
      * Name the recursive method recSummation.
  1. Exponentiation:

The exponent function computes the value of one non-negative integer raised to the power of a second non-negative integer. For example:

exponent(3,4) = 3 raised to the 4th power = 81

exponent(6,2) = 6 raised to the 2nd power = 36

exponent(9,0) = 9 raised to the 0th power = 1

exponent(1,8) = 1 raised to the 8th power = 1

exponent(0,8) = 0 raised to the 8th power = 0

For part 4:

* + - * Name the class Exponent and save it in a file named Exponent.java
      * Both the iterative and recursive methods should accept two int values as arguments and return the answer as a long.
      * Name the iterative method iterExponent.
      * Name the recursive method recExponent.

**Bonus:**

It is possible to have a method be recursive without calling itself directly. For example, method foo might call method bar and methodbar might call method foo. The methods foo and bar would be called *mutually recursive*. In a class named EvenOdd write two mutually recursive methods: one called isEven the other called isOdd that determine if their argument is even or odd, respectively. For example:

isEven(7) = false

isOdd(7) = true

The isEven and isOdd methods should accept a single int as an argument and return a boolean.

Hint: A number is even if the number one less than it is odd.

Why, you ask would anyone write this in such a silly way? Why, for practice and understanding of recursion! What more noble goal could one have?