Homework 7

In this assignment you have to implement several programs, **recursively**. Since the purpose of this assignment is to practice recursion, non-recursive solutions will not be accepted.

1. From Decimal To Binary

Class `intToBin` features the single function `toBinary(int x)`. The function returns the binary representation of `x`, as a string of `'0'` and `'1'` characters. Here is an example of the function's execution:

```
% java IntToBin 5
101
% java IntToBin 8
1000
% java IntToBin 1
1
% java IntToBin 0
0
% java IntToBin 712
1011001000
```

Slide 21 in lecture 2-2 presents an iterative (non-recursive) implementation of this operation.

Implement the `toBinary` function, recursively.

Implementation tips: The base cases are x = 0 and x = 1. In that case the binary representations are simply the strings “0” and “1”. The recursive step consists of three conceptual actions: computing the rightmost binary digit of x, calling the `toBinary` function on a reduced version of x, and combining the results.

2. Palindromes

A palindrome is a string that reads the same forward and backward. For example, "aba" is a palindrome, and so is "madam". 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.

Class `Palindrom` features the single boolean function `isPalindrome(String s)`, that checks if the given string is a palindrome. Here is an example of this program’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, you can 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 $\binom{n}{k}$, and computed by the following formula:

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

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

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

The Binomial function has numerous practical applications, and 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 binomial1(int n, int k) {
    if (k > n) return 0;
    if (k == 0 || n == 0) return 1;
    return binomial1(n - 1, k) + binomial1(n - 1, k - 1);
}
```

Implement the function shown above (all you have to do is copy it from this document...), and play with it. Notice that we called it `binomial1`, for a reason that will become clear later. Use `binomial1` 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 in 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. You don't have to do it for this exercise).

How many teams of 20 players can be made from 50 players? The result is an astronomical number, and the time required to compute it using the above algorithm is longer than your lifetime. The problem is this: The basic `binomial1` function described above 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. One way to optimize this computation is to use *memiozation*.

Implement an optimized version of the `binomial(int n, int k)` function, using memoization.

Implementation tips

In the skeletal implementation that we supplied, the `binomial` function is *overloaded*, having two signatures: `binomial(int, int)` and `binomial(int, int k, int[][])`.

`binomial(n, k)`: That's the function call we use for computing $\binom{n}{k}$. But, this function only "sets the stage": It creates a 2D array, let's call it `memo`, and then initializes all its elements to `-1`. Finally, it calls `binomial(n, k, memo)`, which does all the heavy lifting.

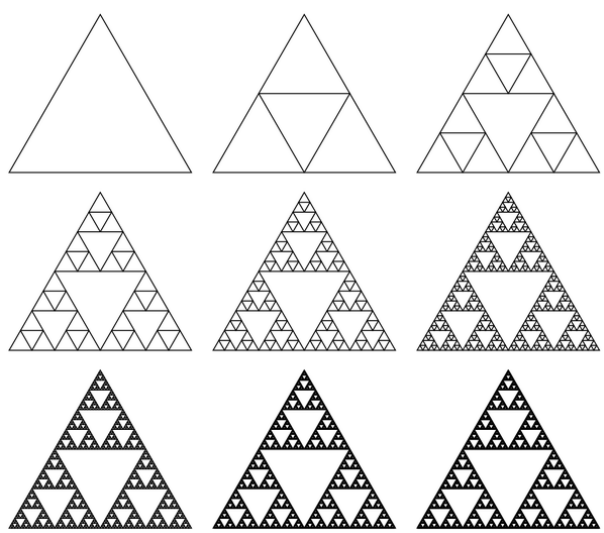
`binomial(n, k, memo)`: This function uses a very similar recursive implementation to that of the basic `binomial1` function. The key difference is that before computing anything, it first checks the `memo` array, to see if it already made this computation. If so, the function returns the value stored in `memo`. Otherwise, the function computes the value, stores it in `memo`, and returns it. Notice that `memo` must be passed, as a parameter, in every recursive call. That's why we called it `memo`: It serves as a shared memory throughout the recursive computation process.

4. Sierpinski's triangle

In this exercise we revisit the fractal known as *Sierpinski's triangle*. In Lecture 8-1 we showed how this fractal can be created using an iterative, random implementation. Here is another, more direct strategy for creating this fractal:

1. Draw an equilateral triangle.
2. Subdivide it into four smaller equilateral 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`.

Implementation notes

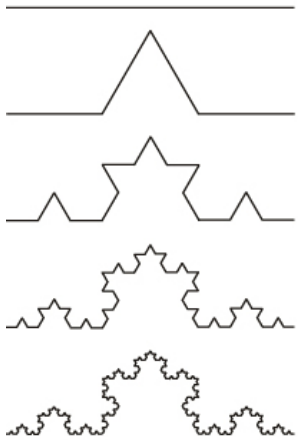
1. Use the standard canvas. Compute the triangle's height, and decide where to draw it in the canvas (figure out the coordinates).
2. Use `StdDraw` to do the line drawings. If you want to erase some of the drawings, you can redraw them using the standard background color, which is *white*.
3. To switch to drawing using white color, use `StdDraw.setPenColor(StdDraw.WHITE);`
   To switch back to black, use `StdDraw.setPenColor(StdDraw.BLACK);`
   (`BLACK` and `WHITE` are two fixed `Color` objects, implemented in the `StdDraw` class as class-level `final static` variables).
4. The depth of the recursion can be controlled by passing a parameter that becomes smaller in each recursive step. See the given `Sierpinski` class skeleton.
5. The overall implementation is very similar to the technique we used in the implementation of the "H fractal" in lecture 7-1.

5. Snowflakes

The so-called *Koch Curve* can be used to create fractal, snowflake-like images. The curve 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 steps 2 and 3 on each one of the resulting 4 segments.

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

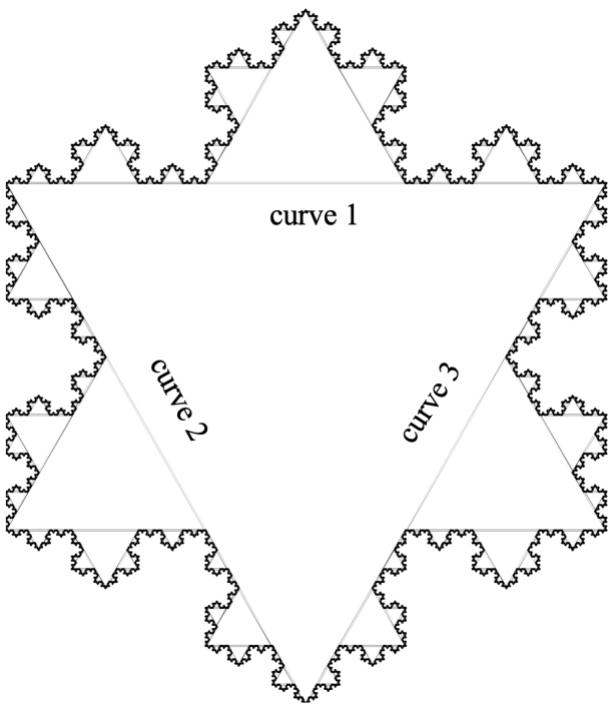


In the `Koch` class, the function `curve(int n, double x1, double y1, double x2, double y2)` draws a Koch curve of depth `n` on the standard canvas. The drawing starts at coordinate `(x1, y1)` and ends at coordinate `(x2, y2)`.

Implement the `curve` function, recursively.

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 $p3 = (\frac{\sqrt{3}}{6}(y_1 - y_2) + \frac{1}{2}(x_1 + x_2)\ ,\ \frac{\sqrt{3}}{6}(x_2 - x_1) + \frac{1}{2}(y_1 + y_2)\ )$

The so-called Koch Snowflake can be constructed by joining three Koch curves, as follows:



This implementation of this function is given. Of course, it will work only after you implement the `curve` function. This is a nice drawing, and it may be rewarding to sit back and watch the computer draw it, using your `curve` implementation.