Exercise 1

Consider the class

`com.andreamazzon.session3.encapsulation.lazyinitialization.LinearCongruentialGenerator`

of the Java course project. We saw that due to overflows, the number

```
long congruence = (a * randomNumbers[indexOfInteger] + c) % modulus
```

can be negative, and to fix this problem we incremented it by `modulus`. However, even if this still produces a (pseudo) random natural number smaller than `modulus`, it is not mathematically completely correct. To better understand the problem, consider 4 bits, giving 16 integers from -8 to $+7$. Take the modulus equal to 3. Here 8 % $3 = 2$. But 8 corresponds to -8 due to the overflow, (-8) % $3 = -(8$ % $3) = -2$ and $-2 + 3 = 1$. So our method does not give what we would expect. Indeed, the operations *overflow* and % do not commute, and one has first to fix the overflow and then to apply the operation %.
Think about a good way to do this in the case when

$$\texttt{Long.MAX\_VALUE} < a\cdot\texttt{modulus} + \texttt{c} < 2\cdot\ \texttt{Long.MAX\_VALUE},$$

and correct the method `generate()` written in `LinearCongruentialGenerator` accordingly, changing the values of `modulus` and `a` so that the above condition holds. You can first copy and paste the class `LinearCongruentialGenerator` in your exercise project, and test it as in

`com.andreamazzon.session3.encapsulation.lazyinitialization.PseudoRandomNumbersTesting`.

Check, for example by Mathematica, that the natural number you should get if there was no overflow issue (i.e., if you could represent a number of whatever size) is indeed

```
(a * previousRandomNumber + c) % modulus
```

where `previousRandomNumber` is the previous number in the list.
Note: possible values are `a = 6553590L`, `modulus = (long) Math.pow(2, 48)`. Choosing `seed = 2814749763100L` in the test class, the value of the first generated pseudo random natural number after the seed must be `253301221688051`.

Exercise 2

This exercise gives an example of the computations of prices of options via Monte-Carlo. The idea is that, exploiting the results you have seen in the lecture, one can approximate the price of an option as

$$\frac{1}{n}\sum_{i=1}^{n} p_i,$$

where $p_i = p(\omega_i)$, $i = 1, \dots, n$ for large n, are different realizations of the payoff of the option.
Write a class `DigitalOption` implementing the interface

`com.andreamazzon.handout2.EuropeanTypeOptionMonteCarlo`.

There you find two methods, i.e.,

```
getPayoff(StochasticProcessSimulatorInterface underlyingProcess)
```

and

```
getPrice(StochasticProcessSimulatorInterface underlyingProcess),
```

whose description you find in the interface, that return the realizations of the payoff at maturity time and the Monte-Carlo price (i.e., the average of the payoff realizations), respectively, of an European option whose underlying is described by the object `underlyingProcess`.

Note that `StochasticProcessSimulatorInterface` is the interface you can find in

`com.andreamazzon.handout0`.

You have here to implement these two methods for a digital option, i.e., an option that for an underlying S at maturity time i has payoff 1 if $S(i) - K > 0$ and 0 vice versa, where $K > 0$ is a given strike.

Test your code by creating an object of this class with maturity 7 and strike 100, and let it call the method `getPrice` by giving it an object of type

`com.andreamazzon.handout0.BinomialModelSimulator`,

with initial value 100, $u = 1.5$, $d = 0.5$, interest rate 0, final time 7, number of simulations 100000 and seed of your choice. The price you get should be approximately equal to 0.22.

Hint: what you can do to implement `getPayoff` is basically to see which method(s) of the interface `StochasticProcessSimulatorInterface` might help you, and make `underlyingProcess` call it/them. Remember that, due to polymorphism, `underlyingProcess` can be of whatever class implementing `StochasticProcessSimulatorInterface`. The method `getPrice` might rely on the call of `getPayoff`.