Ejercicios extra para la práctica 1

Ejercicio [`esPrimo`]. Define una función `esPrimo` que determine si un entero positivo es un número primo. Asegúrate de que funcione para los tipos `Int` e `Integer`. Si el argumento no es un entero positivo la función debe señalar el error con un mensaje apropiado.

```
> esPrimo (5::Int)
True

> esPrimo (5::Integer)
True

> esPrimo 1
False

> esPrimo 6
False

> esPrimo 0
*** Exception: esPrimo: argumento negativo o cero
```

Ejercicio [`cocienteYResto`]. Define una función `cocienteYResto` que dados dos naturales devuelva el cociente y resto de dividir el primero entre el segundo. Por ejemplo:

```
> cocienteYResto 25 4
(6,1)
```

Ten en cuenta las siguientes **restricciones**:

1. La función **no** puede utilizar las funciones `div` ni `mod`, ni ninguna otra operación de división predefinida. Solo se pueden utilizar sumas y restas.
2. La función debe funcionar para los tipos `Int` e `Integer`:

   ```
   > cocienteYResto (25::Int) (4::Int)
   (6,1)
   it :: (Int, Int)

   > cocienteYResto (25::Integer) (4::Integer)
   (6,1)
   it :: (Integer, Integer)
   ```

3. Si el divisor es 0, la función debe señalar el error con el mensaje apropiado:

   ```
   > cocienteYResto 25 0
   *** Exception: división por 0
   ```

4. Si alguno de los argumentos es negativo, la función debe señalar el error con el mensaje apropiado:

   ```
   > cocienteYResto (-6) 2
   *** Exception: argumentos negativos

   > cocienteYResto 6 (-2)
   *** Exception: argumentos negativos
   ```

Define además una propiedad QuickCheck `prop_cocienteYResto_OK` que compruebe que tu función es correcta:

```
> quickCheck prop_cocienteYResto_OK
+++ OK, passed 100 tests.
it :: ()
```

Ejercicio [`libreDeCuadrados`]. Un entero positivo n se dice que está libre de cuadrados si no es divisible por ningún cuadrado perfecto —es decir, el cuadrado de un entero— excepto el 1. Esto implica que en la descomposición de n en factores primos no se repite ningún factor.

Define una función `libreDeCuadrados :: Integer -> Bool` que determine si un entero positivo está libre de cuadrados. Si el argumento no es un entero positivo la función debe mostrar un mensaje de error apropiado.

```
> libreDeCuadrados 1
True

> libreDeCuadrados 2
True

> libreDeCuadrados 24
False

> libreDeCuadrados 25
False

> libreDeCuadrados 26
True

> libreDeCuadrados (-1)
*** Exception: libreDeCuadrados: argumento cero o negativo
```

Ejercicio [`raizEntera`]. Se define la raíz cuadrada entera de un entero no negativo n como el mayor entero r tal que $r^2 \leq n$.

Define una función `raizEntera :: Integer -> Integer` que dado un entero no negativo n devuelva su raíz entera. La función debe implementar un algoritmo que compruebe secuencialmente todos los posibles candidatos a raíz $0, 1, 2, 3, \ldots$ hasta encontrar el valor adecuado. Si el argumento es un entero negativo la función debe mostrar un mensaje de error apropiado.

```
> raizEntera 25
5

> raizEntera 24
4

> raizEntera 35
5

> raizEntera 36
6

> raizEntera (-1)
*** Exception: raizEntera: argumento negativo
```

Comprueba que la función `raizEntera` está correctamente definida mediante la siguiente propiedad QuickCheck:

```
prop_raizEntera_OK n =
   n >= 0 ==> truncate (sqrt (fromIntegral n)) == raizEntera n
```

Supongamos que queremos calcular la raíz entera de 712. El algoritmo anterior comprobaría todos los valores secuencialmente: $0, 1, 2, 3\ldots$ hasta alcanzar el valor deseado. Se puede calcular la raíz entera de manera más rápida sin comprobar secuencialmente los candidatos. La idea consiste en trabajar con un **intervalo** donde sabemos que se encuentra la solución. Sabemos que la raíz entera debe pertenecer al intervalo $[0, 712]$. Tomamos ahora el punto medio del intervalo $\frac{712}{2} = 356$. Comprobamos que 356 no es la raíz, luego la raíz debe pertenecer a uno de los subintervalos $[0, 356]$ o $[356, 712]$. Dado que $356^2 > 712$, la raíz debe pertenecer al primer subintervalo, $[0, 356]$. Esto permite reducir a la mitad los candidatos en un solo paso. El proceso continúa dividiendo el intervalo que contiene la solución por la mitad hasta encontrar la raíz o un intervalo no reducible (formado por dos enteros consecutivos).

Define una función `raizEnteraRapida :: Integer -> Integer` que implemente el algoritmo anteriormente descrito. Si el argumento es un entero negativo la función debe mostrar un mensaje de error apropiado.

```
> raizEnteraRapida 5
2

> raizEnteraRapida 25
5

> raizEnteraRapida 24
4

> raizEnteraRapida 35
5

> raizEnteraRapida (-1)
*** Exception: raizEnteraRapida: argumento negativo
```

Compara la eficiencia de `raizEntera` y `raizEnteraRapida` realizando los siguientes cálculos:

```
> raizEntera (1234567^2)
1234567

> raizEntera (12345678^2)
12345678

> raizEnteraRapida (1234567^2)
1234567

> raizEnteraRapida (12345678^2)
12345678
```

Define una propiedad QuickCheck que compruebe que `RaizEntera` y `RaizEnteraRapida` devuelven el mismo resultado.

Ejercicio [`Harshad`]. Un número de Harshad es un entero positivo que es divisible por la suma de sus dígitos. Por ejemplo, 18 es un número de Harshad pues $1 + 8 = 9$ y 18 es divisible entre 9 ($18 \bmod 9 = 0$).

Define una función `sumaDigitos :: Integer -> Integer` que dado un natural devuelva la suma de sus dígitos. Si el argumento no es un natural la función debe señalar el error con un mensaje apropiado.

```
> sumaDigitos 5
5

> sumaDigitos 0
0

> sumaDigitos 18
9

> sumaDigitos 123456789
45
```

```
> sumaDigitos (-1)
*** Exception: sumaDigitos: argumento negativo
```

Define una función `harshad :: Integer -> Bool` que determine si un entero positivo es un número de Harshad. Si el argumento no es un entero positivo la función debe señalar el error con un mensaje apropiado.

```
> harshad 18
True

> harshad 19
False

> harshad 156
True

> harshad 1729
True

> harshad 0
*** Exception: harshad: argumento no positivo
```

Un número de Harshad es múltiple si el cociente de dividirlo entre la suma de sus dígitos es también un número de Harshad. Por ejemplo, 6804 es un número de Harshad múltiple, pues 6804 es un número de Harshad y, además, $6804 \div 18 = 378$, y 378 es también un número de Harshad.

Define una función `multipleHarshad :: Integer -> Bool` que determine si un entero positivo es un número de Harshad múltiple.

```
> harshadMultiple 7
True

> harshadMultiple 54
True

> harshadMultiple 55
False

> harshadMultiple 117
False

> harshadMultiple 144
False

> harshadMultiple 0
*** Exception: harshad: argumento no positivo
```

Un número de Harshad es n veces múltiple de Harshad si podemos aplicar n veces la definición de multiplicidad sobre los números de Harshad que vamos obteniendo al calcular los sucesivos cocientes. Por ejemplo, 6804 es un número 5 veces múltiple de Harshad:

- 6804 es de Harshad
- $378 = 6804 \div 18$ es de Harshad
- $21 = 378 \div 18$ es de Harshad
- $7 = 21 \div 3$ es de Harshad
- $1 = 7 \div 7$ es de Harshad

Define una función `vecesHarshad :: Integer -> Integer` que dado un entero positivo determine cuántas veces es múltiple de Harshad. Si el número no es de Harshad debe devolver 0.

```
> vecesHarshad 1
1
```

```
> vecesHarshad 5
2

> vecesHarshad 6804
5

> vecesHarshad 10080000
6

> vecesHarshad 2016502858579884466176
12

> vecesHarshad 1000
1

> vecesHarshad 15
0

> vecesHarshad 0
*** Exception: harshad: argumento no positivo
```

Bloem demostró que los números de la forma 1008×10^n son $n + 2$ veces múltiples de Harshad. Utiliza este resultado para definir una propiedad QuickCheck `prop_Bloem_Harshad_OK` para comprobar el correcto funcionamiento de `vecesHarshad`.

Ejercicio [`cerosDe`]. Define una función `cerosDe :: Integer -> Integer` que dado un entero devuelva el número de ceros en que acaba.

```
> cerosDe 0
1

> cerosDe 1
0

> cerosDe 10
1

> cerosDe 100
2

> cerosDe 10001
0

> cerosDe 12300000
5

> cerosDe (-300)
2
```

Define una propiedad QuickCheck que compruebe el funcionamiento de `cerosDe`:

```
prop_cerosDe_OK n m = ...
```

La propiedad toma 2 argumentos, `n` y `m`, donde `n` es un entero arbitrario y `m` es un entero entre 0 y 1000. Utiliza `n` y `m` para construir un entero que acabe en exactamente `m` ceros y comprueba que, en efecto, `cerosDe` devuelve `m` para tal entero.

Utiliza la función anterior para calcular en cuántos ceros acaban los factoriales de 10, 100, 1000 y 10000.

Ejercicio [`Fibonacci`]. Los números de Fibonacci se definen con la siguiente recurrencia:

$$Fib(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{si } n > 1 \end{cases}$$

Define una función `fib :: Integer -> Integer` que dado un natural devuelva su Fibonacci. Si el argumento no es un natural la función debe señalar el error con un mensaje apropiado.

```
> fib 5
5

> fib 7
13

> fib 9
34

> fib 15
610

> fib 32
2178309

> fib (-1)
*** Exception: fib: argumento negativo
```

Observa que el tiempo de cálculo de Fibonacci crece rápidamente en función del argumento. Dependiendo del procesador que utilices, puede llevar bastante tiempo calcular el Fibonacci de 30 o el de 40. Esto se debe a que la recurrencia de Fibonacci presenta recursión doble (el caso recursivo tiene 2 llamadas recursivas) y, además, se repite buena parte del cálculo (calcular el Fibonacci de 30 requiere calcular el Fibonacci de 29 y el de 28; y el Fibonacci de 29 vuelve a requerir el Fibonacci de 28, y así sucesivamente).

Define una función `llamadasFib` que dado un natural n calcule el número de llamadas a `fib` que deben realizarse para calcular el Fibonacci de n.

```
> llamadasFib 0
1

> llamadasFib 1
1

> llamadasFib 2
3

> llamadasFib 3
5

> llamadasFib 4
9

> llamadasFib 5
15
```

¿Cuántas llamadas son necesarias para calcular `fib 30`? ¿Y `fib 36`?

Podemos reducir drásticamente el número de llamadas para calcular el Fibonacci y evitar repetir trabajo si definimos Fibonacci mediante una función recursiva con 2 acumuladores:

```
fib' :: Integer -> Integer
fib' n = fibAc n 0 1
  where
    fibAc ...
```

El primer acumulador de `fibAc` representa el Finonacci de i y el segundo representa el Fibonacci de $i+1$. Los acumuladores se inicializan a los Fibonaccis de 0 y 1, respectivamente. Completa la definición de `fibAc`.

```
> fib' 0
0

> fib' 1
1

> fib' 5
5

> fib' 15
610

> fib' 30
832040

> fib' 60
1548008755920

> fib' 100
354224848179261915075

> fib' (-1)
*** Exception: fib': argumento negativo
```

Observa que con `fib'` podemos calcular Fibonaccis muy elevados en un tiempo despreciable. ¿Cuantas llamadas son necesarias para calcular `fib 30`? ¿Y `fib 36`?

Define una propiedad QuickCheck que compruebe que `fib` y `fib'` son equivalentes. Dado que el tiempo de cálculo de `fib` crece rápidamente, asegúrate de que la propiedad solo comprueba la equivalencia hasta el Fibonacci de 30.

Una forma más rápida aún de calcular los números de Fibonacci consiste en aplicar la fórmula de Binet:

$$Fib(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$$

donde φ es la razón áurea y se define como $\varphi = \frac{1+\sqrt{5}}{2}$.

Define una función `binet :: Integer -> Integer` que dado un natural devuelva su Fibonacci.

```
> binet 1
1

> binet 5
5

> binet 15
610

> binet 30
832040

> binet 60
```

```
1548008755920

> binet (-1)
*** Exception: binet: argumento negativo
```

Para definir `binet` tendrás que utilizar la función predefinida `round` que convierte un número flotante en entero mediante redondeo:

```
> round 1.1
1

> round 1.4
1

> round 1.5
2

> round 1.9
2
```

Observa que `fib'` y `binet` no son equivalentes para Fibonaccis suficientemente elevados:

```
> binet 200
280571172992509965361722520092440986124288

> fib' 200
280571172992510140037611932413038677189525

> fib' 200 - binet 200
174675889412320597691065237
```

La fórmula de Binet es correcta, pero asume que los cálculos se realizan en $\mathbb{R}$ de forma exacta. En la práctica, los cálculos se realizan con el tipo `Double` que tiene una precisión limitada. Por el contrario, el tipo `Integer` tiene una precisión ilimitada. No todo valor del tipo `Integer` se puede representar de forma exacta en `Double`:

```
> (fromIntegral (12345678901234567890::Integer))::Double
1.2345678901234567e19
```

Esto significa que `fib'` podrá calcular de forma exacta Fibonaccis muy elevados (limitado solo por los recursos de la máquina), pero `binet` fallará a partir de ciertos valores.

Define una propiedad QuickCheck que te ayude a determinar a partir de qué valor los resultados devueltos por `binet` son incorrectos.

Sugerencia: Define una propiedad QuickCheck para comprobar la equivalencia de `fib'` y `binet`. Esta propiedad fallará; utiliza los contraejemplos de QuickCheck para determinar el valor a partir del cual los cálculos de `binet` son incorrectos.