Homework 8

In this assignment you have to implement six different programs. Each program must be implemented recursively. Since the purpose of this assignment is to practice recursion, non-recursive solutions will not be accepted.

1. From Decimal To Binary

The void function `integerToBinary(int n)` prints the binary representation of `n`.

Assume that `n` is non-negative.

Here is an example of the function's execution:

```
% java IntegerToBinary 8
1000
% java IntegerToBinary 23
10111
% java IntegerToBinary 563
1000110011
```

Slides 20-21 in lecture 2-2 present an iterative (non-recursive) implementation of this operation.

Implement the `integerToBinary` function, recursively.

Implementation tip: Each step of the recursion should print one binary digit, and call the `IntegerToBinary` function with a smaller `n`, not necessarily in this order.

2. Palindromes

A *palindrome* is a string that reads the same forward or backward. For examples, "`madam`" is a palindrome, and so is "`aviddiva`". A string of length 0 or 1 is considered a palindrome. Slide 9 in lecture 2-2 presents an iterative (non-recursive) algorithm for checking if a given string is a palindrome.

The boolean function `palindrome(String s)` checks if the given string is a palindrome, recursively. Here is an example of the function's execution:

```
% java Palindrome you
false
% java Palindrome kayak
true
```

Assume that the string contains only letter characters (no spaces, no punctuation marks).

Implement the `Palindrome` function, recursively.

Implementation tip: To make the string smaller in each recursive step, use Java's `substring` function, whose description can be found in the String class API.

3. Binomial Coefficient

How many different Poker hands of 5 cards can be drawn from a deck of 52 cards? The answer is 2,598,960. In general, how many subsets of *k* elements exist in a set of *n* elements? The answer is given by a famous function called "the Binomial coefficient", denoted and computed as follows:

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

If $k > n$, the function is defined as $\binom{n}{k} = 0$.

If $k = 0$, or $n = 0$, the function is defined as $\binom{n}{k} = 1$.

Binomial coefficients have numerous practical applications, and many interesting mathematical properties. One of them is the following useful result, known as Pascal's identity:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

This recursive formula can be readily implemented as follows:

```
public static int binomial(int n, int k) {
    if (k > n) return 0;
    if (k == 0 || n == 0) return 1;
    return binomial(n - 1, k) + binomial(n - 1, k - 1);
}
```

Implement the function shown above, and play with it. Use it to compute how many teams of 11 football players can be made from 20 players. Then try to compute the same with 30 players. Then with 40 players. The computation of $\binom{40}{11}$ will take a while, resulting with a negative number! That's an example of *overflow*: The number of 11-player teams chosen from 40 players is 2,311,801,440, which is greater than the maximum `int` value in Java. To check your calculations, you can use this binomial coefficient calculator.

If you will change the function's implementation to return a `long` value instead of an `int` value, it should work fine, as long as the result is no greater than 9,223,372,036,854,775,807, which is the maximum `long` value in Java.

Now try to compute how many teams of 20 players can be made from 50 players. But don't hold your breath: The time to compute $\binom{50}{20}$ using the above algorithm is longer than your lifetime. The problem is this: Even though the algorithm shown above is correct, it suffers from the same handicap of the Fibonacci function described in lecture 8-1: It computes the same binomial functions over and over, exponentially. Therefore, it is terribly inefficient.

Implement the `binomial(int n, int k)`, *efficiently*.

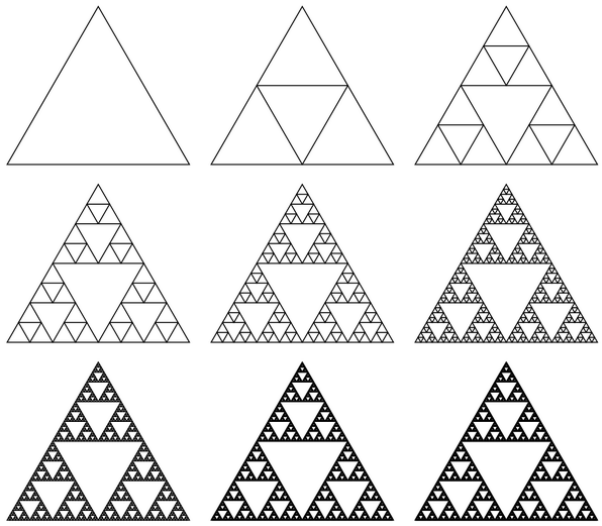
Implementation tip: Start by implementing (copy-pasting) the code shown above. Then modify your implementation it to use a version of the *memoization technique* discussed in the Week 8 Recitation. If you'll do it right, your program should be able to calculate $\binom{50}{20}$ in a reasonable time.

4. Sierpinski's triangle

In this exercise we revisit the fractal known as *Sierpinski's triangle*. In slides 6-9 of lecture 6-1 we showed how this fractal can be created using an iterative and random approach. Here is another, more direct strategy for creating this fractal:

1. Draw an equilateral triangle.
2. Subdivide it into four smaller quilateral triangles.
3. Ignore the central triangle, and repeat step 2 on the remaining triangles.

Below is an illustration that shows the first 9 steps of this algorithm:



The void function `sierpinski(int n)` draws a Sierpinski triangle of depth `n`. Your implementation should use the `StdDraw` library. Decide yourself where to place the fractal on the canvas, and what should be the equilateral triangle's size.

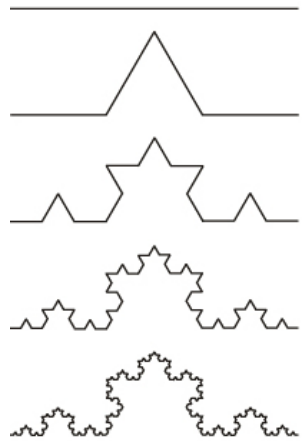
Implementation tip: One way to control the recursion depth is to use a parameter than becomes smaller in each recursive step. See the given `Sierpinski` class skeleton.

5. Koch Snowflake

The so-called *Koch Curve* (not the snowflake) can be generated as follows:

1. Draw a straight line.
2. Divide it into 3 equal segments.
3. Construct an equilateral triangle on the middle segment, removing the original segment.
4. Repeat step 2 and 3 on each one of the resulting 4 segments.

Below is an illustration of the first 5 steps of this algorithm:

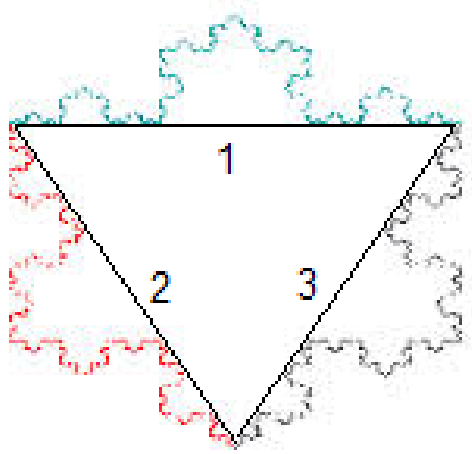


In the `Koch` class, the void function `curve(int n, double x1, double y1, double x2, double y2)` draws a Koch Curve of depth `n` with endpoints at `(x1, y1)` and `(x2, y2)`.

Implement the `curve` function, recursively, and test it.

You may find the following result useful: Let $p1 = (x_1, y_1)$ and $p2 = (x_2, y_2)$ be two points in the plane, and let L be the line that connects them. If you wish to construct an equilateral triangle based on the middle segment of L, then the coordinate of the triangle's vertex lying outside L is $p3 = \left(\frac{\sqrt{3}}{6}(y_1 - y_2) + \frac{1}{2}(x_1 + x_2) \ , \ \frac{\sqrt{3}}{6}(x_2 - x_1) + \frac{1}{2}(y_1 + y_2) \right)$

The so-called Koch Snowflake is constructed by joining 3 Koch Curves in the following manner:



In the `Koch` class, the void function `snowflake(int n)` constructs a Koch snowflake of depth `n`.

Implement the `kochSnowflake` function (which uses the `curve` function).

Submission

Before submitting your work for grading, make sure that your code is written according to our Java Coding Style Guidelines. In addition to the 5 code files, create one PDF document that lists the code of the programs that you wrote, with the indentation. You may want to experiment with several "paste special" options, for transferring the code from your code editor to your word processing software. Whatever you do, the printed code must be well indented, and easy to read a review. Use the font Consolas, or Arial, size 12. The name of this file should be HW8Code.pdf.

Submit the following code files:

- `IntegerToBinary.java`
- `Palindromes.java`
- `Binomial.java`
- `Sierpinski.java`
- `Koch`

Compress the 5 files into a file named HW8.zip. Upload two filed to Moodle: HW8.zip, and HW8Code.pdf.

Submission deadline: December 12, 2021, 23:55.

Get feedback: To get feedback (without grading) about your programs before submitting them, use GETFEED, anytime, as many times as you want.