Solution to exercise 1

In this exercise, you want to take care of possible overflows in the computation of

$$\texttt{r = (a * randomNumbers[indexOfInteger] + c) \% modulus}$$

under the assumption

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

If the operation `a * randomNumbers[indexOfInteger] + c` produces an overflow, i.e., if the result is bigger than `Long.MAX_VALUE`, the value produced in Java is negative (note that a multiple overflow is prevented by the assumption $a{\cdot}\texttt{modulus}{+}c < 2{\cdot}\ \texttt{Long.MAX\_VALUE}$).
In this case, two values play an essential role: the true mathematical value of `a * randomNumbers[indexOfInteger] + c` and the number you get in your program. The first one can be written as

$$\texttt{Long.MAX\_VALUE} + \texttt{valueOverflow} = -\ \texttt{Long.MIN\_VALUE} + \texttt{valueOverflow}\ -1,$$

where `valueOverflow` is the size of the overflow got in the operation, whereas the number produced by Java is

$$\texttt{Long.MIN\_VALUE} + \texttt{valueOverflow}\ -1.$$

The goal of the exercise is to find a way to get the natural number

$$\texttt{r} = -\ \texttt{Long.MIN\_VALUE} + \texttt{valueOverflow}\ -1\ \ \texttt{\% modulus},$$

only looking at the observed number

$$\texttt{observedNumber} = \texttt{Long.MIN\_VALUE} + \texttt{valueOverflow}\ -1. \tag{1}$$

By the distributive property of the `%` operation, we have

$$\begin{aligned}
&-\ \texttt{Long.MIN_VALUE} + \texttt{valueOverflow}\ -1\ \ \texttt{\% modulus}\\
&= \big(\ (-\texttt{Long.MIN_VALUE}\ \ \texttt{\% modulus}) + (\texttt{valueOverflow}\ -1\ \ \texttt{\% modulus})\big)\,\texttt{\% modulus}\\
&= (\texttt{modulusOfMinusMinValue}+\ \texttt{modulusOverflowMinusOne})\ \ \texttt{\% modulus},
\end{aligned}$$

where

$$\texttt{modulusOfMinusMinValue} = \texttt{-Long.MIN\_VALUE}\ \ \texttt{\% modulus}$$

and

$$\texttt{modulusOverflowMinusOne} = \texttt{valueOverflow}\ \ -1\ \ \texttt{\% modulus}.$$

Note that `modulusOfMinusMinValue + modulusOverflowMinusOne` is positive and less then `Long.MAX_VALUE` if $2\cdot$ `modulus` $<$ `Long.MAX_VALUE`, so it is not affected by overflows, and this is the right correction to the overflow that we have to perform before applying `%`.
Now it only remains to get `valueOverflow` from the observed number (1), i.e.

$$\texttt{valueOverflow} = \texttt{observedNumber}\ -\ \texttt{Long.MIN\_VALUE}\ {+}1.$$

Note that in the case when `-Long.MIN_VALUE % modulus`$=0$, that is for example the case of the `LinearCongruentialGenerator` class, it is enough to compute `valueOverflow` -1 `% modulus`. However, in this case one can also correct the overflow after the `%`, just adding `modulus` to the result, as we did in `LinearCongruentialGenerator`.
Indeed, we have

$$\texttt{Long.MIN\_VALUE} = (-k)\cdot\texttt{modulus} = (-k+1)\cdot\texttt{modulus} - \texttt{modulus},$$

where $k \geq 1$ is a natural number, so calling $\texttt{modulusOverflowMinusOne} = \texttt{valueOverflow}\ -1\ \texttt{\%}\ \texttt{modulus}$ we have

$$\texttt{observedNumber \% modulus} = (-\texttt{modulus} + \texttt{modulusOverflowMinusOne})\ \texttt{\% modulus}.$$

The latter is a negative number, so it is the value returned by Java when we ask it to compute `observedNumber % modulus`. So it is enough to add `modulus` to the result to obtain `modulusOverflowMinusOne`, which is the number we want, as observed above.