Relación de Ejercicios 1

Para realizar estos ejercicios, crea un nuevo fichero (con extensión hs) con identificador formado por tus iniciales de apellidos y nombre, seguido de Rel1 y seguido del ejercicio o ejercicios que contiene (ejemplo: RMR-Rel1-2-6.hs); añade al principio de tu fichero la siguiente cabecera, reemplazando los datos necesarios:

```
-------------------------------------------------------------------------------
-- Estructuras de Datos. 2º Curso. ETSI Informática. UMA
--
-- (completa y sustituye los siguientes datos)
-- Titulación: Grado en Ingeniería ............................ [Informática | del Software | de Computadores].
-- Alumno: APELLIDOS, NOMBRE
-- Fecha de entrega:  DIA | MES | AÑO
--
-- Relación de Ejercicios 1. Ejercicios resueltos: ..........
--
-------------------------------------------------------------------------------
import Test.QuickCheck
```

1. Tres enteros positivos *x*, *y*, *z* constituyen una terna pitagórica si $x^2+y^2=z^2$, es decir, si son los lados de un triángulo rectángulo.

a) Define la función

```
esTerna :: Integer -> Integer -> Integer -> Bool
```

que compruebe si tres valores forman una terna pitagórica. Por ejemplo:

```
Main> esTerna 3 4 5          Main> esTerna 3 4 6
True                         False
```

b) Es fácil demostrar que para cualesquiera *x* e *y* enteros positivos con $x>y$, la terna $(x^2-y^2, 2xy, x^2+y^2)$ es pitagórica. Usando esto, escribe una función `terna` que tome dos parámetros y devuelva una terna pitagórica. Por ejemplo:

```
Main> terna 3 1
(8,6,10)
Main> esTerna 8 6 10
True
```

c) Lee y entiende la siguiente propiedad, para comprobar que todas las ternas generadas por la función `terna` son pitagóricas:

```
p_ternas x y = x>0 && y>0 && x>y ==> esTerna l1 l2 h
  where
    (l1,l2,h) = terna x y
```

d) Comprueba esta propiedad usando *QuickCheck* (recuerda importar `Test.QuickCheck` al principio de tu programa y copiar la propiedad en tu fichero). Verás que la respuesta es parecida a:

```
Main> quickCheck p_ternas
*** Gave up! Passed only 62 tests
```

lo que indica que, aunque sólo se generaron 62 casos de pruebas con las condiciones precisas, todos estos los casos pasaron la prueba.

2. Define una función polimórfica

```
intercambia :: (a,b) -> (b,a)
```

que intercambie de posición los datos de la tupla:

```
Main> intercambia (1,True)          Main> intercambia ('X','Y')
(True,1)                            ('Y','X')
```

3. Este ejercicio versa sobre ordenación de tuplas.

a) Define una función sobrecargada para tipos con orden

```
ordena2 :: Ord a => (a,a) -> (a,a)
```

que tome una tupla con dos valores del mismo tipo y la devuelva ordenada de menor a mayor:

```
Main> ordena2 (10,3)                Main> ordena2 ('a','z')
(3,10)                              ('a','z')
```

Copia en tu fichero las siguientes propiedades relativas a la función `ordena2`:

```
p1_ordena2 x y = enOrden (ordena2 (x,y))
  where enOrden (x,y) = x<=y

p2_ordena2 x y = mismosElementos (x,y) (ordena2 (x,y))
   where
     mismosElementos (x,y) (z,v) = (x==z && y==v) || (x==v && y==z)
```

(o, alternativamente, `(x,y)==(z,v)||(x,y)==(v,z)`)

Entiende lo que cada una significa, y compruébalas usando *QuickCheck*.

b) Define una función sobrecargada para tipos con orden

```
ordena3 :: Ord a => (a,a,a) -> (a,a,a)
```

que tome una tupla con tres valores del mismo tipo y la devuelva ordenada, con los elementos de menor a mayor:

```
Main> ordena3 (10,3,7)
(3,7,10)
```

c) Escribe propiedades análogas a las del apartado anterior pero para esta función, y compruébalas usando *QuickCheck*.

4. Aunque ya existe una función predefinida (`max :: Ord a => a -> a -> a`) para calcular el máximo de dos valores, el objetivo de este ejercicio es que definas tu propia versión de dicha función.

a) Como no está permitido redefinir una función predefinida, define una nueva y llámala `max2 :: Ord a => a -> a -> a` de forma que satisfaga:

```
Main> 10 `max2` 7                   Main> max2 'a' 'z'
10                                  'z'
```

b) Define las siguientes propiedades que debería verificar tu función `max2` y compruébalas con *QuickCheck* (recuerda importar `Test.QuickCheck` al principio de tu programa):

i. `p1_max2`: el máximo de dos números *x* e *y* coincide o bien con *x* o bien con *y*.
ii. `p2_max2`: el máximo de *x* e *y* es mayor o igual que *x* , así como mayor o igual que *y.*
iii. `p3_max2`: si *x* es mayor o igual que *y*, entonces el máximo de *x* e *y* es *x*.
iv. `p4_max2`: si *y* es mayor o igual que *x*, entonces el máximo de *x* e *y* es *y*.

5. Define una función sobrecargada para tipos con orden

```
entre :: Ord a => a -> (a,a) -> Bool
```

que tome un valor *x* además de una tupla con dos valores (*min*,*max*) y compruebe si *x* pertenece al intervalo determinado por *min* y *max*, es decir, si *x* ∈ [*min*,*max*], devolviendo `True` o `False` según corresponda. Por ejemplo:

```
Main> 5 `entre` (1,10)
True
```

```
Main> entre 'z' ('a','d')
False
```

6. Define una función sobrecargada para tipos con igualdad

```
iguales3 :: Eq a => (a,a,a) -> Bool
```

que tome una tupla con tres valores del mismo tipo y devuelva `True` si todos son iguales. Por ejemplo:

```
Main> iguales3 ('z','a','z')
False
Main> iguales3 (5+1,6,2*3)
True
```

7. Recuerda que el cociente y el resto de la división de enteros se corresponde con las funciones predefinidas `div` y `mod`.

a) Define una función `descomponer` que, dada una cantidad positiva de segundos, devuelva la descomposición en horas, minutos y segundos en forma de tupla, de modo que los minutos y segundos de la tupla estén en el rango 0 a 59. Por ejemplo:

```
descomponer 5000 → (1,23,20)          descomponer 100 → (0,1,40)
```

Para ello, completa la siguiente definición:

```
type TotalSegundos = Integer
type Horas         = Integer
type Minutos       = Integer
type Segundos      = Integer
descomponer :: TotalSegundos -> (Horas,Minutos,Segundos)
descomponer x = (horas, minutos, segundos)
  where
    horas = ...
    ...
```

b) Comprueba la corrección de tu función verificando con *QuickCheck* que cumple la siguiente propiedad:

```
p_descomponer x = x>=0 ==> h*3600 + m*60 + s == x
                           && entre m (0,59)
                           && entre s (0,59)
  where (h,m,s) = descomponer x
```

8. Sea la siguiente definición que representa que un euro son 166.386 pesetas:

```
unEuro :: Double
unEuro = 166.386
```

a) Define una función `pesetasAEuros` que convierta una cantidad (de tipo `Double`) de pesetas en los correspondientes euros. Por ejemplo:

```
pesetasAEuros 1663.86 → 10.0
```

b) Define la función `eurosAPesetas` que convierta euros en pesetas. Por ejemplo:

`eurosAPesetas 10 → 1663.86`

c) Sea la siguiente propiedad, que establece que si pasamos una cantidad de pesetas a euros y los euros los volvemos a pasar a pesetas, obtenemos las pesetas originales (es decir, que las funciones definidas previamente son inversas):

```
p_inversas x = eurosAPesetas (pesetasAEuros x) == x
```

Compruébala con *QuickCheck* para ver que no se verifica. ¿por qué falla? (**pista**: estamos trabajando con números flotantes).

9. Sea el siguiente operador que comprueba si dos valores de tipo `Double` son aproximadamente iguales:

```
infix 4 ~=
(~=) :: Double -> Double -> Bool
x ~= y = abs (x-y) < epsilon
 where epsilon = 1/1000
```

Por ejemplo: `(1/3) ~= 0.33 → False` `(1/3) ~= 0.333 → True`

Copia esta definición de operador en tu fichero de programa, y cambia la propiedad `p_inversas` del ejercicio anterior para que verifique que si pasamos una cantidad de pesetas a euros y los euros los volvemos a pasar a pesetas, obtenemos las pesetas originales **aproximadamente**. Comprueba con *QuickCheck* que esta propiedad sí se verifica.

10. Consideremos la ecuación de segundo grado $ax^2 + bx + c = 0$.

a) Define una función `raíces` que tome tres parámetros (correspondientes a los coeficientes a, b y c de la ecuación) y devuelva una tupla con las dos soluciones reales de la ecuación (para calcular la raíz cuadrada, usa la función predefinida `sqrt`). Recuerda que el discriminante se define como b^2-$4ac$ y que la ecuación tiene raíces reales si el discriminante no es negativo. Por ejemplo:

```
raíces 1 (-2) 1.0 → (1.0,1.0)
raíces 1.0 2 4 → Exception: Raíces no reales
```

b) Sea la siguiente propiedad que comprueba que las valores devueltos por la función `raíces` son efectivamente raíces de la ecuación:

```
p1_raíces a b c  = esRaíz r1 && esRaíz r2
  where
   (r1,r2) = raíces a b c
   esRaíz r = a*r^2 + b*r + c ~= 0
```

Comprueba esta propiedad con *QuickCheck* y verifica que falla. Piensa por qué falla, y añade condiciones a la propiedad para que no falle, es decir, completa las interrogaciones:

```
p2_raíces a b c  = ??????? && ?????? ==> esRaíz r1 && esRaíz r2
  where
    (r1,r2) = raíces a b c
    esRaíz r = a*r^2 + b*r + c ~= 0
```

de forma que se verifique el siguiente diálogo:

```
Main> quickCheck p2_raíces
+++ OK, passed 100 tests
```

11. Define una función `esMúltiplo` sobrecargada para tipos integrales que tome dos valores x e y, y devuelva `True` si x es múltiplo de y. Por ejemplo:

`esMúltiplo 9 3 → True` `esMúltiplo 7 3 → False`

12. Define el operador de implicación lógica `(==>>) :: Bool -> Bool -> Bool` de forma que sea asociativo a la izquierda, con precedencia menor que los operadores conjunción y disyunción:

```
Main> 3 < 1 ==>> 4 > 2
True
Main> 3 < 1 || 3 > 1 ==>> 4 > 2 && 4 < 2
False
```

Ayuda: puedes escribir ecuaciones directamente para la definición del operador, o bien patrones, completando definiciones tales como:

```
False ==>> y   = True
???   ==>> ??? = ???
```

13. Los años bisiestos son los años múltiplos de 4. Una excepción a esta regla son los años múltiplos de 100, que sólo se consideran bisiestos si además son múltiplos de 400. Define una función `esBisiesto` que tome como parámetro un año y devuelva `True` si es bisiesto. Por ejemplo:

```
esBisiesto 1984 → True        esBisiesto 1985 → False
esBisiesto 1800 → False       esBisiesto 2000 → True
```

Ayuda: utiliza el operador de implicación lógica y la siguiente frase: *"n* es bisiesto si satisface las dos condiciones siguientes: (a) es múltiplo de 4, y (b) si *n* es múltiplo de 100 entonces *n* es múltiplo de 400".

14. Aunque ya existe en Haskell el operador predefinido (^) para calcular potencias, el objetivo de este problema es que definas tus propias versiones recursivas de este operador.

a) A partir de la propiedad $b^n = b \ b^{n-1}$ define una función recursiva `potencia` que tome un entero *b* y un exponente natural *n* y devuelva b^n. Por ejemplo:

```
potencia 2 3 → 8
```

b) A partir de la siguiente propiedad:

$$b^n = \begin{cases} \left(b^{\frac{n}{2}}\right)^2, & \text{si } n \text{ es par} \\ b \cdot \left(b^{\frac{n-1}{2}}\right)^2, & \text{si } n \text{ es impar} \end{cases}$$

define (sin usar la función del apartado anterior) una función recursiva `potencia'` que tome un entero *b* y un exponente natural *n* y devuelva b^n. Por ejemplo:

```
potencia' 2 3 → 8             potencia' 2 4 → 16
```

c) Comprueba con QuickCheck la corrección de ambas funciones mediante la siguiente propiedad:

```
p_pot b n = n>=0 ==> potencia b n == sol
                     && potencia' b n == sol
  where sol = b^n
```

d) Teniendo en cuenta que elevar al cuadrado equivale a realizar un producto, determina el número de productos que realizan ambas funciones para elevar cierta base a un exponente *n*.
Ayuda: para analizar la eficiencia de potencia' considera exponentes que sean potencia de 2.

15. Dado un conjunto finito con todos sus elementos diferentes, llamamos permutación a cada una de las posibles ordenaciones de los elementos de dicho conjunto. Por ejemplo, para el conjunto {1,2,3}, existen un total de 6 permutaciones de sus elementos: {1,2,3}, {1,3,2}, {2,1,3}, {2,3,1}, {3,1,2} y {3,2,1}. El número de permutaciones posibles para un conjunto con *n* elementos viene dada por el factorial de *n* (se suele escribir *n*!), que se define como el producto de todos los números naturales menores o iguales a *n*. Escribe una función `factorial` que tome como parámetro un número natural y devuelva su factorial. Dado que el factorial crece muy rápido, usa el tipo `Integer`, es decir, `factorial :: Integer -> Integer`. Por ejemplo:

`factorial 3` → `6`   `factorial 20` → `2432902008176640000`

16. Este ejercicio estudia la división entera (exacta) de números enteros.

a) Define una función `divideA` que compruebe si su primer argumento divide exactamente al segundo. Por ejemplo:

`2 `divideA` 10` → `True`   `4 `divideA` 10` → `False`

b) Lee, entiende y comprueba con *QuickCheck* la siguiente propiedad referente a la función `divideA`:

```
p1_divideA x y =  y/=0 && y `divideA` x ==> div x y * y == x
```

c) Escribe una propiedad `p2_divideA` para comprobar usando *QuickCheck* que si un número divide a otros dos, también divide a la suma de ambos.

17. La mediana de un conjunto de valores es aquel valor tal que el 50% de los valores del conjunto son menores o iguales a él, y los restantes mayores o iguales. Queremos definir una función para calcular la mediana de los valores de una tupla de cinco elementos

```
mediana :: Ord a => (a,a,a,a,a) -> a
```

de forma que se tenga: `mediana (3,20,1,10,50)` → `10`

Observa que se satisface 1 , 3 ≤ 10 ≤ 20,50. Teniendo en cuenta este detalle, define la función a través de ecuaciones con guardas, completando el siguiente esquema:

```
mediana (x,y,z,t,u)
        | x > z          = mediana (z,y,x,t,u)
        | y > z          = mediana (x,z,y,t,u)
        . . .
```