Introduction to Computer Science, IDC Herzliya, 2016

HW 8: Recursion

In this assignment you will write gain practice in recursive programming. For each exercise, you must present a recursive solution; other solutions will not be accepted. In particular, **the use of loops is not allowed.** Your implementation may or may not include private functions, as you see fit.

Before you start implementing any program, it's essential to write pseudo-code on paper, trace its execution *on paper*, and convince yourself that you got the logic right.

1. **Multiplication**

   The function `public static int multiply(int a, int b)` returns the product of two integers. We assume that both of them are non-negative. Here is an example of the function's execution:

   ```
   % multiply(3,5)
   15
   ```

   Implement this function recursively, **without using Java's * (multiplication) operator** .

   Implementation hint: In some recursive implementations (e.g. `listPermutations`), we design and write two functions, of which the user gets to see only one: a "top" `public` function for getting the process going, and a second `private` function, which we call $auxiliary$, that actually gets the job done (thanks to overloading, both functions can have the same name). This setting has several advantages. One of them is that it enables us to put into the signature of the auxiliary function additional parameters that can help us in the implementation (once again, note that the user sees only the top function). For example, if we want to pass information from one recursive call to another, we can do so by using some additional parameter, designed for this purpose (e.g. the `prefix` parameter in `listPermutations`), and put it in the auxiliary function's signature.

   To make a long story short, you may want to implement your recursive multiplication solution using both a top function and an auxiliary function. If so, you need to decide what parameter in addition to a and b the auxiliary function will need to have. Note that the top function (the one that the user sees) can be used to initialize this parameter, and then get the auxiliary function into motion.

2. **Greatest Common Divisor**

   The greatest common divisor (gcd) of two non-negative integers (at least one of which is non-zero) is the largest positive integer that divides both integers with no remainder. For example, the gcd of 12 and 8 is 4. We can compute the gcd recursively using the following three observations:

   a. If p, q are positive integers such that $p > q$ then the gcd of p and q is equal to the gcd of q and $p\%q$ (the remainder left when dividing p by q). For example, the gcd of 12 and 8 is equal to the gcd of 8 and 4.
   b. $p\%q$ (the remainder left when dividing p by q) is an integer strictly smaller than q .
   c. If p is a positive integer, then the gcd of p and 0 is p . For example, the gcd of 4 and 0 is 4.

When we talked about the `Fraction` class, we used a gcd function in order to reduce the fraction (for example, reducing 20/40 to 1/2 by dividing both 20 and 40 by gcd(20,40)). But, we didn't show the gcd implementation. Well, now is the time to plug this hole. Implement the function `public static int gcd(int p, int q)`, which returns the gcd of its two parameters. Assume that both parameters are non-negative integers, and that at least one of them is non-zero. Here is an example of the function's execution:

```
% gcd(21,18)
3
```

3. **Maximal Path**

Given an $m \times n$ rectangular array of non-negative integers, a $path$ through the array is constructed as follows:

1. Start at the top-left corner, coordinates $(0,0)$, and end at the bottom-right corner, coordinates $(m-1,n-1)$.
2. At each step, move either one cell to the right, or one cell down.

For example, consider the following 3×3 array:

3	4	5
2	2	0
1	0	1

Here are some examples of valid paths:

3	**4**	**5**
2	2	**0**
1	0	**1**

3	**4**	5
2	**2**	**0**
1	0	**1**

3	**4**	5
2	**2**	0
1	**0**	**1**

The cost of a path is defined as the sum of all the values along the way. For example, the cost of the first path above is 13, while the cost of the second and third paths is 10. We say that a path is $maximal$ if it has the largest cost among all possible paths.

a. Write a function `public static int maxPath(int[][] arr)` that computes the cost of the maximal path in the given array.

   Hint: suppose we want to compute the cost of the maximal sub-path starting at some location (i,j) and ending at the bottom-right corner. We can do this by adding the cost of the value at (i,j) to the maximum of the following two values: the cost of the maximal path starting at $(i+1,j)$ and the cost of the maximal path starting at $(i,j+1)$.

Here is an example of the function's execution on the array given above:

```
% maxPath(arr)
13
```

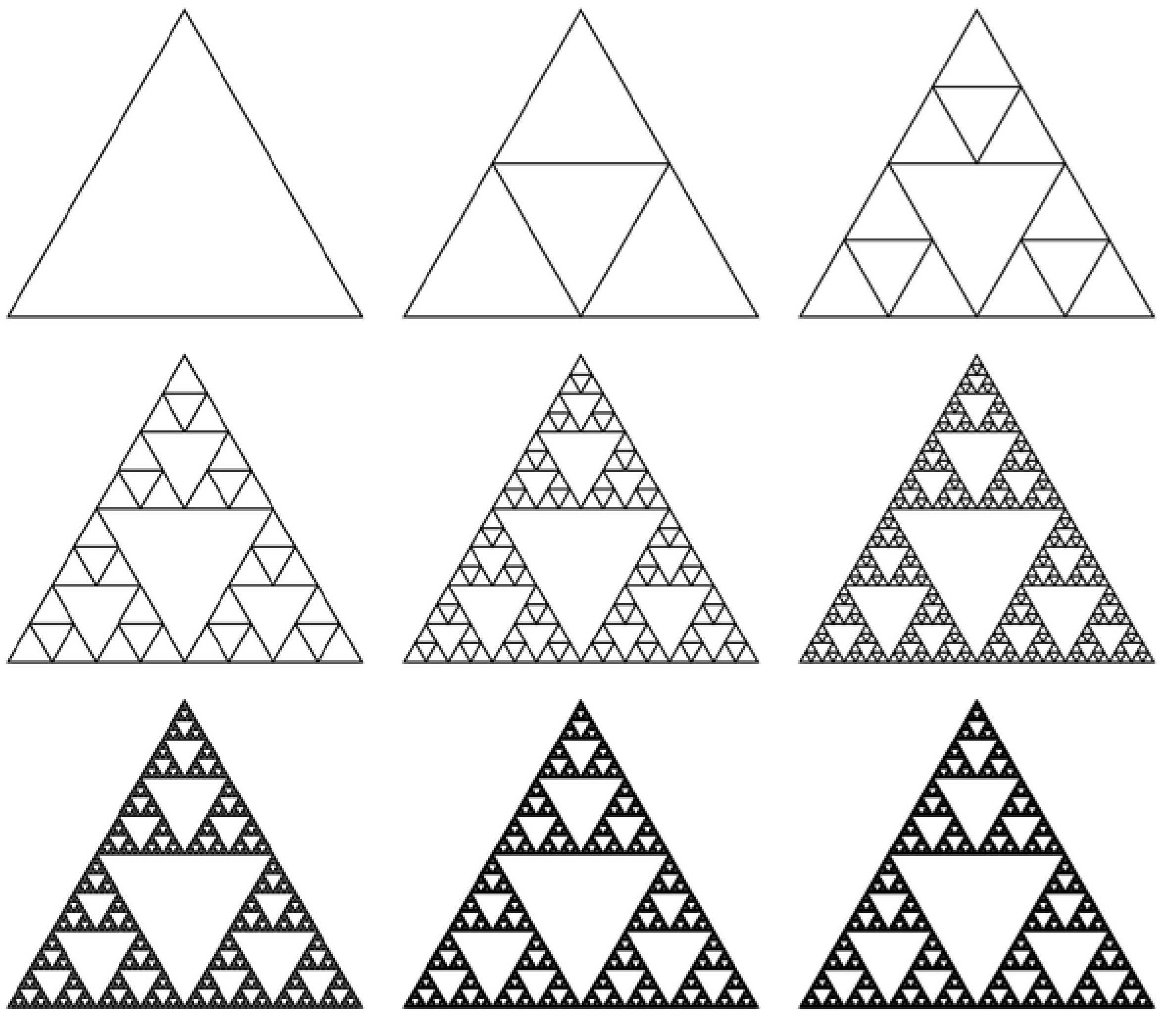
b. Write an efficient version of the `maxPath` function and call it `effMaxPath`. The two functions have the same signature, except that `effMaxPath` also receives an `int[][] memo` parameter. This array is used to record the costs of sub-paths already computed in order to avoid computing the cost of a sub-path more than once. This is an example of a general technique called *memoization*.

4. **Sierpinski triangles**

Consider the following recursive construction of the *Sierpinski triangle*:

a. Draw an equilateral triangle.
b. Subdivide it into four smaller congruent equilateral triangles.
c. Repeat step *b* with all the triangles except for the middle one.

Here is an illustration of the first 9 levels of this process:



Write a function with the signature `public static void sierpinski (int n)`. The function should use `StdDraw` to draw a *Sierpinski triangle* of *n* levels. You will want to use an auxiliary function for the actual recursion. You will need to decide what parameters in addition to n the auxiliary function will need to receive. You are free to choose any coordinates for the vertices of the outermost (largest) triangle, as long as it is equilateral.

You may find the following tip useful: If $P_1 = (x_1, y_1), P_2 = (x_2, y_2)$ are two points in the plane, then any point P_3 lying on the line segment connecting P_1 to P_2 can be expressed as $P_3 = t \cdot P_1 + (1-t) \cdot P_2$ where $0 \leq t \leq 1$. That is, we may write $P_3 = (tx_1 + (1-t)x_2, ty_1 + (1-t)y_2)$. In the case of the Sierpinski triangle we need the midpoint of the line segment, which corresponds to the case $t = \frac{1}{2}$.

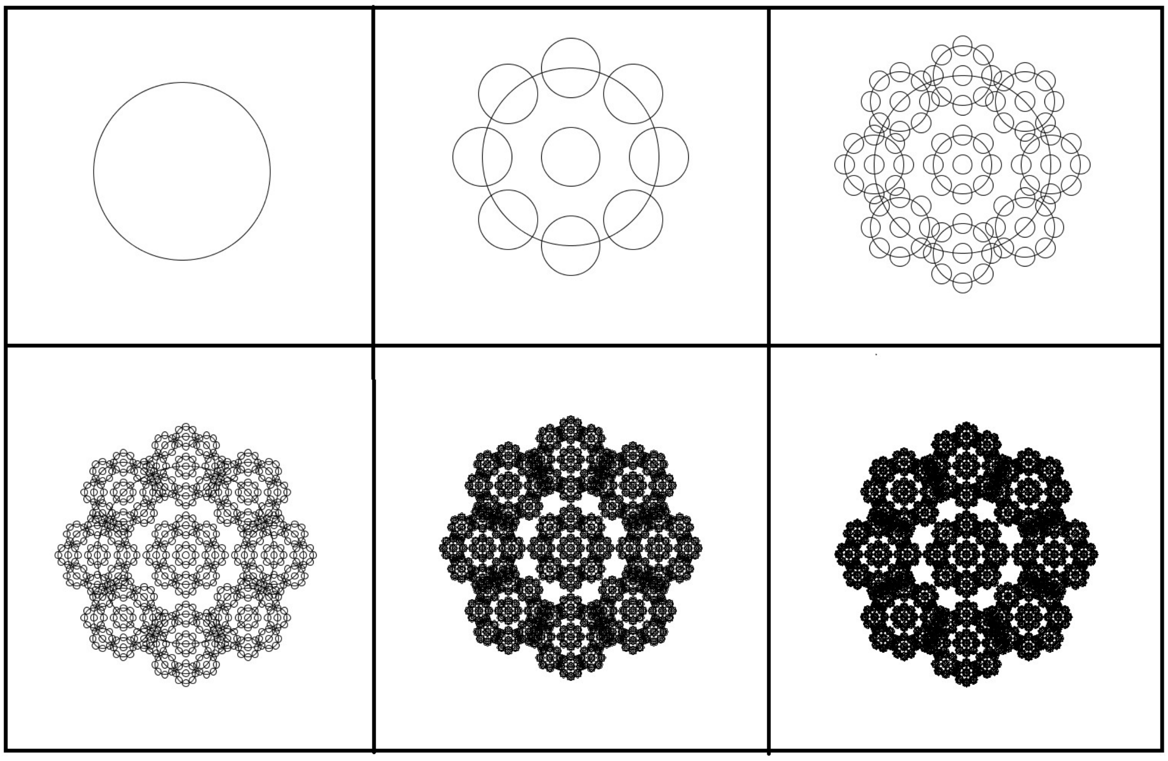
Use StdDraw.line() in order to draw a line.

5. **Banach Curve**

The so-called *Banach Curve* can be generated as follows:

a. Draw a circle.
b. Draw 9 new circles scaled down in size by a factor of 3 . One of these circles will have the same center as the original circle. The centers of the remaining 8 circles need to be equally spaced along the circumference of the original circle.
c. Repeat step *b* with each of the new circles.

Here is an illustration of the first 6 levels of this process:



Write a function with the signature `public static void banach (int n)` that draws a *Banach curve* of *n* levels. Use the `StdDraw.circle()` function to draw a circle.

You may find the following tip useful: If $P = (x, y)$ is a point in the plane, then the circle of radius r with center at P is the set of all points of the form $(x + r * \cos(t), y + r * \sin(t))$ where $0 \leq t \leq 2\pi$.

You will want to use an auxiliary function for the actual recursion. In addition to n the auxiliary function will need to receive the coordinates of the center of a circle as well as its radius. You may use the tip above in order to calculate the coordinates of the centers of the circles at the next level.

Submission: Before submitting any program, take some time to inspect your code, and make sure that it is written according to our **Java Coding Style Guidelines**. Also, make sure that each program starts with the program header described in the **Homework Submission Guidelines**. Any deviations from these guidelines will result in points penalty.

Submit the following files only:

- `Multiply.java` - containing the `multiply` function, or functions.
- `GCD.java` - containing the `gcd` function.
- `MaxPath.java` - containing the `maxPath` and `effMaxPath` functions.
- `Sierpinski.java` - containing the `sierpinski` function.
- `Banach.java` - containing the `banach` function.

Each program must include a `main` function for testing purposes. The `main` function needs to get the relevant inputs as command line arguments.

Deadline: Submit your assignment no later than Sunday, January 22, 23:55.