Homework 7

The purpose of this assignment is gaining practice in recursive programming. Therefore, in each exercise, you must provide a **recursive** solution; other solutions will not be accepted. Your implementation may or may not include private functions, as you see fit.

1. Golden ratio

Write a class called `GoldenRatio` that takes an integer input `N` and computes an approximation to the golden ratio using the following recursive formula:

$$f(n) = \begin{cases} 1, if\ n = 0 \\ 1 + 1/f(n-1), if\ n > 0 \end{cases}$$

You are required to implement **two** functions:

- `goldenRec(n)` that calculates the golden ratio using recursion.
- `goldenIter(n)` that calculates the golden ratio using iteration.

A `main`, which is already supplied, will print the results of both the recursion and the iteration.

2. Permutations

A permutation of `n` elements is one of the n! possible orderings of the elements.
Write a class called `Permutations` that takes an integer command-line argument `n` and prints all n! permutations of the first `n` letters of the alphabet, starting at `a` (assume that `n` is no greater than 26).

For example, when `n = 3` you should get the following output (but do not worry about the order in which you enumerate them):

```
> java Permutations 3
bca cba cab acb bac abc
```

Hint: Before you start thinking of the code, try to write on a piece of paper all the permutations of "`abcd`".

3. First TTH

Probabilities can sometimes be calculated using recursion. For example, the probability of winning at least one coin flip out of ten coin flips (*Q(10)*) equals the probability to win the first flip (*Q(1)*), plus the probability to lose the first flip (*1-Q(1)*) and then to win at least one of nine coin flips (*Q(9)*). Importantly, the probability to win just a single coin flip is exactly ½. Therefore,

$$Q(n) = Q(1) + (1-Q(1)) * Q(n-1) = ½ + ½\ Q(n-1).$$

In that spirit, suppose we flip a coin multiple times, resulting in a sequence of heads (H) and tails (T). We stop flipping the coin when we get the sequence TTH (tail-tail-head). What is the probability that we have flipped exactly *n* coins?

For example, these are sequences of nine coin flips that end with the first TTH:

HHTHHH**TTH**, THHHHT**TTH**, HTHTHH**TTH**, THTTTT**TTH**

These are not sequences of nine coin flips that end with TTH, either because TTH already appears before the end, or because they do not end with TTH:

TTT**TTH**TTH, HTHHTHTTT, HTH**TTH**TTH, TTTTTTTTT

Write a class called `TTH` that takes a single command-line argument *n* and prints the probability *P(n)* of flipping exactly *n* coins and get TTH on the last three flips.

Important: the calculation of *P(n)* must be done recursively. To do so, note that a sequence of *n* coin flips that ends with the first TTH must fulfill one of the following:

1. Start with H followed by a sequence of *n-1* flips that ends with the first TTH.
2. Start with TH followed by a sequence of *n-2* flips that ends with the first TTH.
3. Start with *n-1* Ts, followed by an H.

This can also be represented by the following formula:

$$P(n) = Prob(H) * P(n-1) + Prob(TH) * P(n-2) + Prob(n-1\ Ts) * Prob(H)$$

$$= \tfrac{1}{2} P(n-1) + \tfrac{1}{4} P(n-2) + 1/2^n$$

where *Prob(X)* is the probability for *X* to occur.

Use this formula to implement a recursive function `calcP(n)` that calculates *P(n)*.

Examples:

```
> java TTH 2

0.0

> java TTH 3

0.125

> java TTH 10

0.052734375

> java TTH 20

0.006450653076171875
```

5. Memoization

The calculation we did above for *P(n)* was wasteful: *P(5)* is calculated using *P(4)* and *P(3)*, and *P(4)* also uses *P(3)*, but it re-calculates it.

To avoid this wastefulness, we can use *memoization*. This technique is very powerful: the time to calculate *P(20)*, for example, is 100-fold longer without memoization.

To use memoization, we initialize a memory array `double[] memory` that saves the results of *P(n)* values that we have already calculated. This array is passed to the recursive function. Then, when attempting to calculate a *P(n)* value inside the function `calcPmem`, we:

1. If *P(n)* is not in the memory, we calculate it and put it in the array;
2. Return the value of *P(n)* from the memory.

Note that *P(n)>0* for *n>2*, so we can initialize the memory array to zeros at the beginning and then fill in the values of *P(n)* as we go along.

Add another function `calcPmem(int n, double[] memory)` that calculates *P(n)* while using memoization.

5. Fractal Tree

A random fractal tree can be generated using the following fractal rule:

a. Randomly choose a branch length.
b. Adjust branch color and width.
c. Draw a branch.
d. Repeat to draw two branches at the end of the drawn branch with two random angles.
e. Stop when you reach a specified recursion depth.

You can see an example of a full tree (recursion depth of 14) at the bottom of this page and watch an animation of the tree been drawn in this link.

Write a function in the class `FractalTree` that draws a random fractal tree with given level *n*, using the signature

`public static void drawBranch(int level, double x0, double y0, double length, double angle)`.

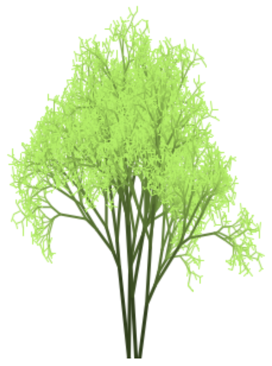
A corresponding main function, as well as some constants, are already given.

Implementation details:

- The length of the line you draw at each call to `drawBranch` should be a random number between 0 and `length`.
- The maximum length of the branch (`length`) starts at 100 for the trunk and is reduced by a factor of 0.8 at each level of the tree.
- The angle of the branch you draw at each call should be exactly `angle`.
- The color of the branch should be adjusted at each call such that the red, green, and blue values are a weighted average of the color of the root (`rootColor`) and the color of the leaves (`leafColor`). The weight is the ratio between the current recursion depth (`level`) and the maximal recursion depth (`recursionDepth`). For example, the red value should be

`rootRed * w + leafRed * (1-w)` where `w = level / recursionDepth`.

- The width should be adjusted to be `trunkWidth * w` (where w is the same as above), but not lower than `minWidth`.
- The angles of the next branches should be randomly chosen:
  - For one branch, the angle should be between the current angle (`angle`) plus a random value between 0 and `π/4 * m`, where `m=6/level`.
  - For the other branch, the angle should be the current angle minus a random value between 0 and `π/4 * m`, where `m=6/level`.
- The root of the tree should have the highest `level`. Make sure to decrement `level`, and to halt when it reaches zero.



Submission

Before submitting your solution, 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**. Both documents can be found in the Moodle site, it is your responsibility to read them. Any deviations from these guidelines will result in points penalty. Submit the following files:

- ➢ `GoldenRatio.java`
- ➢ `Permutations.java`
- ➢ `TTH.java`
- ➢ `FractalTree.java`

Deadline: Submit Homework 7 no later than January 12, 2020, 23:55. You are welcome to submit earlier.