Homework 7

This homework focuses on recursion. The five problems described below must be solved recursively. **Non recursive solutions will not be accepted, even if they are correct.**

1. Calculating the GCD (10 points)

The *Greatest Common Divisor* (GCD) of two positive integers *x* and *y* is the largest positive integer that divides both *x* and *y*. For example:

gcd(4,6) = 2
gcd(54,24) = 6
gcd(56, 42) = 14

It turns out that the *gcd* function can be evaluated recursively, as follows:

$$gcd(x,y) = \begin{cases} x & \text{if } x = y \\ gcd(x-y,y) & \text{if } x > y \\ gcd(x,y-x) & \text{if } x < y \end{cases}$$

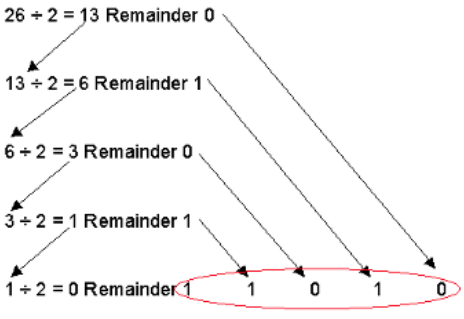
Select several small *x* and *y* values, go through the steps of the recursive evaluation process described above, and convince yourself that the process converges and computes the GCD of *x* and *y*. Then, find the skeleton of the `gcd(int x, int y)` function in the file `GCD.java`, and implement it recursively according to the above definition.

2. Base conversions (15 points)

Lecture 2-2 (slides 17 – 27) described how numbers can be represented using decimal (base 10) digits, and binary (base 2) digits. In slides 25 – 26 we showed how a decimal representation can be converted into a binary representation, using the following algorithm:

$26_{dec} = ?_{bin}$

Algorithm:



$26_{dec} = 11010_{bin}$

In each step of this algorithm we extract a binary digit (`0` or `1`) by dividing the given number by the binary base = 2. It turns out that this algorithm can be generalized, as follows.

If we wish to find the representation of a decimal number according to *any* base, we use the same algorithm, dividing the number repetitively by that base. For example, what is the representation of 26 in base 6 (which uses the digits `0`, `1`, `2`, `3`, `4`, and `5`)? If you will execute the above algorithm with 6 instead of 2, you should get that 26 in base 6 is 42 (let's double check: $4 \cdot 6^1 + 2 \cdot 6^0 = 26$. Looks like the algorithm is working). At this stage we suggest that you practice some base conversions by hand, using the above algorithm. Check your calculations either manually, as we did above, or using this calculator.

The function `String convert(int x, int base)` returns a string containing the digit-representation of `x` according to the given `base`. Assume that $2 \leq \text{base} \leq 10$. For example, `convert(26,2)` returns `"11010"`, `convert(234,3)` returns `"22200"`, and `convert(767,10)` returns `"767"`. Find the function skeleton in the file `Bases.java`, and implement it recursively.

3. Array processing (15 points)

Many array processing tasks can be implemented recursively. Here is an example of a recursive strategy that computes the sum of an array elements (using informal notation):

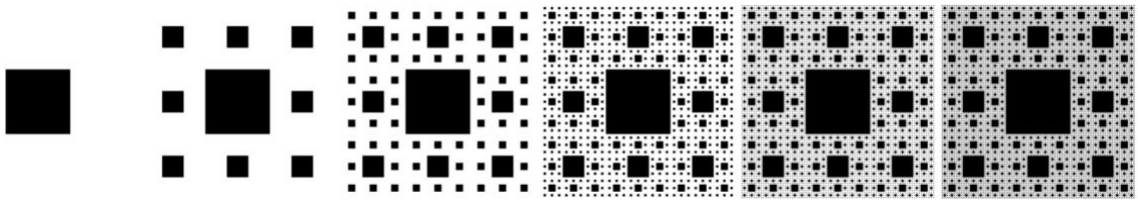
```
sumArr({1, 2, 3, 4}) = sumArr({1, 2, 3}) + 4
sumArr({1, 2, 3}) = sumArr({1, 2}) + 3
```

Etc.

In order to implement this recursive calculation for any given array, we use two functions: The public function `int sumArr(int[] arr)`, and the private function int `sumArr(int[] arr, int n)`, which computes the sum of the given array up to, and including, element `n`. The public function serves as the "driver function", also known as the "auxiliary function", and the private function does the actual work. Find the skeletons of both functions in the file `SumArray.java`, and implement them.

4. Sierpinski's Carpet (20 points)

A "Sierpinski Carpet" is a fractal that can be created according to the following pattern (from left to right):



The drawing rule is as follows:

a. Start with an empty square.

b. Divide it into 9 equal smaller squares.

c. Fill in the middle square, and repeat step b on the remaining squares.

The void function `drawCarpet(int n)` draws a Sierpinski Carpet of depth `n`. Find the function skeleton in the file `FractalDraw.java`, and implement it.

Implementation tips:

1. The general strategy of your drawing solution should be similar to the strategy used in the solution of the `drawH` fractal presented in lecture 8-1.

2. To draw filled squares, use the `StdDraw.filledSquare` function.

3. In each step of the recursion you have to do two basic things: draw a filled square, and make 8 recursive calls, to draw the smaller squares.

4. Before starting to write any code, make sure that you understand how to calculate the coordinates of the centers of the 8 smaller squares, and the length of the square's side. Use pen and paper! Give the keyboard some rest!

5. Note that trying to draw such fractals with depths greater than 10 may require a considerable computation time. If you want to kill the drawing process, close the drawing window, or type `CONTROL-C` in the console.

6. For your convenience, here is the API of `StdDraw.filledSquare`:

```
// Draws a filled square of the specified size, centered at (x, y).
public static void filledSquare(double x, double y, double halfLength)
Parameters:
x - the x-coordinate of the center of the square
y - the y-coordinate of the center of the square
halfLength - one half the length of the square's length
```

5. Maximal Value Path (40 points)

Given an $n \times m$ 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 $(n-1, m-1)$
2. At each step, move either one cell to the right, or one cell down.

For example, consider the following 3 × 3 array, and three possible paths in it:

3	4	5
2	2	0
1	0	1

The data

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

one path

value = 13

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

another path

value = 10

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

yet another path

value = 10

The *value* of a path is defined as the sum of all the values along the way. The maximal path is the path that has the maximal value among all the paths. The task is to find the value of the maximal path in a given array.

Here is one way to solve the problem. Suppose we want to compute the value of the maximal sub-path starting at some location (i,j) and ending at the bottom-right corner. As with any other location, we can go either one step down, or one step to the right. With that in mind, the value of the maximal path starting at location (i,j) can be computed by adding the array value

at location (i,j) to the maximum of two values: The value of the maximal path starting at location $(i+1, j)$, and the value of the maximal path starting at location $(i, j+1)$. This solution is quite beautiful, but wasteful (do you see why?). The good news is that it can be improved. Read on.

Part A (20 points)

Write a function `int maxVal(int[][] arr)` that computes the value of the maximal path in the given 2D array. Use the recursive strategy described above.

Part B (20 points)

Write a function `int effMaxVal(int[][] arr)` that computes the value of the maximal path in the given 2D array, efficiently.

The efficient implementation is quite similar to the basic implementation. The big difference is that the efficient version creates a a 2-dimensional array which is used to record the values of the sub-paths that were already computed. This is done in order to avoid computing the cost of a sub-path more than once. This is an example the *memoization* technique mentioned in lecture 8-1 (in the Fibonacci example). This technique will be discussed further in the recitations, so stay tuned.

Find the function skeletons in the file `MaxVal.java`, and implement them.

Submission

Zip the following files into the file `hw7.zip`:

- `GCD.java`
- `Bases.java`
- `SumArray.java`
- `FractalDraw.java`
- `MaxValPath.java`

Deadline: Submit Homework 7 no later than Sunday, December 27, 2020, 23:55. You are welcome to submit earlier.