LAB 3: RECURSION

F27SG – SOFTWARE DEVELOPMENT 3 (5 MARKS)

1. RECURSIVE METHODS (3 POINTS)

In this part you will implement some recursive methods. You need to start a new project from scratch.

A. RECURSIVE SUM OF NUMBERS

Write a recursive method `sum` that, given a number n, returns the sum of all positive numbers up to n - that is, it computes:

```
sum(n) = 1 + 2 + ... + n
```

See **Hint 1** if you are stuck.

B. RECURSIVE MULTIPLICATION USING ONLY ADDITION

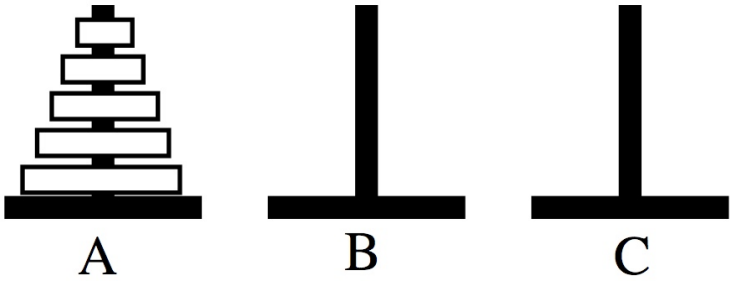
Write a recursive method `multiply` which given two (positive) integers `m` and `n` as arguments, computes `m*n` using only addition (and recursion). See **Hint 2** if you are stuck.

C. COMPUTE FIBONACCI NUMBER

The Fibonacci sequence is 0,1,1,2,3,5,8,13,21,34,.... You should write a method `Fibonacci` which given n as an argument, returns the nth Fibonacci number using recursion, that is `Fibonacci(0) = 0, FiboNacci(1) = 1, FiboNacci(2) = 1, FiboNacci(3) = 2, FiboNacci(4) = 3, FiboNacci(5) = 5, FiboNacci(6) = 8, FiboNacci(7) = 13, FiboNacci(8) = 21` and so on. See **Hint 3** if you are stuck.

2. THE TOWER OF HANOI (2 POINTS)

The Tower of Hanoi is a game that has a very elegant recursive solution. The goal is to move all rings of tower A to tower C:



However, you have to adhere to the following rules

- you can only move one ring at a time
- you can only move the top ring
- you cannot put a larger ring on top of a smaller ring

To better understand the game, there are several places online where you can play it. For example here:

http://www.mathsisfun.com/games/towerofhanoi.html

Your job is to implement a recursive method that solves the game for *n* number of rings. You should use Strings to represent the name of the towers (for example “A”,”B” and “C”):

```
public static void movePile(int n,String from,String to,String via){

   // your code

}
```

Use the following method to represent the move of a single ring between two towers (provided in `Lab3.java`):

```
 public static void moveOneRing(String from,String to){

   System.out.println("Ring moved from " + from + " to " + to);

 }
```

See **Hint 4** if you get stuck.

ADDITIONAL CHALLENGE: TAIL RECURSION

Many problems have very elegant recursive solutions. However, when using recursion extra memory is used since we have to push each method call to the program stack (see lecture 3). For example, in the factorial example

```
public static int factorial(int n){
      if (n == 0) return 1;
      else  return n*factorial(n-1);
}
```

each call is pushed to the program stack, and when we reach the base case each step is popped and the result is multiplied with n. A more efficient type of recursion is called **tail recursion** and it happens when the recursive call is the very last operation of a method. In such cases the compiler can optimize the code by turning the recursion into normal iteration. For example, we can turn the factorial method into a tail recursive version by carry with us the result so far:

```
public static int tailFactorial(int n,int res){
      if (n == 0) return res;
      else return tailFactorial(n-1,n*res);
}
```

On termination we do not need to step back through each call made, but we can return the value directly. The task here is to turn the `sum` and `multiplication` methods from part 1 into tail recursive variants `tailSum` and `tailMultiply`:

```
public static int tailSum(int n,int sum){
      // your code
}
public static int tailMultiply(int m,int n,int sum){
   // your code
}
```

HINTS

Hint 1! *The method is very similar to the factorial function from Lecture 5.*

Hint 2! *m * n can be seen as m added together n times. This can be implemented recursively where each step adds m to the result of m * (n-1), until n reaches 0.*

Hint 3! *A Fibonacci number is the sum of the previous two Fibonacci numbers. You therefore need to two base cases (why?), which are the first two numbers in the sequence.*

Hint 4! *If you want to move N rings from tower A to tower B using tower C, then you can reduce the problem to first move the top (N-1) rings from tower A to tower C (using B), then move ring N from A to B, and then move the top (N-1) from C to B using A. If you continue the recursion this way then you will end up with a base case when N=1 where you only have to move one ring.*