CS2133: Computer Science II
Assignment 2

Prof. Christopher Crick

1 Recursion (10 points)

Create a file called Factorial.java. This file should have the following method:

```
public static long calculate(long n)
```

Factorial.calculate should recursively calculate $n!$, where $0! = 1$ and $n! = n(n-1)!$. The method should also print out an error and exit if $n < 0$ or $n > 20$, since factorial is not defined for negative numbers and will overflow Java's long variable with larger numbers (if we used an int, it would overflow even sooner!).

Inside Factorial.java, also include a main method which runs a couple of tests on Factorial.calculate:

```
java Factorial
Factorial.calculate(0) returned 1.  Test passed!
Factorial.calculate(5) returned 120.  Test passed!
```

(Obviously, if these tests do not return 1 and 120 respectively, the tests should fail and print out an appropriate message.)

2 Fibonacci revisited (10 points)

Create a file called FibTest.java. Refactor your Fib.java from assignment 1 to be a static method in FibTest. Odds are that your assignment 1 answer is coded in iterative style, where you are looping from 1 to n and adding up numbers as you go. If not, you should write an iterative Fibonacci calculator that does so (and refactor the version that you wrote for assignment 1 for the next part of the problem). This method should be called

```
public static int fibIter(int n)
```

The recurrence relation for the Fibonacci sequence is the following:

$$\begin{aligned} fib(1) &= 1 \\ fib(2) &= 1 \\ fib(n) &= fib(n-1) + fib(n-2) \end{aligned}$$

Write another static method in the same FibTest class, this time in recursive style, using this relationship.

```
public static int fibRecur(int n)
```

In FibTest's main method, write a series of tests that establishes that both of these functions work correctly. Finally, using Java's **System.currentTimeMillis()** function, print out the time it takes to execute FibTest.fibIter(40) and FibTest.fibRecur(40).

3 π revisited (10 points)

In Assignment 1, you used the Gregory formula to compute an approximation of π. Here, you will use Ramanujan's series, discovered in 1910. It converges *much* faster than Gregory's, and has been used to calculate π to billions of digits. Implement Ramanujan.java exactly like Gregory.java, taking a number n specified by the user on the command line and calculating π using the first n terms of the Ramanujan series. The program should print this approximate value of π, as well as the percentage error between this value and the one provided by Java in the constant Math.PI.

Ramanujan's series: $\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103+26390k)}{(k!)^4 396^{4k}}$

Notice that this formula makes use of the factorial function. Call your Factorial.calculate method from Problem 1.

Extra credit: Look at the Java API definitions of BigDecimal and BigInteger. Rework your Factorial and Ramanujan code (call the classes something else, like RamanujanBig, so that you don't overwrite your regular-credit work) to use these instead, and calculate π to 100 digits or more. Note that, in order for the terms to be perfectly accurate, you also need a BigDecimal representation of $\sqrt{2}$. There is unfortunately no native library support for calculating this; however, you can implement it yourself using a BigDecimal version of the evaluate method you will write in Problem 6.

You will get a little extra credit simply by explaining (in comments in your Ramanujan code) how BigDecimal and BigInteger differ from ordinary doubles and ints, and why the latter are insufficient for calculating very precise values of π.

4 Root finding (20 points)

The roots of a continuous function are points at which the value of the function is equal to zero. If we happen to know a value for which the function is positive and another value for which it is negative, then we know for certain that somewhere between the two values lies a zero. In other words, if $f(a) > 0$ and $f(b) < 0$, then there exists a value x between a and b for which $f(x) = 0$.

This fact suggests that we can zero in on the value of x using the following algorithm, where ϵ is the amount of error we are willing to tolerate:

1. Compute x, where x is halfway between a and b, ie $x = \frac{(a+b)}{2}$.
2. If $|a - x| < \epsilon$ return x.
3. If $f(x)$ has the same sign as $f(a)$, the root lies between x and b. Recursively perform the algorithm with x as the new a.
4. Otherwise, the root lies between a and x. Recursively perform the algorithm with x as the new b.

Write a class FunctionTest.java and implement the following method:

```
public static double findRoot(double a, double b, double epsilon)
```

Within findRoot(), use Math.sin() as the function to be evaluated. In your main function, print out the root of $sin(x)$ that falls between 3 and 4, to within 0.00000001. Does the number look familiar?

5 Polynomials (30 points)

Write a class Poly.java to represent polynomials (ie, functions of the form $ax^n + bx^{n-1} + \cdots + cx^2 + dx + e$). Implement the following methods:

- Poly(int[] coefficients): Constructor for an array of coefficients where $c[n]$ is the coefficient of x^n. In other words, the polynomial $2x^5 + 3x^4 - 8x^2 + 4$ would be represented by the array $[4, 0, -8, 0, 3, 2]$. This way, if I want to know the coefficient of x^2, I look at $c[2]$, and get -8. This convenience is worth the confusion that arises because we usually write arrays left-to-right starting from the smallest index, whereas we usually write polynomials left-to-right starting from the largest coefficient.
- int degree(): Return the power of the highest non-zero term
- String toString(): Overriding the Object class's toString method, this should return a String representation of the polynomial using x as the variable, arranged in decreasing order of exponent and printing nothing for terms with a coefficient of zero. For example, the Poly represented by the array $[4, 0, -8, 0, 3, 2]$ should return the string:

  ```
  2x^5+3x^4-8x^2+4
  ```

- Poly add(Poly a): To add two polynomials, simply add together each scale degree in turn. Thus, $(2x^5 + 3x^4 - 8x^2 + 4) + (x^3 + 4x^2 - 2x) = (2x^5 + 3x^4 + x^3 - 4x^2 - 2x + 4)$. Create a method that constructs and returns a new Poly object with a new coefficient array created by adding the coefficients of the current object and the Poly object a passed as an argument to the add() function.
- double evaluate(double x): Return the value of the function at the point x. In other words, if the Poly object represents $2x^5 + 3x^4 - 8x^2 + 4$ and evaluate(2.0) is called, the method should calculate $2(2)^5 + 3(2)^4 - 8(2)^2 + 4$ and return 84.

Within the main() method of the Poly class, create a series of tests which exercise each of these methods and report their success or failure.

6 Inheritance (20 points)

The findRoot() function in FunctionTest.java works great, but only specifically for the sine function. That's nice and all, but hardly general. We would like to be able to use the same code for any function at all, and we will use inheritance to make that happen. Create a new class called Function.java that supports an abstract method:

```
public abstract double evaluate(double x)
```

Refactor your FunctionTest.findRoot() method so that it belongs to the Function class, and instead of calling Math.sin(), it calls this evaluate() method. Note that findRoot() will no longer be static.

Write a class SinFunc.java that extends Function and implements evaluate() using Math.sin(). Write a similar class CosFunc.java that does the same for Math.cos(). Copy your code from Poly.java into a new class PolyFunc.java which also extends Function (you shouldn't have to make any substantive changes, but you will have to rename occurrences of the "Poly" type inside your code to "PolyFunc"). Look at that – you've already implemented evaluate() for polynomials!

In Function.main(), create some tests. Instantiate a few functions and find some roots. Verify that the root of $sin(x)$ between 3 and 4 is the same as it was before. Find the root of $cos(x)$ between 1 and 3. Find the positive root ($x > 0$) of $x^2 - 3$ and $x^2 - x - 2$.

Finally, for all of the classes in this problem (Function.java, SinFunc.java, CosFunc.java and PolyFunc.java), write javadoc comments explaining the purpose of each class and the uses of each method.

Turning in

You should have created nine Java files for this assignment: Factorial.java, FibTest.java, Ramanujan.java, FunctionTest.java, Poly.java, Function.java, SinFunc.java, CosFunc.java and PolyFunc.java. Ensure that all of them can be compiled and run from the command line. Create a zip file called assignment_2_your_name.zip and upload it to the Dropbox at oc.okstate.edu. This assignment is due Wednesday, September 14, at noon.