2 Fermat Primality Test (Haskell, 8 points)

In this task, for a given natural number p

1. generate a sequence of pseudorandom natural numbers a such that $2 \leq a < p - 1$,
2. test whether p is prime by checking whether the following equation holds for each generated pseudorandom number a

$$a^{p-1} \equiv 1 \ (\texttt{mod}\ \ p). \tag{1}$$

If (1) holds for all numbers a, it is highly probable that p is prime. This probabilistic primality test is known as the *Fermat Primality Test*. Note, the *Carmichael numbers*, which are composite yet pass the test for all a relatively prime to the respective number, are avoided when testing your implementation of this task.

2.1 Pseudo-random Number Generation

To generate pseudorandom numbers in a given interval, use the *Linear Congruential Generator (LCG)*

$$x_{n+1} = (Ax_n + C) \ \texttt{mod}\ \ M, \tag{2}$$

where A, C and M are constants. This equation generates the next pseudorandom number x_{n+1} from the previous x_n. The number x_0 is the seed.

The number b drawn from (2) can be transformed to the interval $b^{\text{lower}} \leq b' < b^{\text{upper}}$ as

$$b' = (b \ \texttt{mod}\ \ (b^{\text{upper}} - b^{\text{lower}})) + b^{\text{lower}}. \tag{3}$$

2.2 Haskell Assignment

Your task is to implement the Fermat Primality Test in Haskell. There are two parts to this task: first, implement the LCG, and then the test itself.

The LCG generator is represented as

```
data LCG = LCG  Int Int Int Int deriving Show
```

where the four `Int`s in the `LCG` 4-tuple are A, x_n, C, and M w.r.t. (2). Implement the function

```
generate_range :: Int -> Int -> State LCG Int
```

which is used to sample random integers $b^{\text{lower}} \leq b' < b^{\text{upper}}$ where the lower and upper bounds are set in the first and second function input, respectively. The state of the LCG is kept using the `State` monad. So start your code with

```
import Control.Monad.State
```

Note that it is desirable to follow (2) and (3) as closely as possible, since the tasks are tested with constant seed and thus are considered deterministic. The function is used as follows.

```
> runState (generate_range 1 100) (LCG 513 1 1 1024)
(20, LCG 513 514 1 1024)
```

Implement the primality test in the function

```
primality :: Int -> Int -> State LCG Bool
```

where the first input is the potential prime and the second is the number of repetitions of the test (i.e., the number of generated pseudorandom numbers). The function is used as follows:

```
> evalState (primality 17 1000) (LCG 513 1 1 1024)
True
> evalState (primality 42 1000) (LCG 513 1 1 1024)
False
> evalState (primality 9 0) (LCG 513 1 1 1024)
True
```

Hint: To prevent overflow of `Int`, compute $a^{p-1} \mod p$ sequentially using the identity $a^{k+1} \mod p = a \cdot (a^k \mod p) \mod p$, i.e., sequentially multiplying by a and applying modulo to each partial result.