Ejercicios de “Informática de 1ºde Matemáticas” (2010–11)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 1 de julio de 2011

Índice general

Introducción

Este libro es una recopilación de las soluciones de ejercicios de la asignatura de “Informática” (de 1º del Grado en Matemáticas) correspondientes al curso 2010–11.

El objetivo de los ejercicios es complementar la introducción a la programación funcional y a la algorítmica con Haskell presentada en los temas del curso. Los apuntes de los temas se encuentran en Temas de ”Programación funcional[1].

Los ejercicios sigue el orden de las relaciones de problemas propuestos durante el curso y, resueltos de manera colaborativa, en la wiki

[1] http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/2010-11-IM-temas-PF.pdf

Relación 1

Introducción a la programación funcional

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En esta relación se presenta una colección de ejercicios
-- correspondiente a la introducción a la programación funcional
-- presentada en el tema 1 cuyas transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-1.pdf

-- ---------------------------------------------------------------------
-- Ejercicio 1. Calcular de una forma distinta a la que se encuentra en
-- las transparencias el resultado de doble (doble 3).
-- ---------------------------------------------------------------------

-- Otra forma de calcularlo es
--      doble (doble 3)
--    = doble (3 + 3)           [por def. de doble]
--    = (3 + 3) + (3 + 3)       [por def. de doble]
--    = 6 + (3 + 3)             [por def. de +]
--    = 6 + 6                   [por def. de +]
--    = 12                      [por def. de +]

-- Hay más formas, por ejemplo,
--      doble (doble 3)
--    = (doble 3) + (doble 3)   [por def. de doble]
```

```
--    = (doble 3) + (3 +3)      [por def. de doble]
--    = (doble 3) + 6           [por def. de +]
--    = (3 + 3) + 6             [por def. de doble]
--    = 6 + 6                   [por def. de +]
--    = 12                      [por def. de +]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que sum [x] = x para cualquier número x.
-- ---------------------------------------------------------------------

-- La demostración es
--      sum [x]
--    = x + sum []   [por def. de sum]
--    = x + 0        [por def. de sum]
--    = x            [por def. de +]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función producto que calcule el producto de
-- una lista de números y, usando la definición, demostrar que producto
-- [2, 3, 4] = 24.
-- ---------------------------------------------------------------------

-- La definición es
producto []     = 1
producto (x:xs) = x * (producto xs)

-- La demostración es
--      producto [2,3,4]
--    = 2 * (producto [3,4])           [por def. de producto]
--    = 2 * (3 * (producto [4]))       [por def. de producto]
--    = 2 * (3 * (4 * (producto [])))  [por def. de producto]
--    = 2 * (3 * (4 * 1))              [por def. de producto]
--    = 2 * (3 * 4)                    [por def. de *]
--    = 2 * 12                         [por def. de *]
--    = 24                             [por def. de *]

-- ---------------------------------------------------------------------
-- Ejercicio 4. ¿Cómo hay que modificar la definición de ordena de las
-- transparencias para que devuelva la lista ordenada de mayor a menor?
-- ---------------------------------------------------------------------
```

```
-- Hay que sustituir la segunda ecuación por
--    ordena (x:xs) = mayores ++ [x] ++ menores

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. ¿Cuál es el efecto de cambiar <= por < en la
-- definición de ordena?
-- ---------------------------------------------------------------------

-- En el resultado se eliminan las repeticiones de elementos.

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. ¿Cuál es el valor de ordena [2,2,3,1,1] con la nueva
-- definición? Escribir su cálculo.
-- ---------------------------------------------------------------------

-- El cálculo es
--      ordena [2,2,3,1,1]
--    =    {por def. de ordena}
--      (ordena [1,1]) ++ [2] ++ (ordena [3])
--    =    {por def. de ordena}
--      ((ordena []) ++ [1] ++ (ordena [])) ++ [2] ++ (ordena [3])
--    =    {por def. de ordena}
--      ([] ++ [1] ++ []) ++ [2] ++ (ordena [3])
--    =    {por def. de ++}
--      [1] ++ [2] ++ (ordena [3])
--    =    {por def. de ++}
--      [1,2] ++ (ordena [3])
--    =    {por def. de ordena}
--      [1,2] ++ ((ordena []) ++ [3] + (ordena []))
--    =    {por def. de ordena}
--      [1,2] ++ ([] ++ [3] + [])
--    =    {por def. de ++}
--      [1,2] ++ [3]
--    =    {por def. de ++}
--      [1,2,3]
```

Relación 2

Introducción a la programación funcional con Haskell y QuickCheck

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- Esta relación consta de tres partes:
-- 1. Ejercicios correspondientes a la introducción a la programación
--    funcional con Haskell presentada en el tema 2 cuyas transparencias
--    se encuentran en
--       http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-2.pdf
-- 2. Ejercicios para comprobar propiedades de funciones usando
--    QuickCheck.
-- 3. Ejercicios correspondientes a las definiciones elementales de
--    funciones presentadas en el tema 4 cuyas transparencias se
--    encuentran en
--       http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-4.pdf
-- Las funciones definidas calculan
-- * el último elemento de una lista,
-- * la lista sin el último elemento,
-- * el cambio de euros a pesetas,
-- * el cambio de pesetas a euros,
-- * la división de una lista en dos mitades casi iguales,
-- * el resto seguro
-- * la diyunción lógica,
```

```haskell
-- * la multiplicación de 3 números mediante lambda,
-- * el máximo de 2 enteros,
-- * el menor múltiplo mayor o igual que un número dado,

-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares                                 --
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- 1. Introducción a la programación funcional con Haskell             --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Escribir los paréntesis correspondientes a las
-- siguientes expresiones:
--    * 2^3+4
--    * 2*3+4*5
--    * 2+3*4^5
-- ---------------------------------------------------------------------

-- (2^3)+4
-- (2*3)+(4*5)
-- 2+(3*(4^5))

-- ---------------------------------------------------------------------
-- Ejercicio 2. La siguiente definición contiene 4 errores sintácticos.
-- Corregir los errores y comprobar que la definición es correcta
-- calculando con Haskell el valor de n.
--    n = A 'div' length xs
--          where
--              A = 10
--             xs = [1, 2, 3, 4, 5]
-- ---------------------------------------------------------------------

-- La definición correcta es
n = a `div` length xs
    where
      a  = 10
```

```
      xs = [1, 2, 3, 4, 5]

-- La evaluación es
--    *Main> n
--    2

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función ultimo tal que (ultimo xs) es el
-- último elemento de la lista no vacía xs. Por ejemplo,
--    ultimo [2,5,3]  ==  3
-- Dar dos definiciones usando las funciones introducidas en el tema 2 y
-- comprobar su equivalencia con QuickCheck.
--
-- Nota: La función ultimo es la predefinida last.
-- ---------------------------------------------------------------------

-- Primera definición:
ultimo_1 xs = head (reverse xs)

-- Segunda definición:
ultimo_2 xs = xs !! (length xs - 1)

-- La propiedad es
prop_ultimo x xs = ultimo_1 (x:xs) == ultimo_2 (x:xs)

-- La comprobación es
--    *Main> quickCheck prop_ultimo
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función iniciales tal que (iniciales xs) es
-- la lista obtenida eliminando el último elemento de la lista no vacía
-- xs. Por ejemplo,
--    iniciales [2,5,3]  ==  [2,5]
-- Dar dos definiciones usando las funciones introducidas en este tema y
-- comprobar su equivalencia usando QuickCheck.
---
-- Nota: La función iniciales es la predefinida init.
-- ---------------------------------------------------------------------
```

```haskell
-- Primera definición:
iniciales_1 xs = take (length xs - 1) xs

-- Segunda definición:
iniciales_2 xs = reverse (tail (reverse xs))

-- La propiedad es
prop_iniciales x xs = iniciales_1 (x:xs) == iniciales_2 (x:xs)

-- La comprobación es
--    *Main> quickCheck prop_iniciales
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5. Comprobar con QuickCheck que para toda lista no vacía
-- xs, la lista obtenida al añadirle a los iniciales de xs la lista
-- formada por el último elemento de xs es xs.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_iniciales_ultimo x xs =
    iniciales_1 (x:xs) ++ [ultimo_1 (x:xs)] == (x:xs)

-- La comprobación es
--    *Main> quickCheck prop_iniciales_ultimo
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- 2. Comprobación de propiedades con QuickCheck                      --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 6 (Transformación entre euros y pesetas)
-- ---------------------------------------------------------------------
-- Ejercicio 6.1. Calcular cuántas pesetas son 49 euros (1 euro son
-- 166.386 pesetas).
-- ---------------------------------------------------------------------

-- El cálculo es
--    Hugs> 49*166.386
```

```
--    8152.914

-- ---------------------------------------------------------------------
-- Ejercicio 6.2, Definir la constante cambioEuro cuyo valor es 166.386
-- y repetir el cálculo anterior usando la constante definida.
-- ---------------------------------------------------------------------

-- La definición es
cambioEuro = 166.386

-- y el cálculo es
--    Main> 49*cambioEuro
--    8152.914

-- ---------------------------------------------------------------------
-- Ejercicio 6.3. Definir la función pesetas tal que (pesetas x) es la
-- cantidad de pesetas correspondientes a x euros y repetir el cálculo
-- anterior usando la función definida.
-- ---------------------------------------------------------------------

-- La definición es
pesetas x = x*cambioEuro

-- y el cálculo es
--    Main> pesetas 49
--    8152.914

-- ---------------------------------------------------------------------
-- Ejercicio 6.4. Definir la función euros tal que (euros x) es la
-- cantidad de euros correspondientes a x pesetas y calcular los euros
-- correspondientes a 8152.914 pesetas.
-- ---------------------------------------------------------------------

-- La definición es
euros x = x/cambioEuro

-- y el cálculo es
--    Main> euros 8152.914
--    49.0
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 6.5. Definir la propiedad prop_EurosPesetas tal que
-- (prop_EurosPesetas x) se verifique si al transformar x euros en
-- pesetas y las pesetas obtenidas en euros se obtienen x
-- euros. Comprobar la prop_EurosPesetas con 49 euros.
-- ---------------------------------------------------------------------

-- La definición es
prop_EurosPesetas x =
    euros(pesetas x) == x

-- y la comprobación es
--    Main> prop_EurosPesetas 49
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 6.6. Comprobar la prop_EurosPesetas con QuickCheck.
-- ---------------------------------------------------------------------

-- Para usar QuickCheck hay que importarlo escribiendo, al comienzo del
-- fichero, import Test.QuickCheck

-- La comprobación es
--    Main> quickCheck prop_EurosPesetas
--    Falsifiable, after 42 tests:
--    3.625
-- lo que indica que no se cumple para 3.625.

-- ---------------------------------------------------------------------
-- Ejercicio 6.7. Calcular la diferencia entre euros(pesetas 3.625) y
-- 3.625.
-- ---------------------------------------------------------------------

-- El cálculo es
--    Main> euros(pesetas 3.625)-3.625
--    -4.44089209850063e-16

-- ---------------------------------------------------------------------
-- Ejercicio 6.8. Definir el operador ~= tal que (x ~= y) se verifique
-- si el valor absoluto de la diferencia entre x e y es menor que una
```

```
-- milésima. Se dice que x e y son casi iguales.
-- ---------------------------------------------------------------------

-- La definición es
x ~= y = abs(x-y)<0.001

-- ---------------------------------------------------------------------
-- Ejercicio 6.9. Definir la propiedad prop_EurosPesetas' tal que
-- (prop_EurosPesetas' x) se verifique si al transformar x euros en
-- pesetas y las pesetas obtenidas en euros se obtiene una cantidad casi
-- igual a x de euros. Comprobar la prop_EurosPesetas' con 49 euros.
-- ---------------------------------------------------------------------

-- La definición es
prop_EurosPesetas' x =
    euros(pesetas x) ~= x

-- y la comprobación es
--    Main> prop_EurosPesetas' 49
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 6.10. Comprobar la prop_EurosPesetas' con QuickCheck.
-- ---------------------------------------------------------------------

-- La comprobación es
--    Main> quickCheck prop_EurosPesetas'
--    OK, passed 100 tests.
-- lo que indica que se cumple para los 100 casos de pruebas
-- considerados.

-- ---------------------------------------------------------------------
-- 3. Definiciones elementales de funciones                            --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    mitades :: [a] -> ([a],[a])
-- tal que, dada una lista de longitud par, la divida en dos mitades,
-- devolviendo las dos listas correspondientes. Por ejemplo,
```

```haskell
--    *Main> mitades [1,2,3,4,5,6]
--    ([1,2,3],[4,5,6])
-- Dar dos definiciones de mitades y comprobar con QuickCheck que son
-- equivalentes.
-- ---------------------------------------------------------------------

-- Primera definición:
mitades_1 :: [a] -> ([a],[a])
mitades_1 xs = splitAt (length xs `div` 2) xs

-- Segunda definición:
mitades_2 :: [a] -> ([a],[a])
mitades_2 xs = (take n xs, drop n xs)
    where n = length xs `div` 2

-- La propiedad de equivalencia es
prop_mitades :: Eq a => [a] -> Bool
prop_mitades xs = mitades_1 xs == mitades_2 xs

-- La comprobación es
--    *Main> quickCheck prop_mitades
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    tailSeguro :: [a] -> [a]
-- que se comporta como la función tail, excepto que tailSeguro aplica
-- la lista vacía en sí misma y tail devuelve error en ese caso. Dar
-- cuatro definiciones distintas utilizando
--    1. una expresión condicional
--    2. guardas
--    3. ecuaciones
--    4. patrones
-- Comprobar con QuickCheck que las definiciones son equivalentes.
--
-- Indicación: Usar la función null.
-- ---------------------------------------------------------------------

-- Primera definición (con una expresión condicional):
tailSeguro_1 :: [a] -> [a]
```

```haskell
tailSeguro_1 xs = if null xs then [] else tail xs

-- Segunda definición (con guardas):
tailSeguro_2 :: [a] -> [a]
tailSeguro_2 xs | null xs = []
                | otherwise = tail xs

-- Tercera definición (con ecuaciones):
tailSeguro_3 :: [a] -> [a]
tailSeguro_3 [] = []
tailSeguro_3 xs = tail xs

-- Cuarta definición (con patrones):
tailSeguro_4 :: [a] -> [a]
tailSeguro_4 []     = []
tailSeguro_4 (_:xs) = xs

-- La propiedad de equivalencia es
prop_tailSeguro :: Eq a => [a] -> Bool
prop_tailSeguro xs =
    tailSeguro_1 xs == tailSeguro_2 xs &&
    tailSeguro_1 xs == tailSeguro_3 xs &&
    tailSeguro_1 xs == tailSeguro_4 xs

-- La comprobación es
--    *Main> quickCheck prop_tailSeguro
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 9. De manera similar al operador conjunción, demostrar cómo
-- usando emparejamiento de patrones el operador lógico disjunción puede
-- definirse de cuatro formas diferentes (disj_1, disj_2, disj_3 y
-- disj_4). Comprobar con QuickCheck que las definiciones son
-- equivalentes entre si y con el operador (||).
-- ---------------------------------------------------------------------

-- Primera definición:
disj_1 False False = False
disj_1 False True  = True
disj_1 True  False = True
```

```haskell
disj_1 True  True  = True

-- Segunda definición:
disj_2 False False = False
disj_2 _     _     = True

-- Tercera definición:
disj_3 False b = b
disj_3 True  _ = True

-- Cuarta definición:
disj_4 b c | b == c    = b
           | otherwise = True

-- Propiedad de equivalencia:
prop_disj :: Bool -> Bool ->  Bool
prop_disj x y =
    disj_1 x y == disj_2 x y &&
    disj_1 x y == disj_3 x y &&
    disj_1 x y == disj_4 x y &&
    disj_1 x y == x || y

-- La comprobación es
--    *Main> quickCheck prop_disj
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 10. Redefinir la función
--    mult x y z = x*y*z
-- usando expresiones lambda y comprobar con QuickCheck que las
-- definiciones son equivalentes.
-- ---------------------------------------------------------------------

mult x y z = x*y*z

-- La definición es
mult' = \x -> (\y -> (\z -> x*y*z ))

-- La propiedad es
prop_mult x y z = mult x y z == mult' x y z
```

```haskell
-- La comprobación es
--    *Main> quickCheck prop_mult
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 11 (Máximo de enteros)
-- ---------------------------------------------------------------------
-- Ejercicio 11.1. Definir, por casos, la función maxI tal que maxI x y
-- es el máximo de los números enteros x e y. Por ejemplo,
--    maxI 2 5 => 5
--    maxI 7 5 => 7
-- ---------------------------------------------------------------------

-- La definición es
maxI :: Integer -> Integer -> Integer
maxI x y | x >= y    = x
         | otherwise = y

-- ---------------------------------------------------------------------
-- Ejercicio 11.2. En este apartado vamos a comprobar propiedades del
-- máximo.
-- ---------------------------------------------------------------------
-- Ejercicio 11.2.1. Comprobar con QuickCheck que el máximo de x e y es
-- mayor o igual que x y que y.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_MaxIMayor x y =
  maxI x y >= x && maxI x y >= y

-- y la comprobación es
--    Main> quickCheck prop_MaxIMayor
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 11.2.2. Comprobar con QuickCheck que el máximo de x e y es
-- x ó y.
-- ---------------------------------------------------------------------
```

```haskell
-- La propiedad es
prop_MaxIAlguno x y =
  maxI x y == x || maxI x y == y

-- y la comprobación es
--    Main> quickCheck prop_MaxIAlguno
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 11.2.3. Comprobar con QuickCheck que si x es mayor o igual
-- que y, entonces el máximo de x e y es x.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_MaxIX x y =
  x >= y ==> maxI x y == x

-- y la comprobación es
--    Main> quickCheck prop_MaxIX
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 11.2.4. Comprobar con QuickCheck que si y es mayor o igual
-- que x, entonces el máximo de x e y es y.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_MaxIY x y =
  y >= x ==> maxI x y == y

-- y la comprobación es
--    Main> quickCheck prop_MaxIY
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 12 (Menor múltiplo mayor o igual que)
-- ---------------------------------------------------------------------
-- El objetivo de este ejercicio consiste en definir la función
-- menorMultiplo tal que (menorMultiplo n p) es el menor número mayor o
-- igual que n que es múltiplo de p. Por ejemplo,
```

```
--    menorMultiplo 16 5   == 20
--    menorMultiplo 167 25 == 175
-- Para ello, partiremos de la siguiente propiedad
--    menorMultiplo n p = n + (p - (n resto p))
-- es decir, n más la diferencia entre p y el resto de la división entre
-- n y p.

-- ---------------------------------------------------------------------
-- Ejercicio 12.1. Definir la función menorMultiplo.
-- ---------------------------------------------------------------------

-- La definición es
menorMultiplo :: Integer -> Integer -> Integer
menorMultiplo n p =  n + (p - (n `rem` p))

-- ---------------------------------------------------------------------
-- Ejercicio 12.2. Calcular el resultado de menorMultiplo para los
-- ejemplos anteriores.
-- ---------------------------------------------------------------------

-- Los cálculos son:
--    menorMultiplo 16 5    ==  20
--    menorMultiplo 167 25  ==  175

-- ---------------------------------------------------------------------
-- Ejercicio 12.3. En este apartado vamos a estudiar propiedades de la
-- función menorMultiplo.
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 12.3.1. Comprobar con QuickCheck si (menorMultiplo n p) es
-- mayor o igual que n.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_menorMultiplo_1 n p =
    menorMultiplo n p >= n

-- y la comprobación es
--    Main> quickCheck prop_menorMultiplo_1
```

```
--    Falsifiable, after 0 tests:
--    -2
--    -3

-- La propiedad no se verifica. Pero sí se verifica cuando los
-- parámetros son positivos:
prop_menorMultiplo_1' n p =
    n>0 && p>0 ==> menorMultiplo n p >= n

-- En efecto,
--    Main> quickCheck prop_menorMultiplo_1'
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 12.3.2. Comprobar con QuickCheck si (menorMultiplo n p) es
-- múltiplo de p.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_menorMultiplo_2 n p =
    n>0 && p>0 ==> multiplo (menorMultiplo n p) p

-- donde (múltiplo n p) se verifica si n es un múltiplo de p.
multiplo :: Integer -> Integer -> Bool
multiplo n p = n `mod` p == 0

-- La comprobación es
--    Main> quickCheck prop_menorMultiplo_2
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 12.3.3. Comprobar con QuickCheck si (menorMultiplo n n) es
-- igual a n.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_menorMultiplo_3 n =
    n>0 ==> menorMultiplo n n == n

-- La comprobación es
```

```
--    Main> quickCheck prop_menorMultiplo_3
--    Falsifiable, after 0 tests:
--    1

-- La propiedad no se verifica. En efecto,
--    menorMultiplo 1 1  ==  2

-- ---------------------------------------------------------------------
-- Ejercicio 12.4. Corregir los errores de la definición de menorMultiplo
-- escribiendo una nueva función menorMultiplo'. Definir las
-- correspondientes propiedades y comprobarlas.
-- ---------------------------------------------------------------------

-- La definición es
menorMultiplo' :: Integer -> Integer -> Integer
menorMultiplo' n p | multiplo n p = n
                   | otherwise    = n + (p - (n `rem` p))

-- Las propiedades son
prop_menorMultiplo'_1 n p =
    n>0 && p>0 ==> menorMultiplo' n p >= n

prop_menorMultiplo'_2 n p =
    n>0 && p>0 ==> multiplo (menorMultiplo' n p) p

prop_menorMultiplo'_3 n =
    n>0 ==> menorMultiplo' n n == n

-- y sus comprobaciones son
--    Main> quickCheck prop_menorMultiplo'_1
--    OK, passed 100 tests.
--
--    Main> quickCheck prop_menorMultiplo'_2
--    OK, passed 100 tests.
--
--    Main> quickCheck prop_menorMultiplo'_3
--    OK, passed 100 tests.
```

Relación 3

Definiciones elementales de funciones (1)

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En esta relación se presentan ejercicios con definiciones elementales
-- (no recursivas) de funciones correspondientes al tema 4 cuyas
-- transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-4.pdf
-- En concreto, se estudian funciones para calcular
-- * las raíces de las ecuaciones cuadráticas y
-- * la disyunción excluyente.

-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares                                 --
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1 (Raices de una ecuación de segundo grado)
-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función raices de forma que raices a b c
-- devuelve la lista de las raices reales de la ecuación
-- ax^2 + bx + c = 0. Por ejemplo,
--    raices 1 (-2) 1 => [1.0,1.0]
```

```haskell
--    raices 1 2 3    => [-1.0,-2.0]
-- Escribir distintas definiciones de la función raices.
-- ---------------------------------------------------------------------

raices_1 :: Double -> Double -> Double -> [Double]
raices_1 a b c = [(-b+d)/t,(-b-d)/t]
    where d = sqrt (b^2 - 4*a*c)
          t = 2*a

raices_2 :: Double -> Double -> Double -> [Double]
raices_2 a b c
    | d >= 0    = [(-b+e)/(2*a), (-b-e)/(2*a)]
    | otherwise = error "No tine raices reales"
    where d = b^2-4*a*c
          e = sqrt d

-- Sesión:
-- *Main> raices_2 1 (-2) 1
-- [1.0,1.0]
-- *Main> raices_2 1 2 3
-- *** Exception: No tine raices reales
-- *Main> raices_2 1 3 2
-- [-1.0,-2.0]
-- *Main> raices_1 1 (-2) 1
-- [1.0,1.0]
-- *Main> raices_1 1 2 3
-- [NaN,NaN]
-- *Main> raices_1 1 3 2
-- [-1.0,-2.0]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickCheck si todas las definiciones
-- coinciden.
-- ---------------------------------------------------------------------

prop_raices12 :: Double -> Double -> Double -> Bool
prop_raices12 a b c =
  raices_2 a b c == raices_1 a b c

-- *Main> quickCheck prop_raices12
```

```haskell
-- Falsifiable, after 0 tests:
-- 0.0
-- 0.0
-- 0.0

prop_raices12_b :: Double -> Double -> Double -> Property
prop_raices12_b a b c =
  a/=0 && not(b==0 &&c==0) && (b^2 - 4*a*c) >= 0
  ==>
  raices_2 a b c == raices_1 a b c

-- *Main> quickCheck prop_raices12_b
-- OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comprobar con QuickCheck que las definiciones son
-- correctas; es decir, que los elementos del resultado son raices de la
-- ecuación.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_raices1_correcta a b c =
  a /= 0 && (b^2-4*a*c) >= 0
  ==> (a*x^2 + b*x + c) ~= 0 && (a*y^2 + b*y + c) ~= 0
  where [x,y]  = raices_1 a b c
        x ~= y = abs(x-y)<0.001

-- La comprobación es
--    *Main> quickCheck prop_raices1_correcta
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2. La disyunción excluyente xor de dos fórmulas se verifica
-- si una es verdadera y la otra es falsa.
-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir la función xor_1 que calcule la disyunción
-- excluyente a partir de la tabla de verdad. Usar 4 ecuaciones, una por
-- cada línea de la tabla.
-- ---------------------------------------------------------------------
```

```haskell
xor_1 :: Bool -> Bool -> Bool
xor_1 True  True  = False
xor_1 True  False = True
xor_1 False True  = True
xor_1 False False = False

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función xor_2 que calcule la disyunción
-- excluyente a partir de la tabla de verdad y patrones. Usar 2
-- ecuaciones, una por cada valor del primer argumento.
-- ---------------------------------------------------------------------

xor_2 :: Bool -> Bool -> Bool
xor_2 True  y = not y
xor_2 False y = y

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Definir la función xor_3 que calcule la disyunción
-- excluyente a partir de la disyunción (||), conjunción (&&) y negación
-- (not). Usar 1 ecuación.
-- ---------------------------------------------------------------------

xor_3 :: Bool -> Bool -> Bool
xor_3 x y = (x || y) && not (x && y)

-- ---------------------------------------------------------------------
-- Ejercicio 2.4. Definir la función xor_4 que calcule la disyunción
-- excluyente a partir de desigualdad (/=). Usar 1 ecuación.
-- ---------------------------------------------------------------------

xor_4 :: Bool -> Bool -> Bool
xor_4 x y = x /= y

-- ---------------------------------------------------------------------
-- Ejercicio 2.5. Comprobar con QuickCheck la equivalencia de las 4
-- definiciones anteriores.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_xor x y =
```

```
    xor_1 x y == xor_2 x y &&
    xor_2 x y == xor_3 x y &&
    xor_3 x y == xor_4 x y

-- La comprobación es
--    *Main> quickCheck prop_xor
--    +++ OK, passed 100 tests.
```

Relación 4

Definiciones elementales de funciones (2)

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En esta relación se amplían los ejercicios con definiciones
-- elementales (no recursivas) de funciones correspondientes al tema 4
-- cuyas transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-4.pdf
-- En concreto, se estudian funciones para calcular
-- * la media de 3 números,
-- * la suma de euros de una colección de monedas,
-- * el volumen de la esfera,
-- * el área de una corona circular,
-- * la intercalación de pares,
-- * el reagrupamiento de ternas,
-- * la rotación de listas,
-- * el rango de una lista,
-- * el reconocimiento de palíndromos,
-- * la igualdad y diferencia de 3 elementos,
-- * la igualdad de 4 elementos,
-- * el máximo de 3 elementos,
-- * el módulo de un vector,
-- * el cuadrante de un punto,
-- * el intercambio de coordenadas,
-- * el simétrico de un punto,
```

```
-- * la división segura y
-- * el área de un triángulo mediante la fórmula de Herón.

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función media3 tal que (media3 x y z) es
-- la media aritmética de los números x, y y z. Por ejemplo,
--    media3 1 3 8     ==  4.0
--    media3 (-1) 0 7  ==  2.0
--    media3 (-3) 0 3  ==  0.0
-- ---------------------------------------------------------------------

media3 x y z = (x+y+z)/3

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función sumaMonedas tal que
-- (sumaMonedas a b c d e) es la suma de los euros correspondientes a
-- a monedas de 1 euro, b de 2 euros, c de 5 euros, d 10 euros y
-- e de 20 euros. Por ejemplo,
--    sumaMonedas 0 0 0 0 1  ==  20
--    sumaMonedas 0 0 8 0 3  == 100
--    sumaMonedas 1 1 1 1 1  ==  38
-- ---------------------------------------------------------------------

sumaMonedas a b c d e = 1*a+2*b+5*c+10*d+20*e

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función volumenEsfera tal que
-- (volumenEsfera r) es el volumen de la esfera de radio r. Por ejemplo,
--    volumenEsfera 10  ==  4188.790204786391
-- Indicación: Usar la constante pi.
-- ---------------------------------------------------------------------

volumenEsfera r = (4/3)*pi*r^3

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función areaDeCoronaCircular tal que
-- (areaDeCoronaCircular r1 r2) es el área de una corona circular de
-- radio interior r1 y radio exterior r2. Por ejemplo,
--    areaDeCoronaCircular 1 2 == 9.42477796076938
--    areaDeCoronaCircular 2 5 == 65.97344572538566
```

```
--    areaDeCoronaCircular 3 5 == 50.26548245743669
-- ---------------------------------------------------------------------

areaDeCoronaCircular r1 r2 = pi*(r2^2 -r1^2)

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función intercala que reciba dos listas xs e
-- ys de dos elementos cada una, y devuelva una lista de cuatro
-- elementos, construida intercalando los elementos de xs e ys. Por
-- ejemplo,
--    intercala [1,4] [3,2]  ==  [1,3,4,2]
-- ---------------------------------------------------------------------

intercala [x1,x2] [y1,y2] = [x1,y1,x2,y2]

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función reagrupa que tome una tupla cuyos
-- elementos son tres tuplas de tres elementos cada una, y actúe del
-- siguiente modo:
--    reagrupa ((1,2,3),(4,5,6),(7,8,9))  ==  ((1,4,7),(2,5,8),(3,6,9))
-- ---------------------------------------------------------------------

reagrupa ((x1,x2,x3),(y1,y2,y3),(z1,z2,z3)) =
    ((x1,y1,z1),(x2,y2,z2),(x3,y3,z3))

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función rota1 tal que (rota1 xs) es la lista
-- obtenida poniendo el primer elemento de xs al final de la lista. Por
-- ejemplo,
--    rota1 [3,2,5,7]  ==  [2,5,7,3]
-- ---------------------------------------------------------------------

rota1 xs = tail xs ++ [head xs]

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función rota tal que (rota n xs) es la lista
-- obtenida poniendo los n primeros elementos de xs al final de la
-- lista. Por ejemplo,
--    rota 1 [3,2,5,7]  ==  [2,5,7,3]
--    rota 2 [3,2,5,7]  ==  [5,7,3,2]
```

```
--    rota 3 [3,2,5,7]  ==  [7,3,2,5]
-- ---------------------------------------------------------------------

rota n xs = drop n xs ++ take n xs

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función rango tal que (rango xs) es la
-- lista formada por el menor y mayor elemento de xs.
--    rango [3,2,7,5]  ==  [2,7]
-- Indicación: Se pueden usar minimum y maximum.
-- ---------------------------------------------------------------------

rango xs = [minimum xs, maximum xs]

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función palindromo tal que (palindromo xs) se
-- verifica si xs es un palíndromo; es decir, es lo mismo leer xs de
-- izquierda a derecha que de derecha a izquierda. Por ejemplo,
--    palindromo [3,2,5,2,3]    ==  True
--    palindromo [3,2,5,6,2,3]  ==  False
-- ---------------------------------------------------------------------

palindromo xs = xs == reverse xs

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función tresIguales tal que
-- (tresIguales x y z) se verifica si los elementos x, y y z son
-- iguales. Por ejemplo,
--    tresIguales 4 4 4  ==  True
--    tresIguales 4 3 4  ==  False
-- ---------------------------------------------------------------------

tresIguales x y z = x == y && y == z

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función tresDiferentes tal que
-- (tresDiferentes x y z) se verifica si los elementos x, y y z son
-- distintos. Por ejemplo,
--    tresDiferentes 3 5 2  ==  True
--    tresDiferentes 3 5 3  ==  False
```

```
-- ---------------------------------------------------------------------

tresDiferentes x y z = x /= y && x /= z && y /= z

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la función cuatroIguales tal que
-- (cuatroIguales x y z u) se verifica si los elementos x, y, z y u son
-- iguales. Por ejemplo,
--    cuatroIguales 5 5 5 5   ==  True
--    cuatroIguales 5 5 4 5   ==  False
-- Indicación: Usar la función tresIguales.
-- ---------------------------------------------------------------------

cuatroIguales x y z u = x == y && tresIguales y z u

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función maxTres tal que (maxTres x y z) es
-- el máximo de x, y y z. Por ejemplo,
--    maxTres 6 2 4  ==  6
--    maxTres 6 7 4  ==  7
--    maxTres 6 7 9  ==  9
-- ---------------------------------------------------------------------

maxTres x y z = max x (max y z)

-- ---------------------------------------------------------------------
-- Ejercicio 15. Definir la función modulo tal que (modulo v) es el
-- módulo del vector v. Por ejemplo,
--    modulo (3,4)  ==  5.0
-- ---------------------------------------------------------------------

modulo (x,y) = sqrt(x^2+y^2)

-- ---------------------------------------------------------------------
-- Ejercicio 16. Definir la función cuadrante tal que (cuadrante p) es
-- es cuadrante del punto p (se supone que p no está sobre los
-- ejes). Por ejemplo,
--    cuadrante (3,5)    ==  1
--    cuadrante (-3,5)   ==  2
--    cuadrante (-3,-5)  ==  3
```

```
--    cuadrante (3,-5)   ==  4
-- ---------------------------------------------------------------------

cuadrante (x,y)
    | x > 0 && y > 0 = 1
    | x < 0 && y > 0 = 2
    | x < 0 && y < 0 = 3
    | x > 0 && y < 0 = 4

-- ---------------------------------------------------------------------
-- Ejercicio 17. Definir la función intercambia tal que (intercambia p)
-- es el punto obtenido intercambiando las coordenadas del punto p. Por
-- ejemplo,
--    intercambia (2,5)  ==  (5,2)
--    intercambia (5,2)  ==  (2,5)
-- ---------------------------------------------------------------------

intercambia (x,y) = (y,x)

-- ---------------------------------------------------------------------
-- Ejercicio 18. Definir la función simetricoH tal que (simetricoH p) es
-- el punto simétrico de p respecto del eje horizontal. Por ejemplo,
--    simetricoH (2,5)   ==  (2,-5)
--    simetricoH (2,-5)  ==  (2,5)
-- ---------------------------------------------------------------------

simetricoH (x,y) = (x,-y)

-- ---------------------------------------------------------------------
-- Ejercicio 19. Definir la función divisionSegura tal que
-- (divisionSegura x y) es x/y si y no es cero e y 9999 en caso
-- contrario. Por ejemplo,
--    divisionSegura 7 2  ==  3.5
--    divisionSegura 7 0  ==  9999.0
-- ---------------------------------------------------------------------

divisionSegura _ 0 = 9999
divisionSegura x y = x/y

-- ---------------------------------------------------------------------
```

```haskell
-- Ejercicio 12. En geometría, la fórmula de Herón, descubierta por
-- Herón de Alejandría, dice que el área de un triángulo cuyo lados
-- miden a, b y c es la raíz cuadrada de s(s-a)(s-b)(s-c) donde s es el
-- semiperímetro
--    s = (a+b+c)/2
-- Definir la función area tal que (area a b c) es el área de un
-- triángulo de lados a, b y c. Por ejemplo,
--    area 3 4 5  ==  6.0
-- ---------------------------------------------------------------------

area a b c = sqrt (s*(s-a)*(s-b)*(s-c))
    where s = (a+b+c)/2
```

Relación 5

Definiciones por comprensión (1)

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En esta relación se presentan ejercicios con definiciones por
-- comprensión correspondientes al tema 5 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-5.pdf
-- En concreto, se estudian funciones para calcular
-- * la suma de los cuadrados de los n primeros números,
-- * listas con un elemento replicado,
-- * ternas pitagóricas,
-- * números perfectos,
-- * producto cartesiano,
-- * posiciones de un elemento en una lista,
-- * producto escalar y
-- * la solución del problema 1 del proyecto Euler.

-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares                                 --
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.Char
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir, por comprensión, la función
--    sumaDeCuadrados :: Integer -> Integer
-- tal que (sumaDeCuadrados n) es la suma de los cuadrados de los
-- primeros n números; es decir, 1^2 + 2^2 + ... + n^2. Por ejemplo,
--    sumaDeCuadrados 3    ==  14
--    sumaDeCuadrados 100  ==  338350
-- ---------------------------------------------------------------------

sumaDeCuadrados :: Integer -> Integer
sumaDeCuadrados n = sum [x^2 | x <- [1..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir por comprensión la función
--    replica :: Int -> a -> [a]
-- tal que (replica n x) es la lista formada por n copias del elemento
-- x. Por ejemplo,
--    replica 3 True  ==  [True, True, True]
-- Nota: La función replica es equivalente a la predefinida replicate.
-- ---------------------------------------------------------------------

replica :: Int -> a -> [a]
replica n x = [x | _ <- [1..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2, Comprobar con QuickCheck que para todo número entero
-- positivo n (menor que 100) y todo elemento x, se tiene que la
-- longitud de (replica n x) es n.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_replica :: Int -> Int -> Bool
prop_replica n x = length (replica n' x) == n'
                   where n' = mod (abs n) 100

-- La comprobación es
--    *Main> quickCheck prop_replica
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
```

```haskell
-- Ejercicio 3.1. Una terna (x,y,z) de enteros positivos es pitagórica
-- si x^2 + y^2 = z^2. Usando una lista por comprensión, definir la
-- función
--    pitagoricas :: Int -> [(Int,Int,Int)]
-- tal que (pitagoricas n) es la lista de todas las ternas pitagóricas
-- cuyas componentes están entre 1 y n. Por ejemplo,
--    pitagoricas 10  ==  [(3,4,5),(4,3,5),(6,8,10),(8,6,10)]
-- ---------------------------------------------------------------------

pitagoricas :: Int -> [(Int,Int,Int)]
pitagoricas n = [(x,y,z) | x <- [1..n],
                           y <- [1..n],
                           z <- [1..n],
                           x^2 + y^2 == z^2]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    numeroDePares :: (Int,Int,Int) -> Int
-- tal que (numeroDePares t) es el número de elementos pares de la terna
-- t. Por ejemplo,
--    numeroDePares (3,5,7)  ==  0
--    numeroDePares (3,6,7)  ==  1
--    numeroDePares (3,6,4)  ==  2
--    numeroDePares (4,6,4)  ==  3
-- ---------------------------------------------------------------------

numeroDePares :: (Int,Int,Int) -> Int
numeroDePares (x,y,z) = sum [1 | n <- [x,y,z], even n]

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir la función
--    conjetura :: Int -> Bool
-- tal que (conjetura n) se verifica si todas las ternas pitagóricas
-- cuyas componentes están entre 1 y n tiene un número impar de números
-- pares. Por ejemplo,
--    conjetura 10  ==  True
-- ---------------------------------------------------------------------

conjetura :: Int -> Bool
conjetura n = and [odd (numeroDePares t) | t <- pitagoricas n]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 3.4. Demostrar la conjetura para todas las ternas
-- pitagóricas.
-- ---------------------------------------------------------------------

-- Sea (x,y,z) una terna pitagórica. Entonces x^2+y^2=z^2. Pueden darse
-- 4 casos:
--
-- Caso 1: x e y son pares. Entonces, x^2, y^2 y z^2 también lo
-- son. Luego el número de componentes pares es 3 que es impar.
--
-- Caso 2: x es par e y es impar. Entonces, x^2 es par, y^2 es impar y
-- z^2 es impar. Luego el número de componentes pares es 1 que es impar.
--
-- Caso 3: x es impar e y es par. Análogo al caso 2.
--
-- Caso 4: x e y son impares. Entonces, x^2 e y^2 también son impares y
-- z^2 es par. Luego el número de componentes pares es 1 que es impar.

-- ---------------------------------------------------------------------
-- Ejercicio 4. Un entero positivo es perfecto si es igual a la suma de
-- sus factores, excluyendo el propio número.
--
-- Definir por comprensión la función
--    perfectos :: Int -> [Int]
-- tal que (perfectos n) es la lista de todos los números perfectos
-- menores que n. Por ejemplo,
--    perfectos 500  ==  [6,28,496]
-- Indicación: Usar la función factores del tema 5.
-- ---------------------------------------------------------------------

-- La función factores del tema es
factores :: Int -> [Int]
factores n = [x | x <- [1..n], n `mod` x == 0]

-- La definición es
perfectos :: Int -> [Int]
perfectos n = [x | x <- [1..n], sum (init (factores x)) == x]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 5. La función
--    pares :: [a] -> [b] -> [(a,b)]
-- definida por
--    pares xs ys = [(x,y) | x <- xs, y <- ys]
-- toma como argumento dos listas y devuelve la listas de los pares con
-- el primer elemento de la primera lista y el segundo de la
-- segunda. Por ejemplo,
--    *Main> pares [1..3] [4..6]
--    [(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6)]
--
-- Definir, usando dos listas por comprensión con un generador cada una,
-- la función
--    pares' :: [a] -> [b] -> [(a,b)]
-- tal que pares' sea equivalente a pares.
--
-- Comprobar con QuickCheck que las dos definiciones son equivalentes.
--
-- Indicación: Utilizar la función predefinida concat y encajar una
-- lista por comprensión dentro de la otra.
-- ---------------------------------------------------------------------

-- La definición de pares es
pares :: [a] -> [b] -> [(a,b)]
pares xs ys = [(x,y) | x <- xs, y <- ys]

-- La redefinición de pares es
pares' :: [a] -> [b] -> [(a,b)]
pares' xs ys = concat [[(x,y) | y <- ys] | x <- xs]

-- La propiedad es
prop_pares :: (Eq a, Eq b) => [a] -> [b] -> Bool
prop_pares xs ys = pares xs ys == pares' xs ys

-- La comprobación es
--    *Main> quickCheck prop_pares
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 6. En el tema se ha definido la función
```

```haskell
--    posiciones :: Eq a => a -> [a] -> [Int]
-- tal que (posiciones x xs) es la lista de las posiciones ocupadas por
-- el elemento x en la lista xs. Por ejemplo,
--    posiciones 5 [1,5,3,5,5,7]  ==  [1,3,4]
--
-- Definir, usando la función busca (definida en el tema 5), la función
--    posiciones' :: Eq a => a -> [a] -> [Int]
-- tl que posiciones' sea equivalente a posiciones.
--
-- Comprobar con QuickCheck que posiciones' es equivalente a
-- posiciones.
-- ---------------------------------------------------------------------

-- La definición de posiciones es
posiciones :: Eq a => a -> [a] -> [Int]
posiciones x xs =
    [i | (x',i) <- zip xs [0..n], x == x']
    where n = length xs - 1

-- La definición de busca es
busca :: Eq a => a -> [(a, b)] -> [b]
busca c t = [v | (c', v) <- t, c' == c]

-- La redefinición de posiciones es
posiciones' :: Eq a => a -> [a] -> [Int]
posiciones' x xs = busca x (zip xs [0..])

-- La propiedad de equivalencia es
prop_posiciones :: Eq a => a -> [a] -> Bool
prop_posiciones x xs = posiciones x xs == posiciones' x xs

-- La comprobación es
--    *Main> quickCheck prop_posiciones
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 7. El producto escalar de dos listas de enteros xs y ys de
-- longitud n viene dado por la suma de los productos de los elementos
-- correspondientes.
--
```

```haskell
-- Definir por comprensión la función
--    productoEscalar :: [Int] -> [Int] -> Int
-- tal que (productoEscalar xs ys) es el producto escalar de las listas
-- xs e ys. Por ejemplo,
--    productoEscalar [1,2,3] [4,5,6]  ==  32
--
-- Usar QuickCheck para comprobar la propiedad conmutativa del producto
-- escalar.
-- ---------------------------------------------------------------------

productoEscalar :: [Int] -> [Int] -> Int
productoEscalar xs ys = sum [x*y | (x,y) <- zip xs ys]

prop_conmutativa_productoEscalar xs ys =
   productoEscalar xs ys == productoEscalar ys xs

-- ---------------------------------------------------------------------
-- Ejercicio 8 (Problema 1 del proyecto Euler) Definir la función
--    euler1 :: Integer -> Integer
-- (euler1 n) es la suma de todos los múltiplos de 3 ó 5 menores que
-- n. Por ejemplo,
--    euler1 10  ==  23
--
-- Calcular la suma de todos los múltiplos de 3 ó 5 menores que 1000.
-- ---------------------------------------------------------------------

euler1 :: Integer -> Integer
euler1 n = sum [x | x <- [1..n-1], multiplo x 3 || multiplo x 5]
    where multiplo x y = mod x y == 0

-- Cálculo:
--    *Main> euler1 1000
--    233168
```

Relación 6

Definiciones por comprensión (2)

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En esta relación se presentan más ejercicios con definiciones por
-- comprensión correspondientes al tema 5 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-5.pdf
-- En concreto, se estudian funciones para calcular
-- * el valor aproximado del número e,
-- * el límite de sen(x)/x cuando x tiende a cero,
-- * el valor de pi mediante la aproximación de Leibniz y
-- * la solución del problema 9 del proyecto Euler.

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función aproxE tal que (aproXE n) es la
-- lista cuyos elementos son los términos de la sucesión (1+1/m)**m
-- desde 1 hasta n. Por ejemplo,
--    aproxE 1 == [2.0]
--    aproxE 4 == [2.0,2.25,2.37037037037037,2.44140625]
-- ---------------------------------------------------------------------

aproxE n = [(1+1/m)**m | m <- [1..n]]

-- ---------------------------------------------------------------------
```

```haskell
-- Ejercicio 1.2. ¿Cuál es el límite de la sucesión (1+1/m)**m ?
-- ---------------------------------------------------------------------

-- El límite de la sucesión es el número e.

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función errorE tal que (errorE x) es el
-- menor número de términos de la sucesión (1+1/m)**m necesarios para
-- obtener su límite con un error menor que x. Por ejemplo,
--    errorAproxE 0.1    ==  13.0
--    errorAproxE 0.01   ==  135.0
--    errorAproxE 0.001  ==  1359.0
-- Indicación: En Haskell, e se calcula como (exp 1).
-- ---------------------------------------------------------------------

errorAproxE x = head [m | m <- [1..], abs((exp 1) - (1+1/m)**m) < x]

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir la función aproxLimSeno tal que
-- (aproxLimSeno n) es la lista cuyos elementos son los términos de la
-- sucesión
--    sen(1/m)
--    --------
--      1/m
-- desde 1 hasta n. Por ejemplo,
--    aproxLimSeno 1 == [0.8414709848078965]
--    aproxLimSeno 2 == [0.8414709848078965,0.958851077208406]
-- ---------------------------------------------------------------------

aproxLimSeno n = [sin(1/m)/(1/m) | m <- [1..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. ¿Cuál es el límite de la sucesión sen(1/m)/(1/m) ?
-- ---------------------------------------------------------------------

-- El límite es 1.

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Definir la función errorLimSeno tal que
-- (errorLimSeno x) es el menor número de términos de la sucesión
```

```
-- sen(1/m)/(1/m) necesarios para obtener su límite con un error menor
-- que x. Por ejemplo,
--    errorLimSeno 0.1     ==   2.0
--    errorLimSeno 0.01    ==   5.0
--    errorLimSeno 0.001   ==  13.0
--    errorLimSeno 0.0001  ==  41.0
-- ---------------------------------------------------------------------

errorLimSeno x = head [m | m <- [1..], abs(1 - sin(1/m)/(1/m)) < x]

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función calculaPi tal que (calculaPi n) es
-- la aproximación del número pi calculada mediante la expresión
--    4*(1 - 1/3 + 1/5 - 1/7 + ...+ (-1)**n/(2*n+1))
-- Por ejemplo,
--    calculaPi 3    ==  2.8952380952380956
--    calculaPi 300  ==  3.1449149035588526
-- ---------------------------------------------------------------------

calculaPi n = 4 * sum [(-1)**x/(2*x+1) | x <- [0..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función errorPi tal que
-- (errorPi x) es el menor número de términos de la serie
--    4*(1 - 1/3 + 1/5 - 1/7 + ...+ (-1)**n/(2*n+1))
-- necesarios para obtener pi con un error menor que x. Por ejemplo,
--    errorPi 0.1    ==   9.0
--    errorPi 0.01   ==  99.0
--    errorPi 0.001  ==  999.0
-- ---------------------------------------------------------------------

errorPi x = head [n | n <- [1..], abs (pi - (calculaPi n)) < x]

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir la función suma tal (suma n) es la suma de los
-- n primeros números. Por ejemplo,
--    suma 3  ==  6
-- ---------------------------------------------------------------------

suma n = sum [1..n]
```

```haskell
-- Otra definición es
suma' n = (1+n)*n `div` 2

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Los triángulo aritmético se forman como sigue
--     1
--     2  3
--     4  5  6
--     7  8  9 10
--    11 12 13 14 15
--    16 16 18 19 20 21
-- Definir la función linea tal que (linea n) es la línea n-ésima de los
-- triángulos aritméticos. Por ejemplo,
--    linea 4  ==  [7,8,9,10]
--    linea 5  ==  [11,12,13,14,15]
-- ---------------------------------------------------------------------

linea n = [suma (n-1)+1..suma n]

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Definir la función triangulo tal que (triangulo n) es
-- el triángulo aritmético de altura n. Por ejemplo,
--    triangulo 3  ==  [[1],[2,3],[4,5,6]]
--    triangulo 4  ==  [[1],[2,3],[4,5,6],[7,8,9,10]]
-- ---------------------------------------------------------------------

triangulo n = [linea m | m <- [1..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. (Problema 9 del Proyecto Euler). Una terna pitagórica
-- es una terna de números naturales (a,b,c) tal que a<b<c y
-- a^2+b^2=c^2. Por ejemplo (3,4,5) es una terna pitagórica.
--
-- Definir la función
--    ternasPitagoricas :: Integer -> [[Integer]]
-- tal que (ternasPitagoricas x) es la lista de las ternas pitagóricas
-- cuya suma es x. Por ejemplo,
--    ternasPitagoricas 12  ==  [(3,4,5)]
--    ternasPitagoricas 60  ==  [(10,24,26),(15,20,25)]
```

```haskell
-- ---------------------------------------------------------------------

ternasPitagoricas :: Integer -> [(Integer,Integer,Integer)]
ternasPitagoricas x = [(a,b,c) | a <- [1..x],
                                 b <- [a+1..x],
                                 c <- [x-a-b],
                                 a^2 + b^2 == c^2]

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Definir la constante euler9 tal que euler9 es producto
-- abc donde (a,b,c) es la única terna pitagórica tal que a+b+c=1000.
-- Calcular el valor de euler9.
-- ---------------------------------------------------------------------

euler9 = a*b*c
    where (a,b,c) = head (ternasPitagoricas 1000)

-- El cálculo del valor de euler9 es
--    *Main> euler9
--    31875000
```

Relación 7

Definiciones por comprensión: El cifrado César

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En el tema 5, cuyas transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-5.pdf
-- se estudió, como aplicación de las definiciones por comprennsión, el
-- cifrado César. El objetivo de esta relación es modificar el programa
-- de cifrado César para que pueda utilizar también letras
-- mayúsculas. Por ejemplo,
--    *Main> descifra "Ytit Ufwf Sfif"
--    "Todo Para Nada"
-- Para ello, se propone la modificación de las funciones
-- correspondientes del tema 5.

-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares                                 --
-- ---------------------------------------------------------------------

import Data.Char

-- (minuscula2int c) es el entero correspondiente a la letra minúscula
-- c. Por ejemplo,
--    minuscula2int 'a'  ==  0
--    minuscula2int 'd'  ==  3
```

```haskell
--    minuscula2int 'z'  ==  25
minuscula2int :: Char -> Int
minuscula2int c = ord c - ord 'a'

-- (mayuscula2int c) es el entero correspondiente a la letra mayúscula
-- c. Por ejemplo,
--    mayuscula2int 'A'  ==  0
--    mayuscula2int 'D'  ==  3
--    mayuscula2int 'Z'  ==  25
mayuscula2int :: Char -> Int
mayuscula2int c = ord c - ord 'A'

-- (int2minuscula n) es la letra minúscula correspondiente al entero
-- n. Por ejemplo,
--    int2minuscula 0   ==  'a'
--    int2minuscula 3   ==  'd'
--    int2minuscula 25  ==  'z'
int2minuscula :: Int -> Char
int2minuscula n = chr (ord 'a' + n)

-- (int2mayuscula n) es la letra minúscula correspondiente al entero
-- n. Por ejemplo,
--    int2mayuscula 0   ==  'A'
--    int2mayuscula 3   ==  'D'
--    int2mayuscula 25  ==  'Z'
int2mayuscula :: Int -> Char
int2mayuscula n = chr (ord 'A' + n)

-- (desplaza n c) es el carácter obtenido desplazando n caracteres el
-- carácter c. Por ejemplo,
--    desplaza   3  'a'  ==  'd'
--    desplaza   3  'y'  ==  'b'
--    desplaza (-3) 'd'  ==  'a'
--    desplaza (-3) 'b'  ==  'y'
--    desplaza   3  'A'  ==  'D'
--    desplaza   3  'Y'  ==  'B'
--    desplaza (-3) 'D'  ==  'A'
--    desplaza (-3) 'B'  ==  'Y'
desplaza :: Int -> Char -> Char
desplaza n c
```

```
    | elem c ['a'..'z'] = int2minuscula ((minuscula2int c+n) `mod` 26)
    | elem c ['A'..'Z'] = int2mayuscula ((mayuscula2int c+n) `mod` 26)
    | otherwise         = c

-- (codifica n xs) es el resultado de codificar el texto xs con un
-- desplazamiento n. Por ejemplo,
--    *Main> codifica   3  "En Todo La Medida"
--    "Hq Wrgr Od Phglgd"
--    *Main> codifica (-3) "Hq Wrgr Od Phglgd"
--    "En Todo La Medida"
codifica :: Int -> String -> String
codifica n xs = [desplaza n x | x <- xs]

-- tabla es la lista de la frecuencias de las letras en castellano, Por
-- ejemplo, la frecuencia de la 'a' es del 12.53%, la de la 'b' es
-- 1.42%.
tabla :: [Float]
tabla = [12.53, 1.42, 4.68, 5.86, 13.68, 0.69, 1.01,
         0.70, 6.25, 0.44, 0.01,  4.97, 3.15, 6.71,
         8.68, 2.51, 0.88, 6.87,  7.98, 4.63, 3.93,
         0.90, 0.02, 0.22, 0.90,  0.52]

-- (porcentaje n m) es el porcentaje de n sobre m. Por ejemplo,
--    porcentaje 2 5  ==  40.0
porcentaje :: Int -> Int -> Float
porcentaje n m = (fromIntegral n / fromIntegral m) * 100

-- (letras xs) es la cadena formada por las letras de la cadena xs. Por
-- ejemplo,
--    letras "Esto Es Una Prueba"  ==  "EstoEsUnaPrueba"
letras :: String -> String
letras xs = [x | x <- xs, elem x (['a'..'z']++['A'..'Z'])]

-- (ocurrencias x xs) es el número de veces que ocurre el carácter x en
-- la cadena xs. Por ejemplo,
--    ocurrencias 'a' "Salamanca"  ==  4
ocurrencias :: Char -> String -> Int
ocurrencias x xs = length [x' | x' <- xs, x == x']

-- (frecuencias xs) es la frecuencia de cada una de las letras de la
```

```haskell
-- cadena xs. Por ejemplo,
--    *Main> frecuencias "En Todo La Medida"
--    [14.3,0,0,21.4,14.3,0,0,0,7.1,0,0,7.1,
--     7.1,7.1,14.3,0,0,0,0,7.1,0,0,0,0,0,0]
frecuencias :: String -> [Float]
frecuencias xs =
    [porcentaje (ocurrencias x xs') n | x <- ['a'..'z']]
    where xs' = [toLower x | x <- xs]
          n   = length (letras xs)

-- (chiCuad os es) es la medida chi cuadrado de las distribuciones os y
-- es. Por ejemplo,
--    chiCuad [3,5,6] [3,5,6]  ==  0.0
--    chiCuad [3,5,6] [5,6,3]  ==  3.9666667
chiCuad :: [Float] -> [Float] -> Float
chiCuad os es = sum [((o-e)^2)/e | (o,e) <- zip os es]

-- (rota n xs) es la lista obtenida rotando n posiciones los elementos
-- de la lista xs. Por ejemplo,
--    rota 2 "manolo"  ==  "noloma"
rota :: Int -> [a] -> [a]
rota n xs = drop n xs ++ take n xs

-- (descifra xs) es la cadena obtenida descodificando la cadena xs por
-- el anti-desplazamiento que produce una distribución de letras con la
-- menor deviación chi cuadrado respecto de la tabla de distribución de
-- las letras en castellano. Por ejemplo,
--    *Main> codifica 5 "Todo Para Nada"
--    "Ytit Ufwf Sfif"
--    *Main> descifra "Ytit Ufwf Sfif"
--    "Todo Para Nada"
descifra :: String -> String
descifra xs =  codifica (-factor) xs
 where
  factor = head (posiciones (minimum tabChi) tabChi)
  tabChi = [chiCuad (rota n tabla') tabla | n <- [0..25]]
  tabla' = frecuencias xs

posiciones :: Eq a => a -> [a] -> [Int]
posiciones x xs =
```

```
    [i | (x',i) <- zip xs [0..], x == x']
```

Relación 8

Definiciones por recursión

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En esta relación se presentan ejercicios con definiciones por
-- recursión correspondientes al tema 6 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-6.pdf
-- En concreto, se estudian funciones para calcular
-- * la potencia entera,
-- * la conjunción de una lista,
-- * la concatenación de una lista de listas,
-- * la creación de listas mediante repeticiones de un elemento,
-- * el elemento n-ésimo de una lista,
-- * la pertenencia de un elemento a una lista,
-- * la mezcla de dos listas ordenadas,
-- * el carácter ordenado de una lista,
-- * el borrado de una ocurrencia de un elemento en una lista,
-- * si una lista es una permutación de otra,
-- * la división de una lista es dos partes casi iguales,
-- * la ordenación por mezcla,
-- * la suma de una lista de números,
-- * el segmento inicial de una lista,
-- * el último elemento de una lista,
-- * el máximo común divisor mediante el algoritmo de Euclides y
-- * la solución del problema 5 del proyecto Euler.
```

```haskell
-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares                                 --
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List (sort, delete)

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir por recursión la función
--    potencia :: Integer -> Integer -> Integer
-- tal que (potencia x n) es x elevado al número natural n. Por ejemplo,
--    potencia 2 3  ==  8
-- ---------------------------------------------------------------------

potencia :: Integer -> Integer -> Integer
potencia m 0     = 1
potencia m (n+1) = m*(potencia m n)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Mostrar cómo se evalúa (potencia 2 3).
-- ---------------------------------------------------------------------

-- El cálculo es
--      potencia 2 3
--    = 2 * (potencia 2 2)              { def. de potencia }
--    = 2 * (2 * (potencia 2 1))        { def. de potencia }
--    = 2 * (2 * (2 * (potencia 2 0)))  { def. de potencia }
--    = 2 * (2 * (2 * 1))               { def. de potencia }
--    = 8                               { def. de * }

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comprobar con QuickCheck que la función potencia es
-- equivalente a la predefinida (^).
-- ---------------------------------------------------------------------

-- La propiedad es
prop_potencia :: Integer -> Integer -> Bool
prop_potencia x n =
    potencia x n' == x^n'
    where n' = abs n
```

```haskell
-- La comprobación es
--    *Main> quickCheck prop_potencia
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir por recursión la función
--    and' :: [Bool] -> Bool
-- tal que (and' xs) se verifica si todos los elementos de xs son
-- verdadero. Por ejemplo,
--    and' [1+2 < 4, 2:[3] == [2,3]]  ==  True
--    and' [1+2 < 3, 2:[3] == [2,3]]  ==  False
-- ---------------------------------------------------------------------

and' :: [Bool] -> Bool
and' []     = True
and' (b:bs) = b && and' bs

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Comprobar con QuickCheck que and' es equivalente a
-- and.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_and :: [Bool] -> Bool
prop_and xs = and' xs == and xs

-- La comprobación es
--    *Main> quickCheck prop_and
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir por recursión la función
--    concat' :: [[a]] -> [a]
-- tal que (concat' xss) es la lista obtenida concatenando las listas de
-- xss. Por ejemplo,
--    concat [[1..3],[5..7],[8..10]]  ==  [1,2,3,5,6,7,8,9,10]
-- ---------------------------------------------------------------------

concat' :: [[a]] -> [a]
```

```haskell
concat' []       = []
concat' (xs:xss) = xs ++ concat' xss

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con QuickCheck que conca' es equivalente a
-- concat.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_concat :: [[Int]] -> Bool
prop_concat xss = concat' xss == concat xss

-- La comprobación es
--    *Main> quickCheck prop_concat
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir por recursión la función
--    replicate' :: Int -> a -> [a]
-- tal que (replicate' n x) es la lista formado por n copias del
-- elemento x. Por ejemplo,
--    replicate' 3 2  ==  [2,2,2]
-- ---------------------------------------------------------------------

replicate' :: Int -> a -> [a]
replicate' 0 _     = []
replicate' (n+1) x = x : replicate' n x

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Comprobar con QuickCheck que replicate' es
-- equivalente a replicate.
-- ---------------------------------------------------------------------

-- La propiedad (limitada a 100 por eficiencia) es
prop_replicate :: Eq a => Int -> a -> Bool
prop_replicate n x =
    replicate' n' x == replicate n' x
    where n' = n `mod` 100

-- La comprobación es
```

```
--    *Main> quickCheck prop_replicate
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Definir por recursión la función
--    selecciona :: [a] -> Int -> a
-- tal que (selecciona xs n) es el n-ésimo elemento de xs. Por ejemplo,
--    selecciona [2,3,5,7] 2  ==  5
-- ---------------------------------------------------------------------

selecciona :: [a] -> Int -> a
selecciona (x:_)  0     = x
selecciona (_:xs) (n+1) = selecciona xs n

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Comprobar con quickCheck que selecciona es
-- equivalente a (!!).
-- ---------------------------------------------------------------------

-- La propiedad sin restricciones es
prop_selecciona_1 :: Eq a => [a] -> Int -> Bool
prop_selecciona_1 xs n =
    selecciona xs n == xs !! n

-- Esta propiedad no se verifica
--    *Main> quickCheck prop_selecciona_1
--    *** Failed! Non-exhaustive patterns in function selecciona'
--    []
--    0

-- La propiedad con restricciones es
prop_selecciona_2 :: Eq a => [a] -> Int -> Property
prop_selecciona_2 xs n =
    0 <= n && n < length xs ==>
    selecciona xs n == xs !! n

-- La propiedad se verifica
--    *Main> quickCheck prop_selecciona_2
--    *** Gave up! Passed only 12 tests.
-- pero sólo 12 de las pruebas cumplen las condiciones.
```

```haskell
-- Otra forma de enunciar la propiedad es
prop_selecciona_3 :: Eq a => [a] -> Int -> Property
prop_selecciona_3 xs n =
    not (null xs) ==> selecciona xs n' == xs !! n'
    where n' = n `mod` (length xs)

-- La comprobación es
--    *Main> quickCheck prop_selecciona_3
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 6.1. Definir por recursión la función
--    elem' :: Eq a => a -> [a] -> Bool
-- tal que (elem' x xs) se verifica si x pertenece a la lista xs. Por
-- ejemplo,
--    elem' 3 [2,3,5]  ==  True
--    elem' 4 [2,3,5]  ==  False
-- ---------------------------------------------------------------------

elem' :: Eq a => a -> [a] -> Bool
elem' x []                = False
elem' x (y:ys) | x == y   = True
               | otherwise = elem' x ys

-- ---------------------------------------------------------------------
-- Ejercicio 6.2. Comprobar con QuickCheck que elem' es equivalente a
-- elem.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_elem :: Eq a => a -> [a] -> Bool
prop_elem x xs = elem' x xs == elem x xs

-- La comprobación es
--    *Main> quickCheck prop_elem
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 7.1. Definir por recursión la función
```

```haskell
--    mezcla :: Ord a => [a] -> [a] -> [a]
-- tal que (mezcla xs ys) es la lista obtenida mezclando las listas
-- ordenadas xs e ys. Por ejemplo,
--    mezcla [2,5,6] [1,3,4]  ==  [1,2,3,4,5,6]
-- ---------------------------------------------------------------------

mezcla :: Ord a => [a] -> [a] -> [a]
mezcla []     ys                = ys
mezcla xs     []                = xs
mezcla (x:xs) (y:ys) | x <= y    = x : mezcla xs (y:ys)
                     | otherwise = y : mezcla (x:xs) ys

-- ---------------------------------------------------------------------
-- Ejercicio 7.2. Definir por recursión la función
--    ordenada :: Ord a => [a] -> Bool
-- tal que (ordenada xs) se verifica si xs es una lista ordenada. Por
-- ejemplo,
--    ordenada [2,3,5]  ==  True
--    ordenada [2,5,3]  ==  False
-- ---------------------------------------------------------------------

ordenada :: Ord a => [a] -> Bool
ordenada []       = True
ordenada [_]      = True
ordenada (x:y:xs) = x <= y && ordenada (y:xs)

-- ---------------------------------------------------------------------
-- Ejercicio 7.3. Comprobar con QuickCheck que la mezcla de dos listas
-- ordenadas es una lista ordenada.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_mezcla_ordenada :: [Int] -> [Int] -> Property
prop_mezcla_ordenada xs ys =
    ordenada xs && ordenada ys ==> ordenada (mezcla xs ys)

-- La comprobación es
--    *Main> quickCheck prop_mezcla_ordenada
--    +++ OK, passed 100 tests.
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 8.1. Definir la función
--    borra :: Eq a => a -> [a] -> [a]
-- tal que (borra x xs) es la lista obtenida borrando una ocurrencia de
-- x en la lista xs. Por ejemplo,
--    borra 1 [1,2,1]  ==  [2,1]
--    borra 3 [1,2,1]  ==  [1,2,1]
-- ---------------------------------------------------------------------

borra :: Eq a => a -> [a] -> [a]
borra x []                 = []
borra x (y:ys) | x == y    = ys
               | otherwise = y : borra x ys

-- ---------------------------------------------------------------------
-- Ejercicio 8.2. Comprobar con QuickCheck que borra es equivalente a
-- la función delete de la librería List.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_borra x xs =
    borra x xs == delete x xs

-- La comprobación es
--    *Main> quickCheck prop_borra
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 9.1. Definir la función
--    esPermutacion :: Eq a => [a] -> [a] -> Bool
-- tal que (esPermutacion xs ys) se verifica si xs es una permutación de
-- ys. Por ejemplo,
--    esPermutacion [1,2,1] [2,1,1]  ==  True
--    esPermutacion [1,2,1] [1,2,2]  ==  False
-- ---------------------------------------------------------------------

-- La definición es
esPermutacion :: Eq a => [a] -> [a] -> Bool
esPermutacion []     []     = True
esPermutacion []     (y:ys) = False
```

```haskell
esPermutacion (x:xs) ys     = elem x ys && esPermutacion xs (borra x ys)

-- ---------------------------------------------------------------------
-- Ejercicio 9.2. Comprobar con QuickCheck que la inversa de una lista
-- es una permutación de la lista.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_InversaEsPermutacion :: [Int] -> Bool
prop_InversaEsPermutacion xs =
    esPermutacion (reverse xs) xs

-- La  comprobación es
--    Main> quickCheck prop_InversaEsPermutacion
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 9.3. Comprobar con QuickCheck que la mezcla de dos listas
-- es una permutación de su unión.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_mezcla_permutacion xs ys =
    esPermutacion (mezcla xs ys) (xs++ys)

-- La comprobación es
--    quickCheck prop_mezcla_permutacion
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 10.1. Definir la función
--    mitades :: [a] -> ([a],[a])
-- tal que (mitades xs) es el par formado por las dos mitades en que se
-- divide xs tales que sus longitudes difieren como máximo en uno. Por
-- ejemplo,
--    mitades [2,3,5,7,9]  ==  ([2,3],[5,7,9])
-- ---------------------------------------------------------------------

mitades :: [a] -> ([a],[a])
mitades xs = splitAt (length xs `div` 2) xs
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 10.2. Definir la función
--    ordMezcla :: Ord a => [a] -> [a]
-- tal que (ordMezcla xs) es la lista obtenida ordenado xs por mezcla
-- (es decir, considerando que la lista vacía y las listas unitarias
-- están ordenadas y cualquier otra lista se ordena mezclando las dos
-- listas que resultan de ordenar sus dos mitades por separado). Por
-- ejemplo,
--    ordMezcla [5,2,3,1,7,2,5]  =>  [1,2,2,3,5,5,7]
-- ---------------------------------------------------------------------

ordMezcla :: Ord a => [a] -> [a]
ordMezcla []  = []
ordMezcla [x] = [x]
ordMezcla xs  = mezcla (ordMezcla ys) (ordMezcla zs)
                where (ys,zs) = mitades xs

-- ---------------------------------------------------------------------
-- Ejercicio 10.3. Comprobar con QuickCheck que la ordenación por mezcla
-- de una lista es una lista ordenada.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_ordMezcla_ordenada :: [Int] -> Bool
prop_ordMezcla_ordenada xs = ordenada (ordMezcla xs)

-- La comprobación es
--    *Main> quickCheck prop_ordMezcla_ordenada
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 10.4. Comprobar con QuickCheck que la ordenación por mezcla
-- de una lista es una permutación de la lista.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_ordMezcla_pemutacion :: [Int] -> Bool
prop_ordMezcla_pemutacion xs = esPermutacion (ordMezcla xs) xs
```

```haskell
-- La comprobación es
--    *Main> quickCheck prop_ordMezcla_permutacion
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 11.1. Definir por recursión la función
--    sum' :: [Int] -> Int
-- tal que (sum' xs) es la suma de los números de xs. Por ejemplo,
--    sum' [2,3,5]  =>  10
-- ---------------------------------------------------------------------

sum' :: [Int] -> Int
sum' []     = 0
sum' (x:xs) = x + sum' xs

-- ---------------------------------------------------------------------
-- Ejercicio 11.2. Comprobar con QuickCheck que sum' es equivalente a
-- la función sum.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_sum xs =
    sum' xs == sum xs

-- La comprobación es
--    *Main> quickCheck prop_sum
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 12.1. Definir por recursión la función
--    take' :: Int -> [a] -> [a]
-- tal que (take' n xs) es la lista de los n primeros elementos de
-- xs. Por ejemplo,
--    take' 3 [4..12]  =>  [4,5,6]
-- ---------------------------------------------------------------------

take' :: Int -> [a] -> [a]
take' 0 _          = []
take' (n+1) []     = []
take' (n+1) (x:xs) = x : take' n xs
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 12.2. Comprobar con QuickCheck que take' es equivalente a
-- la función take.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_take n xs =
    take' n' xs == take n' xs
    where n' = abs n

-- La comprobación es
--    *Main> quickCheck prop_take
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 13.1. Definir por recursión la función
--    last' :: [a] -> a
-- tal que (last xs) es el último elemento de xs. Por ejemplo,
--    last' [2,3,5]  =>  5
-- ---------------------------------------------------------------------

last' :: [a] -> a
last' [x]    = x
last' (_:xs) = last' xs

-- ---------------------------------------------------------------------
-- Ejercicio 13.2. Comprobar con QuickCheck que last' es equivalente a
-- la función last.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_last xs =
    not (null xs) ==> last' xs == last xs

-- La comprobación es
--    *Main> quickCheck prop_last
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
```

```haskell
-- Ejercicio 14.1. Dados dos números naturales, a y b, es posible
-- calcular su máximo común divisor mediante el Algoritmo de
-- Euclides. Este algoritmo se puede resumir en la siguiente fórmula:
--    mcd(a,b) = a,                    si b = 0
--             = mcd (b, a módulo b), si b > 0
--
-- Definir la función
--    mcd :: Integer -> Integer -> Integer
-- tal que (mcd a b) es el máximo común divisor de a y b calculado
-- mediante el algoritmo de Euclides. Por ejemplo,
--    mcd 30 45  ==  15
-- ---------------------------------------------------------------------

-- La definición es
mcd :: Integer -> Integer -> Integer
mcd a 0 = a
mcd a b = mcd b (a `mod` b)

-- ---------------------------------------------------------------------
-- Ejercicio 14.2. Comprobar con QuickCheck que mcd es equivalente a
-- la función gcd.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_mcd_equiv a b =
    a>0 && b>0 ==> mcd a b == gcd a b

-- La comprobación es
--    *Main> quickCheck prop_mcd
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 14.2. Definir y comprobar la propiedad prop_mcd según la
-- cual el máximo común divisor de dos números a y b (ambos mayores que
-- 0) es siempre mayor o igual que 1 y además es menor o igual que el
-- menor de los números a y b.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_mcd a b =
```

```
    a>0 && b>0 ==> m>=1 && m <= min a b
    where m = mcd a b

-- La comprobación es
--    Main> quickCheck prop_mcd
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 14.3. Teniendo en cuenta que buscamos el máximo común
-- divisor de a y b, sería razonable pensar que el máximo común divisor
-- siempre sería igual o menor que la mitad del máximo de a y b. Definir
-- esta propiedad y comprobarla.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_mcd_div a b =
    a > 0 && b > 0 ==> mcd a b <= (max a b) `div` 2

-- Al verificarla, se obtiene
--    Main> quickCheck prop_mcd_div
--    Falsifiable, after 0 tests:
--    3
--    3
-- que la refuta. Pero si la modificamos añadiendo la hipótesis que los
-- números son distintos,
prop_mcd_div' a b =
    a > 0 && b > 0 && a /= b ==> mcd a b <= (max a b) `div` 2

-- entonces al comprobarla
--    Main> quickCheck prop_mcd_div'
--    OK, passed 100 tests.
-- obtenemos que se verifica.

-- ---------------------------------------------------------------------
-- Ejercicio 15 (Problema 5 del proyecto Euler) El problema se encuentra
-- en http://goo.gl/L5bb y consiste en calcular el menor número
-- divisible por los números del 1 al 20. Lo resolveremos mediante los
-- distintos apartados de este ejercicio.
-- ---------------------------------------------------------------------
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 15.1. Definir por recursión la función
--    menorDivisible :: Integer -> Integer -> Integer
-- tal que (menorDivisible a b) es el menor número divisible por los
-- números del a al b. Por ejemplo,
--    menorDivisible 2 5  ==  60
-- Indicación: Usar la función lcm tal que (lcm x y) es el mínimo común
-- múltiplo de x e y.
-- ---------------------------------------------------------------------

menorDivisible :: Integer -> Integer -> Integer
menorDivisible  a b
    | a == b    = a
    | otherwise = lcm a (menorDivisible (a+1) b)

-- ---------------------------------------------------------------------
-- Ejercicio 15.2. Definir la función
--    divisiblePorTodos :: Integer -> Integer -> Integer -> Bool
-- tal que (divisiblePorTodos x a b) se verifica si x es divisible por
-- todos los números entre a y b. Por ejemplo,
--    divisiblePorTodos 60 2 5  ==  True
--    divisiblePorTodos 50 2 5  ==  False
-- ---------------------------------------------------------------------

divisiblePorTodos :: Integer -> Integer -> Integer -> Bool
divisiblePorTodos x a b = and [mod x y == 0 | y <- [a..b]]

-- ---------------------------------------------------------------------
-- Ejercicio 15.3. Comprobar con QuickCheck que si a y b son dos números
-- enteros positivos tales que a <= b, entonces (menorDivisible a b) es
-- divisible por todos los números entre a y b.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_menorDivisible a b =
    a'<=b' ==> divisiblePorTodos m a' b'
    where m = menorDivisible a' b'
          a' = (abs a) + 1
          b' = (abs b) + 1
```

```haskell
-- La comprobación es
--    *Main> quickCheck prop_menorDivisible
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 15.4. Definir la constante
--    euler5 :: Integer
-- tal que euler5 es el menor número divisible por los números del 1 al
-- 20 y calcular su valor.
-- ---------------------------------------------------------------------

-- La definición es
euler5 :: Integer
euler5 = menorDivisible 1 20

-- El cálculo es
--    *Main> euler5
--    232792560
```

Relación 9

Definiciones por recursión y por comprensión (1)

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En esta relación se presentan ejercicios con dos definiciones (una
-- por recursión y otra por comprensión) y la comprobación de la
-- equivalencia de las dos definiciones con QuickCheck. Los ejercicios
-- corresponden a los temas 5 y 6 cuyas transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-5.pdf
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-6.pdf
-- En concreto, se estudian funciones para calcular
-- * la suma de los cuadrados de los n primeros números,
-- * el número de bloques de escaleras triangulares,
-- * la suma de los cuadrados de los impares entre los n primeros números,
-- * la lista de las cifras de un número,
-- * la suma de las cifras de un número,
-- * la pertenencia a las cifras de un número,
-- * el número de cifras de un número,
-- * el número correspondiente a las cifras de un número,
-- * la concatenación de dos números,
-- * la primera cifra de un número,
-- * la última cifra de un número,
-- * el número con las cifras invertidas,
-- * si un número es capicúa,
-- * el exponente de la mayor potencia de un número que divide a otro,
```

```
-- * la lista de los factores de un número,
-- * si un número es primo,
-- * la lista de los factores primos de un número,
-- * la factorización de un número,
-- * la expansion de la factorización de un número,
-- * el número de pasos para resolver el problema de las torres de Hanoi y
-- * la solución del problema 16 del proyecto Euler.

-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares                                 --
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por recursión; la función
--    sumaCuadrados :: Integer -> Integer
-- tal que (sumaCuadrados n) es la suma de los cuadrados de los números
-- de 1 a n. Por ejemplo,
--    sumaCuadrados 4  ==  30
-- ---------------------------------------------------------------------

sumaCuadrados :: Integer -> Integer
sumaCuadrados 0     = 0
sumaCuadrados (n+1) = sumaCuadrados n + (n+1)*(n+1)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickCheck si sumaCuadrados n es igual a
-- n(n+1)(2n+1)/6.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_SumaCuadrados n =
  n >= 0 ==>
    sumaCuadrados n == n * (n+1) * (2*n+1) `div` 6

-- La comprobación es
--    Main> quickCheck prop_SumaCuadrados
--    OK, passed 100 tests.
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir, por comprensión, la función
--    sumaCuadrados' :: Integer --> Integer
-- tal que (sumaCuadrados' n) es la suma de los cuadrados de los números
-- de 1 a n. Por ejemplo,
--    sumaCuadrados' 4  ==  30
-- ---------------------------------------------------------------------

sumaCuadrados' :: Integer -> Integer
sumaCuadrados' n = sum [x^2 | x <- [1..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Comprobar con QuickCheck que las funciones
-- sumaCuadrados y sumaCuadrados' son equivalentes sobre los números
-- naturales.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_sumaCuadrados n =
    n >= 0 ==> sumaCuadrados n == sumaCuadrados' n

-- La comprobación es
--    *Main> quickCheck prop_sumaCuadrados
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Se quiere formar una escalera con bloques cuadrados,
-- de forma que tenga un número determinado de escalones. Por ejemplo,
-- una escalera con tres escalones tendría la siguiente forma:
--        XX
--      XXXX
--    XXXXXX
-- Definir, por recursión, la función
--    numeroBloques :: Integer -> Integer
-- tal que (numeroBloques n) es el número de bloques necesarios para
-- construir una escalera con n escalones. Por ejemplo,
--    numeroBloques 1   == 2
--    numeroBloques 3   == 12
--    numeroBloques 10  == 110
```

```haskell
-- ---------------------------------------------------------------------

numeroBloques :: Integer -> Integer
numeroBloques 0     = 0
numeroBloques (n+1) = 2*(n+1) + numeroBloques n

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, por comprensión, la función
--    numeroBloques' :: Integer -> Integer
-- tal que (numeroBloques' n) es el número de bloques necesarios para
-- construir una escalera con n escalones. Por ejemplo,
--    numeroBloques' 1   == 2
--    numeroBloques' 3   == 12
--    numeroBloques' 10  == 110
-- ---------------------------------------------------------------------

numeroBloques' :: Integer -> Integer
numeroBloques' n = sum [2*x | x <- [1..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Comprobar con QuickCheck que (numeroBloques' n) es
-- igual a n+n^2.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_numeroBloques n =
    n >0 ==> numeroBloques' n == n+n^2

-- La comprobación es
--    *Main> quickCheck prop_numeroBloques
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir, por recursión, la función
--    sumaCuadradosImparesR :: Integer -> Integer
-- tal que (sumaCuadradosImparesR n) es la suma de los cuadrados de los
-- números impares desde 1 hasta n.
--    sumaCuadradosImparesR 1  ==  1
--    sumaCuadradosImparesR 7  ==  84
--    sumaCuadradosImparesR 4  ==  10
```

```haskell
-- ---------------------------------------------------------------------

sumaCuadradosImparesR :: Integer -> Integer
sumaCuadradosImparesR 1 = 1
sumaCuadradosImparesR n
    | odd n     = n^2 + sumaCuadradosImparesR (n-1)
    | otherwise = sumaCuadradosImparesR (n-1)

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir, por comprensión, la función
--    sumaCuadradosImparesC :: Integer -> Integer
-- tal que (sumaCuadradosImparesC n) es la suma de los cuadrados de los
-- números impares desde 1 hasta n.
--    sumaCuadradosImparesC 1  ==  1
--    sumaCuadradosImparesC 7  ==  84
--    sumaCuadradosImparesC 4  ==  10
-- ---------------------------------------------------------------------

sumaCuadradosImparesC :: Integer -> Integer
sumaCuadradosImparesC n = sum [x^2 | x <- [1..n], odd x]

-- Otra definición más simple es
sumaCuadradosImparesC' :: Integer -> Integer
sumaCuadradosImparesC' n = sum [x^2 | x <- [1,3..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir, por recursión, la función
--    cifrasR :: Integer -> [Int]
-- tal que (cifrasR n) es la lista de los cifras del número n. Por
-- ejemplo,
--    cifrasR 320274  ==  [3,2,0,2,7,4]
-- ---------------------------------------------------------------------

cifrasR :: Integer -> [Integer]
cifrasR n = reverse (cifrasR' n)

cifrasR' n
    | n < 10    = [n]
    | otherwise = (n `rem` 10) : cifrasR' (n `div` 10)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir, por comprensión, la función
--    cifras :: Integer -> [Int]
-- tal que (cifras n) es la lista de los cifras del número n. Por
-- ejemplo,
--    cifras 320274  ==  [3,2,0,2,7,4]
-- Indicación: Usar las funciones show y read.
-- ---------------------------------------------------------------------

cifras :: Integer -> [Integer]
cifras n = [read [x] | x <- show n]

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Comprobar con QuickCheck que las funciones cifrasR y
-- cifras son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_cifras n =
    n >= 0 ==>
    cifrasR n == cifras n

-- La comprobación es
--    *Main> quickCheck prop_cifras
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Definir, por recursión, la función
--    sumaCifrasR :: Integer -> Integer
-- tal que (sumaCifrasR n) es la suma de las cifras de n. Por ejemplo,
--    sumaCifrasR 3     ==  3
--    sumaCifrasR 2454  == 15
--    sumaCifrasR 20045 == 11
-- ---------------------------------------------------------------------

sumaCifrasR :: Integer -> Integer
sumaCifrasR n
    | n < 10    = n
    | otherwise = n `rem` 10 + sumaCifrasR (n `div` 10)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Definir, sin usar recursión, la función
--    sumaCifrasNR :: Integer -> Integer
-- tal que (sumaCifrasNR n) es la suma de las cifras de n. Por ejemplo,
--    sumaCifrasNR 3     ==  3
--    sumaCifrasNR 2454  == 15
--    sumaCifrasNR 20045 == 11
-- ---------------------------------------------------------------------

sumaCifrasNR :: Integer -> Integer
sumaCifrasNR n = sum (cifras n)

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Comprobar con QuickCheck que las funciones sumaCifrasR
-- y sumaCifrasNR son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_sumaCifras n =
    n >= 0 ==>
    sumaCifrasR n == sumaCifrasNR n

-- La comprobación es
--    *Main> quickCheck prop_sumaCifras
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    esCifra :: Integer -> Integer -> Bool
-- tal que (esCifra x n) se verifica si x es una cifra de n. Por
-- ejemplo,
--    esCifra 4 1041  ==  True
--    esCifra 3 1041  ==  False
-- ---------------------------------------------------------------------

esCifra :: Integer -> Integer -> Bool
esCifra x n = elem x (cifras n)

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
```

```haskell
--    numeroDeCifras :: Integer -> Integer
-- tal que (numeroDeCifras x) es el número de cifras de x. Por ejemplo,
--    numeroDeCifras 34047  ==  5
-- ---------------------------------------------------------------------

numeroDeCifras :: Integer -> Int
numeroDeCifras x = length (cifras x)

-- ---------------------------------------------------------------------
-- Ejercicio 7.1 Definir, por recursión, la función
--    listaNumeroR :: [Integer] -> Integer
-- tal que (listaNumeroR xs) es el número formado por las cifras xs. Por
-- ejemplo,
--    listaNumeroR [5]        == 5
--    listaNumeroR [1,3,4,7]  == 1347
--    listaNumeroR [0,0,1]    == 1
-- ---------------------------------------------------------------------

listaNumeroR :: [Integer] -> Integer
listaNumeroR xs = listaNumeroR' (reverse xs)

listaNumeroR' :: [Integer] -> Integer
listaNumeroR' [x]    = x
listaNumeroR' (x:xs) = x + 10 * (listaNumeroR' xs)

-- ---------------------------------------------------------------------
-- Ejercicio 7.2. Definir, por comprensión, la función
--    listaNumeroC :: [Integer] -> Integer
-- tal que (listaNumeroC xs) es el número formado por las cifras xs. Por
-- ejemplo,
--    listaNumeroC [5]        == 5
--    listaNumeroC [1,3,4,7]  == 1347
--    listaNumeroC [0,0,1]    == 1
-- ---------------------------------------------------------------------

listaNumeroC :: [Integer] -> Integer
listaNumeroC xs = sum [y*10^n | (y,n) <- zip (reverse xs) [0..]]

-- ---------------------------------------------------------------------
-- Ejercicio 8.1. Definir, por recursión, la función
```

```haskell
--    pegaNumerosR :: Integer -> Integer -> Integer
-- tal que (pegaNumerosR x y) es el número resultante de "pegar" los
-- números x e y. Por ejemplo,
--    pegaNumerosR 12 987   ==  12987
--    pegaNumerosR 1204 7   ==  12047
--    pegaNumerosR 100 100  ==  100100
-- ------------------------------------------------------------------

pegaNumerosR :: Integer -> Integer -> Integer
pegaNumerosR x y
    | y < 10    = 10*x+y
    | otherwise = 10 * pegaNumerosR x (y `div`10) + (y `mod` 10)

-- ------------------------------------------------------------------
-- Ejercicio 8.2. Definir, sin usar recursión, la función
--    pegaNumerosNR :: Integer -> Integer -> Integer
-- tal que (pegaNumerosNR x y) es el número resultante de "pegar" los
-- números x e y. Por ejemplo,
--    pegaNumerosNR 12 987   ==  12987
--    pegaNumerosNR 1204 7   ==  12047
--    pegaNumerosNR 100 100  ==  100100
-- ------------------------------------------------------------------

pegaNumerosNR :: Integer -> Integer -> Integer
pegaNumerosNR x y = listaNumeroC (cifras x ++ cifras y)

-- ------------------------------------------------------------------
-- Ejercicio 8.3. Comprobar con QuickCheck que las funciones
-- pegaNumerosR y pegaNumerosNR son equivalentes.
-- ------------------------------------------------------------------

-- La propiedad es
prop_pegaNumeros x y =
    x >= 0 && y >= 0 ==>
    pegaNumerosR x y == pegaNumerosNR x y

-- La comprobción es
--    *Main> quickCheck prop_pegaNumeros
--    +++ OK, passed 100 tests.
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 9.1. Definir, por recursión, la función
--    primeraCifraR :: Integer -> Integer
-- tal que (primeraCifraR n) es la primera cifra de n. Por ejemplo,
--    primeraCifraR 425  ==  4
-- ---------------------------------------------------------------------

primeraCifraR :: Integer -> Integer
primeraCifraR n
    | n < 10    = n
    | otherwise = primeraCifraR (n `div` 10)

-- ---------------------------------------------------------------------
-- Ejercicio 9.2. Definir, sin usar recursión, la función
--    primeraCifraNR :: Integer -> Integer
-- tal que (primeraCifraNR n) es la primera cifra de n. Por ejemplo,
--    primeraCifraNR 425  ==  4
-- ---------------------------------------------------------------------

primeraCifraNR :: Integer -> Integer
primeraCifraNR n = head (cifras n)

-- ---------------------------------------------------------------------
-- Ejercicio 9.3. Comprobar con QuickCheck que las funciones
-- primeraCifraR y primeraCifraNR son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_primeraCifra x =
    x >= 0 ==>
    primeraCifraR x == primeraCifraNR x

-- La comprobación es
--    *Main> quickCheck prop_primeraCifra
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    ultimaCifra :: Integer -> Integer
-- tal que (ultimaCifra n) es la última cifra de n. Por ejemplo,
```

```
--    ultimaCifra 425  ==  5
-- ---------------------------------------------------------------------

ultimaCifra :: Integer -> Integer
ultimaCifra n = n `rem` 10

-- ---------------------------------------------------------------------
-- Ejercicio 11.1. Definir la función
--    inverso :: Integer -> Integer
-- tal que (inverso n) es el número obtenido escribiendo las cifras de n
-- en orden inverso. Por ejemplo,
--    inverso 42578  ==  87524
--    inverso 203    ==    302
-- ---------------------------------------------------------------------

inverso :: Integer -> Integer
inverso n = listaNumeroC (reverse (cifras n))

-- ---------------------------------------------------------------------
-- Ejercicio 11.2. Definir, usando show y read, la función
--    inverso' :: Integer -> Integer
-- tal que (inverso' n) es el número obtenido escribiendo las cifras de n
-- en orden inverso'. Por ejemplo,
--    inverso' 42578  ==  87524
--    inverso' 203    ==    302
-- ---------------------------------------------------------------------

inverso' :: Integer -> Integer
inverso' n = read (reverse (show n))

-- ---------------------------------------------------------------------
-- Ejercicio 11.3. Comprobar con QuickCheck que las funciones
-- inverso e inverso' son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_inverso n =
    n >= 0 ==>
    inverso n == inverso' n
```

```haskell
-- La comprobación es
--    *Main> quickCheck prop_inverso
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    capicua :: Integer -> Bool
-- tal que (capicua n) se verifica si si las cifras que n son las mismas
-- de izquierda a derecha que de derecha a izquierda. Por ejemplo,
--    capicua 1234  =  False
--    capicua 1221  =  True
--    capicua 4     =  True
-- ---------------------------------------------------------------------

capicua :: Integer -> Bool
capicua n = n == inverso n

-- ---------------------------------------------------------------------
-- Ejercicio 13.1. Definir, por recursión, la función
--    mayorExponenteR :: Integer -> Integer -> Integer
-- tal que (mayorExponenteR a b) es el exponente de la mayor potencia de
-- a que divide b. Por ejemplo,
--    mayorExponenteR 2 8    ==  3
--    mayorExponenteR 2 9    ==  3
--    mayorExponenteR 5 100  ==  2
--    mayorExponenteR 2 60   ==  2
-- ---------------------------------------------------------------------

mayorExponenteR :: Integer -> Integer -> Integer
mayorExponenteR a b
    | mod b a /= 0 = 0
    | otherwise    = 1 + mayorExponenteR a (b `div` a)

-- ---------------------------------------------------------------------
-- Ejercicio 13.2. Definir, por recursión, la función
--    mayorExponenteC :: Integer -> Integer -> Integer
-- tal que (mayorExponenteC a b) es el exponente de la mayor potencia de
-- a que divide a b. Por ejemplo,
--    mayorExponenteC 2 8    ==  3
--    mayorExponenteC 2 9    ==  3
```

```haskell
--    mayorExponenteC 5 100  ==  2
--    mayorExponenteC 5 101  ==  0
-- ---------------------------------------------------------------------

mayorExponenteC :: Integer -> Integer -> Integer
mayorExponenteC a b = head [x-1 | x <- [0..], mod b (a^x) /= 0]

-- ---------------------------------------------------------------------
-- Ejercicio 14.1. Definir la función
--    factores :: Integer -> Integer
-- tal que (factores n) es la lista de los factores de n. Por ejemplo,
--    factores 60  ==  [1,2,3,4,5,6,10,12,15,20,30,60]
-- ---------------------------------------------------------------------

factores :: Integer -> [Integer]
factores n = [x | x <- [1..n], mod n x == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 14.2. Definir la función
--    primo :: Integer -> Bool
-- tal que (primo n) se verifica si n es primo. Por ejemplo,
--    primo 7  ==  True
--    primo 9  ==  False
-- ---------------------------------------------------------------------

primo :: Integer -> Bool
primo x = factores x == [1,x]

-- ---------------------------------------------------------------------
-- Ejercicio 14.3. Definir la función
--    factoresPrimos :: Integer -> [Integer]
-- tal que (factoresPrimos n) es la lista de los factores primos de
-- n. Por ejemplo,
--    factoresPrimos 60  ==  [2,3,5]
-- ---------------------------------------------------------------------

factoresPrimos :: Integer -> [Integer]
factoresPrimos n = [x | x <- factores n, primo x]

-- ---------------------------------------------------------------------
```

```haskell
-- Ejercicio 14.4. Definir la función
--    factorizacion :: Integer -> [(Integer,Integer)]
-- tal que (factorizacion n) es la factorización de n. Por ejemplo,
--    factorizacion 60  ==  [(2,2),(3,1),(5,1)]
-- ---------------------------------------------------------------------

factorizacion :: Integer -> [(Integer,Integer)]
factorizacion n = [(x,mayorExponenteR x n) | x <- factoresPrimos n]

-- ---------------------------------------------------------------------
-- Ejercicio 14.5. Definir, por recursión, la función
--    expansionR :: [(Integer,Integer)] -> Integer
-- tal que (expansionR xs) es la expansion de la factorización de
-- xs. Por ejemplo,
--    expansionR [(2,2),(3,1),(5,1)]  ==  60
-- ---------------------------------------------------------------------

expansionR :: [(Integer,Integer)] -> Integer
expansionR [] = 1
expansionR ((x,y):zs) = x^y * expansionR zs

-- ---------------------------------------------------------------------
-- Ejercicio 14.6. Definir, por comprensión, la función
--    expansionC :: [(Integer,Integer)] -> Integer
-- tal que (expansionC xs) es la expansion de la factorización de
-- xs. Por ejemplo,
--    expansionC [(2,2),(3,1),(5,1)]  ==  60
-- ---------------------------------------------------------------------

expansionC :: [(Integer,Integer)] -> Integer
expansionC xs = product [x^y | (x,y) <- xs]

-- ---------------------------------------------------------------------
-- Ejercicio 14.7. Definir la función
--    prop_factorizacion :: Integer -> Bool
-- tal que (prop_factorizacion n) se verifica si para todos número
-- natural x, menor o igual que n, se tiene que
-- (expansionC (factorizacion x)) es igual a x. Por ejemplo,
--    prop_factorizacion 100  ==  True
-- ---------------------------------------------------------------------
```

```haskell
prop_factorizacion n =
    and [expansionC (factorizacion x) == x | x <- [1..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 15. En un templo hindú se encuentran tres varillas de
-- platino. En una de ellas, hay 64 anillos de oro de distintos radios,
-- colocados de mayor a menor.
--
-- El trabajo de los monjes de ese templo consiste en pasarlos todos a
-- la tercera varilla, usando la segunda como varilla auxiliar, con las
-- siguientes condiciones:
--   * En cada paso sólo se puede mover un anillo.
--   * Nunca puede haber un anillo de mayor diámetro encima de uno de
--     menor diámetro.
-- La leyenda dice que cuando todos los anillos se encuentren en la
-- tercera varilla, será el fin del mundo.
--
-- Definir la función
--    numPasosHanoi :: Integer -> Integer
-- tal que (numPasosHanoi n) es el número de pasos necesarios para
-- trasladar n anillos. Por ejemplo,
--    numPasosHanoi 2   ==  3
--    numPasosHanoi 7   ==  127
--    numPasosHanoi 64  ==  18446744073709551615
-- ---------------------------------------------------------------------

-- Sean A, B y C las tres varillas. La estrategia recursiva es la
-- siguiente:
-- * Caso base (N=1): Se mueve el disco de A a C.
-- * Caso inductivo (N=M+1): Se mueven M discos de A a C. Se mueve el disco
--   de A a B. Se mueven M discos de C a B.
-- Por tanto,

numPasosHanoi :: Integer -> Integer
numPasosHanoi 1     = 1
numPasosHanoi (n+1) = 1 + 2 * numPasosHanoi n

-- ---------------------------------------------------------------------
-- Ejercicio 16. (Problema 16 del proyecto Euler) El problema se
```

```haskell
-- encuentra en http://goo.gl/4uWh y consiste en calcular la suma de las
-- cifras de 2^1000. Lo resolveremos mediante los distintos apartados de
-- este ejercicio.
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 16.1. Definir la función
--    euler16 :: Integer -> Integer
-- tal que (euler16 n) es la suma de las cifras de 2^n. Por ejemplo,
--    euler16 4  ==  7
-- ---------------------------------------------------------------------

euler16 :: Integer -> Integer
euler16 n = sumaCifrasNR (2^n)

-- ---------------------------------------------------------------------
-- Ejercicio 16.2. Calcular la suma de las cifras de 2^1000.
-- ---------------------------------------------------------------------

-- El cálculo es
--    *Main> euler16 1000
--    1366
```

Relación 10

Definiciones por recursión y por comprensión (2)

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En esta relación se presentan ejercicios con dos definiciones (una
-- por recursión y otra por comprensión) y la comprobación de la
-- equivalencia de las dos definiciones con QuickCheck. Además, se hace
-- la traza del cálculo de las definiciones recursivas. Los ejercicios
-- corresponden a los temas 5 y 6 cuyas transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-5.pdf
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-6.pdf
-- En concreto, se estudian funciones para calcular
-- * la lista de los cuadrados de una lista de números,
-- * la lista de los números impares de una lista de números,
-- * la lista de los cuadrados de los números impares de una lista,
-- * la suma de los cuadrados de los números impares de la lista,
-- * la lista de los números entre dos números,
-- * lista de las mitades de los números pares de una lista,
-- * la lista de los números en un rango dado,
-- * la suma de los números positivos de una lista,

-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares                                 --
-- ---------------------------------------------------------------------
```

```haskell
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir, por comprensión, la función
--    cuadradosC :: [Integer] -> [Integer]
-- tal que (cuadradosC xs) es la lista de los cuadrados de xs. Por
-- ejemplo,
--    cuadradosC [1,2,3]  ==  [1,4,9]
-- ---------------------------------------------------------------------

cuadradosC :: [Integer] -> [Integer]
cuadradosC xs = [x*x | x <- xs]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir, por recursión, la función
--    cuadradosR :: [Integer] -> [Integer]
-- tal que (cuadradosR xs) es la lista de los cuadrados de xs. Por
-- ejemplo,
--    cuadradosR [1,2,3]  ==  [1,4,9]
-- ---------------------------------------------------------------------

cuadradosR :: [Integer] -> [Integer]
cuadradosR []     = []
cuadradosR (x:xs) = x*x : cuadradosR xs

-- ---------------------------------------------------------------------
-- Ejercicio 3. Escribir el cálculo de (cuadradosR [1,2,3]).
-- ---------------------------------------------------------------------

{-
     cuadradosR [1,2,3]
   = cuadradosR (1:(2:(3:[])))
   = 1*1:(cuadradosR (2:(3:[])))
   = 1:(cuadradosR (2:(3:[])))
   = 1:(2*2:(cuadradosR (3:[])))
   = 1:(4:(cuadradosR (3:[])))
   = 1:(4:(3*3:(cuadradosR [])))
   = 1:(4:(9:(cuadradosR [])))
   = 1:(4:(9:[]))
   = [1,4,9]
```

```
-}

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir, por comprensión, la función
--    imparesC :: [Integer] -> [Integer]
-- tal que (imparesC xs) es la lista de los números impares de xs. Por
-- ejemplo,
--    imparesC [1,2,3]  ==  [1,3]
-- ---------------------------------------------------------------------

imparesC :: [Integer] -> [Integer]
imparesC xs = [x | x <- xs, odd x]

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir, por recursión, la función
--    imparesR :: [Integer] -> [Integer]
-- tal que (imparesR xs) es la lista de los números impares de xs. Por
-- ejemplo,
--    imparesR [1,2,3]  ==  [1,3]
-- ---------------------------------------------------------------------

imparesR :: [Integer] -> [Integer]
imparesR []                 = []
imparesR (x:xs) | odd x     = x : imparesR xs
                | otherwise = imparesR xs

-- ---------------------------------------------------------------------
-- Ejercicio 6. Escribir el cálculo de (imparesR [1,2,3]).
-- ---------------------------------------------------------------------

{-
     imparesR [1,2,3]
   = imparesR (1:(2:(3:[])))
   = 1:(imparesR (2:(3:[])))
   = 1:(imparesR (3:[]))
   = 1:(3:(imparesR []))
   = 1:(3:[])
   = [1,3]
-}
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir, por comprensión, la función
--    imparesCuadradosC :: [Integer] -> [Integer]
-- tal que (imparesCuadradosC xs) es la lista de los cuadrados de los
-- números impares de xs. Por ejemplo,
--    imparesCuadradosC [1,2,3]  ==  [1,9]
-- ---------------------------------------------------------------------

imparesCuadradosC :: [Integer] -> [Integer]
imparesCuadradosC xs = [x*x | x <- xs, odd x]

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir, por recursión, la función
--    imparesCuadradosR :: [Integer] -> [Integer]
-- tal que (imparesCuadradosR xs) es la lista de los cuadrados de los
-- números impares de xs. Por ejemplo,
--    imparesCuadradosR [1,2,3]  ==  [1,9]
-- ---------------------------------------------------------------------

imparesCuadradosR :: [Integer] -> [Integer]
imparesCuadradosR []                 = []
imparesCuadradosR (x:xs) | odd x     = x*x : imparesCuadradosR xs
                         | otherwise = imparesCuadradosR xs

-- ---------------------------------------------------------------------
-- Ejercicio 9. Escribir el cálculo de (imparesCuadradosR [1,2,3]).
-- ---------------------------------------------------------------------

{-
     imparesCuadradosR [1,2,3]
   = imparesCuadradosR (1:(2:(3:[])))
   = 1*1:(imparesCuadradosR (2:(3:[])))
   = 1:(imparesCuadradosR (2:(3:[])))
   = 1:(imparesCuadradosR (3:[]))
   = 1:(3*3:(imparesCuadradosR []))
   = 1:(9:(imparesCuadradosR []))
   = 1:(9:[])
   = [1,9]
-}
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir, por comprensión, la función
--    sumaCuadradosImparesC :: [Integer] -> Integer
-- tal que (sumaCuadradosImparesC xs) es la suma de los cuadrados de los
-- números impares de la lista xs. Por ejemplo,
--    sumaCuadradosImparesC [1,2,3]  ==  10
-- ---------------------------------------------------------------------

sumaCuadradosImparesC :: [Integer] -> Integer
sumaCuadradosImparesC xs = sum [ x*x | x <- xs, odd x ]

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir, por recursión, la función
--    sumaCuadradosImparesR :: [Integer] -> Integer
-- tal que (sumaCuadradosImparesR xs) es la suma de los cuadrados de los
-- números impares de la lista xs. Por ejemplo,
--    sumaCuadradosImparesR [1,2,3]  ==  10
-- ---------------------------------------------------------------------

sumaCuadradosImparesR :: [Integer] -> Integer
sumaCuadradosImparesR []                 = 0
sumaCuadradosImparesR (x:xs)
    | odd x     = x*x + sumaCuadradosImparesR xs
    | otherwise = sumaCuadradosImparesR xs

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir, usando funciones predefinidas, la función
--    entreL :: Integer -> Integer -> [Integer]
-- tal que (entreL m n) es la lista de los números entre m y n. Por
-- ejemplo,
--    entreL 2 5  ==  [2,3,4,5]
-- ---------------------------------------------------------------------

entreL :: Integer -> Integer -> [Integer]
entreL m n = [m..n]

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir, por recursión, la función
--    entreR :: Integer -> Integer -> [Integer]
-- tal que (entreR m n) es la lista de los números entre m y n. Por
```

```haskell
-- ejemplo,
--    entreR 2 5  ==  [2,3,4,5]
-- ------------------------------------------------------------------

entreR :: Integer -> Integer -> [Integer]
entreR m n | m > n     = []
           | otherwise = m : entreR (m+1) n

-- ------------------------------------------------------------------
-- Ejercicio 14. Definir, por comprensión, la función
--    mitadPares :: [Int] -> [Int]
-- tal que (mitadPares xs) es la lista de las mitades de los elementos
-- de xs que son pares. Por ejemplo,
--    mitadPares [0,2,1,7,8,56,17,18]  ==  [0,1,4,28,9]
-- ------------------------------------------------------------------

mitadPares :: [Int] -> [Int]
mitadPares xs = [x `div` 2 | x <- xs, x `mod` 2 == 0]

-- ------------------------------------------------------------------
-- Ejercicio 15. Definir, por recursión, la función
--    mitadParesRec :: [Int] -> [Int]
-- tal que (mitadParesRec []) es la lista de las mitades de los elementos
-- de xs que son pares. Por ejemplo,
--    mitadParesRec [0,2,1,7,8,56,17,18]  ==  [0,1,4,28,9]
-- ------------------------------------------------------------------

mitadParesRec :: [Int] -> [Int]
mitadParesRec [] = []
mitadParesRec (x:xs)
    | even x    = x `div` 2 : mitadParesRec xs
    | otherwise = mitadParesRec xs

-- ------------------------------------------------------------------
-- Ejercicio 16. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
-- ------------------------------------------------------------------

-- La propiedad es
prop_mitadPares :: [Int] -> Bool
```

```haskell
prop_mitadPares xs =
    mitadPares xs == mitadParesRec xs

-- La comprobación es
--    *Main> quickCheck prop_mitadPares
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 17. Definir, por comprensión, la función
--    enRango :: Int -> Int -> [Int] -> [Int]
-- tal que (enRango a b xs) es la lista de los elementos de xs mayores o
-- iguales que a y menores o iguales que b. Por ejemplo,
--    enRango  5 10 [1..15]   ==  [5,6,7,8,9,10]
--    enRango 10  5 [1..15]   ==  []
--    enRango  5  5 [1..15]   ==  [5]
-- ---------------------------------------------------------------------

enRango :: Int -> Int -> [Int] -> [Int]
enRango a b xs = [x | x <- xs, a <= x, x <= b]

-- ---------------------------------------------------------------------
-- Ejercicio 18. Definir, por recursión, la función
--    enRangoRec :: Int -> Int -> [Int] -> [Int]
-- tal que (enRangoRec a b []) es la lista de los elementos de xs
-- mayores o iguales que a y menores o iguales que b. Por ejemplo,
--    enRangoRec  5 10 [1..15]  ==  [5,6,7,8,9,10]
--    enRangoRec 10 5 [1..15]   ==  []
--    enRangoRec  5 5 [1..15]   ==  [5]
-- ---------------------------------------------------------------------

enRangoRec :: Int -> Int -> [Int] -> [Int]
enRangoRec a b [] = []
enRangoRec a b (x:xs)
    | a <= x && x <= b  = x : enRangoRec a b xs
    | otherwise         = enRangoRec a b xs

-- ---------------------------------------------------------------------
-- Ejercicio 19. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
-- ---------------------------------------------------------------------
```

```haskell
-- La propiedad es
prop_enRango :: Int -> Int -> [Int] -> Bool
prop_enRango a b xs =
    enRango a b xs == enRangoRec a b xs

-- La comprobación es
--    *Main> quickCheck prop_enRango
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 20. Definir, por comprensión, la función
--    sumaPositivos :: [Int] -> Int
-- tal que (sumaPositivos xs) es la suma de los números positivos de
-- xs. Por ejemplo,
--    sumaPositivos [0,1,-3,-2,8,-1,6]  ==  15
-- ---------------------------------------------------------------------

sumaPositivos :: [Int] -> Int
sumaPositivos xs = sum [x | x <- xs, x > 0]

-- ---------------------------------------------------------------------
-- Ejercicio 21. Definir, por recursión, la función
--    sumaPositivosRec :: [Int] -> Int
-- tal que (sumaPositivosRec xs) es la suma de los números positivos de
-- xs. Por ejemplo,
--    sumaPositivosRec [0,1,-3,-2,8,-1,6]  ==  15
-- ---------------------------------------------------------------------

sumaPositivosRec :: [Int] -> Int
sumaPositivosRec [] = 0
sumaPositivosRec (x:xs) | x > 0     = x + sumaPositivosRec xs
                        | otherwise = sumaPositivosRec xs

-- ---------------------------------------------------------------------
-- Ejercicio 22. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
```

```haskell
prop_sumaPositivos :: [Int] -> Bool
prop_sumaPositivos xs =
    sumaPositivos xs == sumaPositivosRec xs

-- La comprobación es
--    *Main> quickCheck prop_sumaPositivos
--    +++ OK, passed 100 tests.
```

Relación 11

Definiciones sobre cadenas, orden superior y plegado

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- Esta relación tiene tres partes. La 1ª parte contiene ejercicios con
-- definiciones por comprensión y recursión. En concreto, en la 1ª
-- parte, se estudian funciones para calcular
-- * la compra de una persona agarrada y
-- * la división de una lista numérica según su media.
--
-- La 2ª parte contiene ejercicios sobre cadenas. En concreto, en la 2ª
-- parte, se estudian funciones para calcular
-- * la suma de los dígitos de una cadena,
-- * la capitalización de una cadena,
-- * el título con las reglas de mayúsculas iniciales,
-- * la búsqueda en crucigramas,
-- * las posiciones de un carácter en una cadena y
-- * si una cadena es una subcadena de otra.
--
-- La 3ª parte contiene ejercicios sobre funciones de orden superior. En
-- concreto, en la 3ª parte, se estudian funciones para calcular
-- * el segmento inicial cuyos elementos verifican una propiedad y
-- * el complentario del segmento inicial cuyos elementos verifican una
--   propiedad.
--
```

```
-- La 4ª parte contiene ejercicios sobre definiciones mediante
-- map, filter y plegado. En concreto, en la 4ª parte, se estudian
-- funciones para calcular
-- * la lista de los valores de los elementos que cumplen una propiedad,
-- * la concatenación de una lista de listas,
-- * la redefinición de la función map y
-- * la redefinición de la función filter.
--
-- Estos ejercicios corresponden a los temas 5, 6 y 7 cuyas
-- transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-5.pdf
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-6.pdf
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-7.pdf

-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares                                 --
-- ---------------------------------------------------------------------

import Data.Char
import Data.List
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Definiciones por comprensión y recursión                            --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Una persona es tan agarrada que sólo compra cuando le
-- hacen un descuento del 10% y el precio (con el descuento) es menor o
-- igual que 199.
--
-- Definir, usando comprensión, la función
--    agarrado :: [Float] -> Float
-- tal que (agarrado ps) es el precio que tiene que pagar por una compra
-- cuya lista de precios es ps. Por ejemplo,
--    agarrado  [45.00, 199.00, 220.00, 399.00]  ==  417.59998
-- ---------------------------------------------------------------------

agarrado :: [Float] -> Float
agarrado ps = sum [p * 0.9 | p <- ps, p * 0.9 <= 199]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por recursión, la función
--    agarradoRec :: [Float] -> Float
-- tal que (agarradoRec ps) es el precio que tiene que pagar por una compra
-- cuya lista de precios es ps. Por ejemplo,
--    agarradoRec  [45.00, 199.00, 220.00, 399.00]  ==  417.59998
-- ---------------------------------------------------------------------

agarradoRec :: [Float] -> Float
agarradoRec [] = 0
agarradoRec (p:ps)
    | precioConDescuento <= 199 = precioConDescuento + agarradoRec ps
    | otherwise                 = agarradoRec ps
    where precioConDescuento = p * 0.9

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comprobar con QuickCheck que ambas definiciones son
-- similares; es decir, el valor absoluto de su diferencia es menor que
-- una décima.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_agarrado :: [Float] -> Bool
prop_agarrado xs = abs (agarradoRec xs - agarrado xs) <= 0.1

-- La comprobación es
--    *Main> quickCheck prop_agarrado
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. La función
--    divideMedia :: [Double] -> ([Double],[Double])
-- dada una lista numérica, xs, calcula el par (ys,zs), donde ys
-- contiene los elementos de xs estrictamente menores que la media,
-- mientras que zs contiene los elementos de xs estrictamente mayores
-- que la media. Por ejemplo,
--    divideMedia [6,7,2,8,6,3,4] ==  ([2.0,3.0,4.0],[6.0,7.0,8.0,6.0])
--    divideMedia [1,2,3]         ==  ([1.0],[3.0])
-- Definir la función divideMedia por filtrado, comprensión y
```

```haskell
-- recursión.
-- ---------------------------------------------------------------------

-- La definición por filtrado es
divideMediaF :: [Double] -> ([Double],[Double])
divideMediaF xs = (filter (<m) xs, filter (>m) xs)
    where m = media xs

-- (media xs) es la media de xs. Por ejemplo,
--    media [1,2,3]       ==  2.0
--    media [1,-2,3.5,4]  ==  1.625
-- Nota: En la definición de media se usa la función fromIntegral tal
-- que (fromIntegral x) es el número real correspondiente al número
-- entero x.
media :: [Double] -> Double
media xs = (sum xs) / fromIntegral (length xs)

-- La definición por comprensión es
divideMediaC :: [Double] -> ([Double],[Double])
divideMediaC xs = ([x | x <- xs, x < m], [x | x <- xs, x > m])
    where m = media xs

-- La definición por recursión es
divideMediaR :: [Double] -> ([Double],[Double])
divideMediaR xs = divideMediaR' xs
    where m = media xs
          divideMediaR' [] = ([],[])
          divideMediaR' (x:xs) | x < m  = (x:ys, zs)
                               | x == m = (ys, zs)
                               | x > m  = (ys, x:zs)
                              where (ys, zs) = divideMediaR' xs

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Comprobar con QuickCheck que las tres definiciones
-- anteriores divideMediaF, divideMediaC y divideMediaR son
-- equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_divideMedia :: [Double] -> Bool
```

```haskell
prop_divideMedia xs =
    divideMediaC xs == d &&
    divideMediaR xs == d
    where d = divideMediaF xs

-- La comprobación es
--    *Main> quickCheck prop_divideMedia
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Comprobar con QuickCheck que si (ys,zs) es el par
-- obtenido aplicándole la función divideMediaF a xs, entonces la suma
-- de las longitudes de ys y zs es menor o igual que la longitud de xs.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_longitudDivideMedia :: [Double] -> Bool
prop_longitudDivideMedia xs =
    length ys + length zs <= length xs
    where (ys,zs) = divideMediaF xs

-- La comprobación es
--    *Main> quickCheck prop_longitudDivideMedia
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2.4. Comprobar con QuickCheck que si (ys,zs) es el par
-- obtenido aplicándole la función divideMediaF a xs, entonces todos los
-- elementos de ys son menores que todos los elementos de zs.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_divideMediaMenores :: [Double] -> Bool
prop_divideMediaMenores xs =
    and [y < z | y <- ys, z <- zs]
    where (ys,zs) = divideMediaF xs

-- La comprobación es
--    *Main> quickCheck prop_divideMediaMenores
--    +++ OK, passed 100 tests.
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2.5. Comprobar con QuickCheck que si (ys,zs) es el par
-- obtenido aplicándole la función divideMediaF a xs, entonces la
-- media de xs no pertenece a ys ni a zs.
-- Nota: Usar la función notElem tal que (notElem x ys) se verifica si y
-- no pertenece a ys.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_divideMediaSinMedia :: [Double] -> Bool
prop_divideMediaSinMedia xs =
    notElem m (ys ++ zs)
    where  m       = media xs
           (ys,zs) = divideMediaF xs

-- La comprobación es
--    *Main> quickCheck prop_divideMediaSinMedia
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Funciones sobre cadenas                                           --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir, por comprensión, la función
--    sumaDigitos :: String -> Int
-- tal que (sumaDigitos xs) es la suma de los dígitos de la cadena
-- xs. Por ejemplo,
--    sumaDigitos "SE 2431 X"  ==  10
-- Nota: Usar las funciones isDigit y digitToInt.
-- ---------------------------------------------------------------------

sumaDigitos :: String -> Int
sumaDigitos xs = sum [digitToInt x | x <- xs, isDigit x]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, por recursión, la función
--    sumaDigitosRec :: String -> Int
-- tal que (sumaDigitosRec xs) es la suma de los dígitos de la cadena
```

```haskell
-- xs. Por ejemplo,
--    sumaDigitosRec "SE 2431 X"  ==  10
-- Nota: Usar las funciones isDigit y digitToInt.
-- ---------------------------------------------------------------------

sumaDigitosRec :: String -> Int
sumaDigitosRec [] = 0
sumaDigitosRec (x:xs)
    | isDigit x  = digitToInt x + sumaDigitosRec xs
    | otherwise  = sumaDigitosRec xs

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_sumaDigitos :: String -> Bool
prop_sumaDigitos xs =
    sumaDigitos xs == sumaDigitosRec xs

-- La comprobación es
--    *Main> quickCheck prop_sumaDigitos
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir, por comprensión, la función
--    mayusculaInicial :: String -> String
-- tal que (mayusculaInicial xs) es la palabra xs con la letra inicial
-- en mayúscula y las restantes en minúsculas. Por ejemplo,
--    mayusculaInicial "sEviLLa"  ==  "Sevilla"
-- Nota: Usar las funciones toLower y toUpper.
-- ---------------------------------------------------------------------

mayusculaInicial :: String -> String
mayusculaInicial []     = []
mayusculaInicial (x:xs) = toUpper x : [toLower x | x <- xs]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir, por recursión, la función
```

```haskell
--    mayusculaInicialRec :: String -> String
-- tal que (mayusculaInicialRec xs) es la palabra xs con la letra
-- inicial en mayúscula y las restantes en minúsculas. Por ejemplo,
--    mayusculaInicialRec "sEviLLa"  ==  "Sevilla"
-- ---------------------------------------------------------------------

mayusculaInicialRec :: String -> String
mayusculaInicialRec [] = []
mayusculaInicialRec (x:xs) = toUpper x : aux xs
    where aux (x:xs) = toLower x : aux xs
          aux []     = []

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_mayusculaInicial :: String -> Bool
prop_mayusculaInicial xs =
    mayusculaInicial xs == mayusculaInicialRec xs

-- La comprobación es
--    *Main> quickCheck prop_mayusculaInicial
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Se consideran las siguientes reglas de mayúsculas
-- iniciales para los títulos:
--    * la primera palabra comienza en mayúscula y
--    * todas las palabras que tienen 4 letras como mínimo empiezan
--      con mayúsculas
-- Definir, por comprensión, la función
--    titulo :: [String] -> [String]
-- tal que (titulo ps) es la lista de las palabras de ps con
-- las reglas de mayúsculas iniciales de los títulos. Por ejemplo,
--    *Main> titulo ["eL","arTE","DE","La","proGraMacion"]
--    ["El","Arte","de","la","Programacion"]
-- ---------------------------------------------------------------------
```

```haskell
titulo :: [String] -> [String]
titulo []     = []
titulo (p:ps) = mayusculaInicial p : [transforma p | p <- ps]

-- (transforma p) es la palabra p con mayúscula inicial si su longitud
-- es mayor o igual que 4 y es p en minúscula en caso contrario
transforma :: String -> String
transforma p | length p >= 4 = mayusculaInicial p
             | otherwise     = minuscula p

-- (minuscula xs) es la palabra xs en minúscula.
minuscula :: String -> String
minuscula xs = [toLower x | x <- xs]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir, por recursión, la función
--    tituloRec :: [String] -> [String]
-- tal que (tituloRec ps) es la lista de las palabras de ps con
-- las reglas de mayúsculas iniciales de los títulos. Por ejemplo,
--    *Main> tituloRec ["eL","arTE","DE","La","proGraMacion"]
--    ["El","Arte","de","la","Programacion"]
-- ---------------------------------------------------------------------

tituloRec :: [String] -> [String]
tituloRec []     = []
tituloRec (p:ps) = mayusculaInicial p : tituloRecAux ps
    where tituloRecAux []     = []
          tituloRecAux (p:ps) = transforma p : tituloRecAux ps

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_titulo :: [String] -> Bool
prop_titulo xs = titulo xs == tituloRec xs

-- La comprobación es
--    *Main> quickCheck prop_titulo
```

```
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Definir, por comprensión, la función
--    buscaCrucigrama :: Char -> Int -> Int -> [String] -> [String]
-- tal que (buscaCrucigrama l pos lon ps) es la lista de las palabras de
-- la lista de palabras ps que tienen longitud lon y poseen la letra l en
-- la posición pos (comenzando en 0). Por ejemplo,
--    *Main> buscaCrucigrama 'c' 1 7 ["ocaso", "casa", "ocupado"]
--    ["ocupado"]
-- ---------------------------------------------------------------------

buscaCrucigrama :: Char -> Int -> Int -> [String] -> [String]
buscaCrucigrama l pos lon ps =
    [p | p <- ps,  length p == lon,
         0 <= pos,  pos < length p,
         p !! pos == l]

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Definir, por recursión, la función
--    buscaCrucigramaRec :: Char -> Int -> Int -> [String] -> [String]
-- tal que (buscaCrucigramaRec l pos lon ps) es la lista de las palabras
-- de la lista de palabras ps que tienn longitud lon y posen la letra l
-- en la posición pos (comenzando en 0). Por ejemplo,
--    *Main> buscaCrucigramaRec 'c' 1 7 ["ocaso", "acabado", "ocupado"]
--    ["acabado","ocupado"]
-- ---------------------------------------------------------------------

buscaCrucigramaRec :: Char -> Int -> Int -> [String] -> [String]
buscaCrucigramaRec letra pos lon [] = []
buscaCrucigramaRec letra pos lon (p:ps)
    | length p == lon && 0 <= pos && pos < length p && p !! pos == letra
      = p : buscaCrucigramaRec letra pos lon ps
    | otherwise
      = buscaCrucigramaRec letra pos lon ps

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
-- ---------------------------------------------------------------------
```

```haskell
-- La propiedad es
prop_buscaCrucigrama :: Char -> Int -> Int -> [String] -> Bool
prop_buscaCrucigrama letra pos lon ps =
    buscaCrucigrama letra pos lon ps == buscaCrucigramaRec letra pos lon ps

-- La comprobación es
--    *Main> quickCheck prop_buscaCrucigrama
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 6.1. Definir, por comprensión, la función
--    posiciones :: String -> Char -> [Int]
-- tal que (posiciones xs y) es la lista de la posiciones del carácter y
-- en la cadena xs. Por ejemplo,
--    posiciones "Salamamca" 'a'  ==  [1,3,5,8]
-- ---------------------------------------------------------------------

posiciones :: String -> Char -> [Int]
posiciones xs y = [n | (x,n) <- zip xs [0..], x == y]

-- ---------------------------------------------------------------------
-- Ejercicio 6.2. Definir, por recursión, la función
--    posicionesRec :: String -> Char -> [Int]
-- tal que (posicionesRec xs y) es la lista de la posiciones del
-- carácter y en la cadena xs. Por ejemplo,
--    posicionesRec "Salamamca" 'a'  ==  [1,3,5,8]
-- ---------------------------------------------------------------------

posicionesRec :: String -> Char -> [Int]
posicionesRec xs y = posicionesAux xs y 0
    where
      posicionesAux [] y n = []
      posicionesAux (x:xs) y n | x == y    = n : posicionesAux xs y (n+1)
                               | otherwise = posicionesAux xs y (n+1)

-- ---------------------------------------------------------------------
-- Ejercicio 6.3. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
-- ---------------------------------------------------------------------
```

```haskell
-- La propiedad es
prop_posiciones :: String -> Char -> Bool
prop_posiciones xs y =
    posiciones xs y == posicionesRec xs y

-- La comprobación es
--    *Main> quickCheck prop_posiciones
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 7.1. Definir, por recursión, la función
--    contieneRec :: String -> String -> Bool
-- tal que (contieneRec xs ys) se verifica si ys es una subcadena de
-- xs. Por ejemplo,
--    contieneRec "escasamente" "casa"   ==  True
--    contieneRec "escasamente" "cante"  ==  False
--    contieneRec "" ""                  ==  True
-- Nota: Se puede usar la predefinida (isPrefixOf ys xs) que se verifica
-- si ys es un prefijo de xs.
-- ---------------------------------------------------------------------

contieneRec :: String -> String -> Bool
contieneRec _ []  = True
contieneRec [] ys = False
contieneRec xs ys = isPrefixOf ys xs || contieneRec (tail xs) ys

-- ---------------------------------------------------------------------
-- Ejercicio 7.2. Definir, por comprensión, la función
--    contiene :: String -> String -> Bool
-- tal que (contiene xs ys) se verifica si ys es una subcadena de
-- xs. Por ejemplo,
--    contiene "escasamente" "casa"      ==  True
--    contiene "escasamente" "cante"     ==  False
--    contiene "casado y casada" "casa"  ==  True
--    contiene "" ""                     ==  True
-- Nota: Se puede usar la predefinida (isPrefixOf ys xs) que se verifica
-- si ys es un prefijo de xs.
-- ---------------------------------------------------------------------
```

```haskell
contiene :: String -> String -> Bool
contiene xs ys = sufijosComenzandoCon xs ys /= []

-- (sufijosComenzandoCon xs ys) es la lista de los sufijos de xs que
-- comienzan con ys. Por ejemplo,
--    sufijosComenzandoCon "abacbad" "ba"  ==  ["bacbad","bad"]
sufijosComenzandoCon :: String -> String -> [String]
sufijosComenzandoCon xs ys = [x | x <- sufijos xs, isPrefixOf ys x]

-- (sufijos xs) es la lista de sufijos de xs. Por ejemplo,
--    sufijos "abc"  ==  ["abc","bc","c",""]
sufijos :: String -> [String]
sufijos xs = [drop i xs | i <- [0..length xs]]

-- ---------------------------------------------------------------------
-- Ejercicio 7.3. Comprobar con QuickCheck que ambas definiciones son
-- equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_contiene :: String -> String -> Bool
prop_contiene xs ys =
    contieneRec xs ys == contiene xs ys

-- La comprobación es
--    *Main> quickCheck prop_contiene
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Funciones de orden superior                                        --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 8. Redefinir por recursión la función
--    takeWhile :: (a -> Bool) -> [a] -> [a]
-- tal que (takeWhile p xs) es la lista de los elemento de xs hasta el
-- primero que no cumple la propiedad p. Por ejemplo,
--    takeWhile (<7) [2,3,9,4,5]  ==  [2,3]
-- ---------------------------------------------------------------------
```

```haskell
takeWhile' :: (a -> Bool) -> [a] -> [a]
takeWhile' _ [] = []
takeWhile' p (x:xs)
    | p x       = x : takeWhile' p xs
    | otherwise = []

-- ---------------------------------------------------------------------
-- Ejercicio 9. Redefinir por recursión la función
--    dropWhile :: (a -> Bool) -> [a] -> [a]
-- tal que (dropWhile p xs) es la lista de eliminando los elemento de xs
-- hasta el primero que cumple la propiedad p. Por ejemplo,
--    dropWhile (<7) [2,3,9,4,5]  =>  [9,4,5]
-- ---------------------------------------------------------------------

dropWhile' :: (a -> Bool) -> [a] -> [a]
dropWhile' _ [] = []
dropWhile' p (x:xs)
    | p x       = dropWhile' p xs
    | otherwise = x:xs

-- ---------------------------------------------------------------------
-- 4. Definiciones mediante map, filter y plegado                      --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 10. Se considera la función
--    filtraAplica :: (a -> b) -> (a -> Bool) -> [a] -> [b]
-- tal que (filtraAplica f p xs) es la lista obtenida aplicándole a los
-- elementos de xs que cumplen el predicado p la función f. Por ejemplo,
--    filtraAplica (4+) (<3) [1..7]  =>  [5,6]
-- Se pide, definir la función
-- 1. por comprensión,
-- 2. usando map y filter,
-- 3. por recursión y
-- 4. por plegado (con foldr).
-- ---------------------------------------------------------------------

-- La definición con lista de comprensión es
filtraAplica_1 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplica_1 f p xs = [f x | x <- xs, p x]
```

```haskell
-- La definición con map y filter es
filtraAplica_2 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplica_2 f p xs = map f (filter p xs)

-- La definición por recursión es
filtraAplica_3 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplica_3 f p [] = []
filtraAplica_3 f p (x:xs) | p x       = f x : filtraAplica_3 f p xs
                          | otherwise = filtraAplica_3 f p xs

-- La definición por plegado es
filtraAplica_4 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplica_4 f p = foldr g []
                     where g x y | p x       = f x : y
                                 | otherwise = y

-- La definición por plegado usando lambda es
filtraAplica_4' :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplica_4' f p =
    foldr (\x y -> if p x then (f x : y) else y) []

-- ---------------------------------------------------------------------
-- Ejercicio 11. Redefinir, usando foldr, la función concat. Por ejemplo,
--    concat' [[1,3],[2,4,6],[1,9]]  ==  [1,3,2,4,6,1,9]
-- ---------------------------------------------------------------------

-- La definición por recursión es
concatR :: [[a]] -> [a]
concatR [] = []
concatR (xs:xss) = xs ++ concatR xss

-- La definición por plegado es
concat' :: [[a]] -> [a]
concat' = foldr (++) []

-- ---------------------------------------------------------------------
-- Ejercicio 14. Redefinir, usando foldr, la función map. Por ejemplo,
--    map' (+2) [1,7,3]  ==  [3,9,5]
-- ---------------------------------------------------------------------
```

```haskell
-- La definición por recursión es
mapR :: (a -> b) -> [a] -> [b]
mapR f [] = []
mapR f (x:xs) = f x : mapR f xs

-- La definición por plegado es
map' :: (a -> b) -> [a] -> [b]
map' f = foldr g []
         where g x xs = f x : xs

-- La definición por plegado usando lambda es
map'' :: (a -> b) -> [a] -> [b]
map'' f = foldr (\x y -> f x:y) []

-- Otra definición es
map''' :: (a -> b) -> [a] -> [b]
map''' f = foldr ((:) . f) []

-- ---------------------------------------------------------------------
-- Ejercicio 15. Redefinir, usando foldr, la función filter. Por
-- ejemplo,
--    filter' (<4) [1,7,3,2]  =>  [1,3,2]
-- ---------------------------------------------------------------------

-- La definición por recursión es
filterR :: (a -> Bool) -> [a] -> [a]
filterR p [] = []
filterR p (x:xs) | p x       = x : filterR p xs
                 | otherwise = filterR p xs

-- La definición por plegado es
filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr g []
            where g x y | p x       = x:y
                        | otherwise = y

-- La definición por plegado y lambda es
filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x y -> if (p x) then (x:y) else y) []
```

Relación 12

Definiciones por plegado

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- Esta relación tiene tres partes. La 1ª parte contiene ejercicios con
-- definiciones por plegado. En concreto, se estudian definiciones por
-- plegado para calcular
-- * el máximo elemento de una lista,
-- * el mínimo elemento de una lista,
-- * la inversa de una lista,
-- * el número correspondiente a la lista de sus cifras,
-- * la suma de las sumas de las listas de una lista de listas,
-- * la lista obtenida borrando las ocurrencias de un elemento y
-- * la diferencia de dos listas.
--
-- En la 2ª parte se propone la modificación del programa de transmición
-- de mensajes, estudiado en el tema 7, para detectar errores.
--
-- En la 3ª parte se propone la implementación de la identidad de
-- Bezout.
--
-- Los ejercicios de esta relación corresponden al tema 7 cuyas
-- transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-7.pdf

-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares                                 --
```

```haskell
-- ---------------------------------------------------------------------

import Data.Char
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- 1. Definiciones por plegado                                         --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Redefinir, mediante plegado, la función maximum.
-- ---------------------------------------------------------------------

-- La definición por recursión es
maximumR :: Ord a => [a] -> a
maximumR [x]      = x
maximumR (x:y:ys) = max y (maximumR (x:ys))

-- La definición por plegado con foldr es
maximum' :: Ord a => [a] -> a
maximum' (x:xs) = (foldr max x) xs

-- La definición de foldr1 es
--    foldr1 :: (a -> a -> a) -> [a] -> a
--    foldr1 _ [x]    =  x
--    foldr1 f (x:xs) =  f x (foldr1 f xs)

-- Otra definición por recursión es
maximumR' :: Ord a => [a] -> a
maximumR' [x]      = x
maximumR' (x:y:ys) = max x (maximumR' (y:ys))

-- Otra definición, usando foldr1, es
maximum'' :: Ord a => [a] -> a
maximum'' = foldr1 max

-- ---------------------------------------------------------------------
-- Ejercicio 2. Redefinir por plegado la función minimum.
-- ---------------------------------------------------------------------
```

```haskell
-- Por analogía con el anterior.

-- Definición con foldr:
minimum' :: Ord a => [a] -> a
minimum' (x:xs) = (foldr min x) xs

-- Otra definición, usando foldr1, es
minimum'' :: Ord a => [a] -> a
minimum'' = foldr1 min

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir, usando foldr, la función
--    inversaFR :: [a] -> [a]
-- tal que (inversaFR xs) es la inversa de la lista xs. Por ejemplo,
--    inversaFR [3,5,2,4,7]  =>  [7,4,2,5,3]
-- ---------------------------------------------------------------------

-- La definición por recursión es
inversaR :: [a] -> [a]
inversaR [] = []
inversaR (x:xs) = (inversaR xs) ++ [x]

-- La definición con foldR es
inversaFR :: [a] -> [a]
inversaFR = foldr f []
    where f x y = y ++ [x]

-- La definición anterior puede simplificarse a
inversaFR' :: [a] -> [a]
inversaFR' = foldr f []
    where f x = (++ [x])

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir, usando foldl, la función
--    inversaFL :: [a] -> [a]
-- tal que (inversaFL xs) es la inversa de la lista xs. Por ejemplo,
--    inversaFL [3,5,2,4,7]  ==  [7,4,2,5,3]
-- ---------------------------------------------------------------------

-- La definición por recursión con acumulador es
```

```haskell
inversaR' :: [a] -> [a]
inversaR' xs = inversaAux [] xs
    where inversaAux a []     = a
          inversaAux a (x:xs) = inversaAux (x:a) xs

-- La definición de foldl es
--    foldl :: (a -> b -> a) -> a -> [b] -> a
--    foldl f z0 xs0 = aux z0 xs0
--        where aux z []     = z
--              aux z (x:xs) = aux (f z x) xs

-- La definción de inversaFL es
inversaFL :: [a] -> [a]
inversaFL = foldl (\a x -> x:a) []

-- La definción de inversaFL puede simplificarse usando flip:
inversaFL' :: [a] -> [a]
inversaFL' = foldl (flip(:)) []

-- ---------------------------------------------------------------------
-- Ejercicio 5. Comprobar con QuickCheck que las funciones reverse,
-- inversaFR e inversaFL son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_inversa :: Eq a => [a] -> Bool
prop_inversa xs =
    inversaFR xs == ys &&
    inversaFL xs == ys
    where ys = reverse xs

-- La comprobación es
--    *Main> quickCheck prop_inversa
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 6. Comparar la eficiencia de inversaFR e inversaFL
-- calculando el tiempo y el espacio que usado en evaluar las siguientes
-- expresiones:
--    head (inversaFR [1..100000])
```

```
--    head (inversaFL [1..100000])
-- ---------------------------------------------------------------------

-- La sesión es
--    *Main> :set +s
--    *Main> head (inversaFR [1..100000])
--    100000
--    (0.41 secs, 20882460 bytes)
--    *Main> head (inversaFL [1..100000])
--    1
--    (0.00 secs, 525148 bytes)
--    *Main> :unset +s

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir, la función
--    dec2ent :: [Int] -> Int
-- tal que (dec2ent xs) es el entero correspondiente a la expresión
-- decimal xs. Por ejemplo,
--    dec2ent [2,3,4,5]  ==  2345
-- Escribir dos definiciones:
-- * dec2entR por recursión
-- * dec2entF usando foldl
-- ---------------------------------------------------------------------

-- La definición por recursión es
dec2entR :: [Int] -> Int
dec2entR xs = dec2entR' 0 xs
    where dec2entR' a []     = a
          dec2entR' a (x:xs) = dec2entR' (10*a+x) xs

-- La definición usando foldl es
dec2entF :: [Int] -> Int
dec2entF = foldl f 0
    where f a x = 10*a+x

-- La definición usando foldl y lambda es
dec2entF' :: [Int] -> Int
dec2entF' = foldl (\a x -> 10*a+x) 0

-- ---------------------------------------------------------------------
```

```haskell
-- Ejercicio 8. Definir, mediante plegado, la función
--    sumll :: Num a => [[a]] -> a
-- tal que (sumll xss) es la suma de las sumas de las listas de xss. Por
-- ejemplo,
--    sumll [[1,3],[2,5]]  ==  11
-- ---------------------------------------------------------------------

-- La definición por recursión es
sumllR :: Num a => [[a]] -> a
sumllR [] = 0
sumllR (x:xs) = sum x + sumllR xs

-- La definición por plegado es
sumll :: Num a => [[a]] -> a
sumll = foldr (\x y -> sum x + y) 0

-- La anterior definición puede simplificarse, teniendo en cuenta que
--   sum = foldr (+) 0
-- como sigue
--    sumll = foldr (\x y -> sum x + y) 0
--          = foldr (\x y -> ((foldr (+) 0) x) + y) 0
--          = foldl (\x y -> ((foldl (+) y) x)) 0
--          = foldl (foldl (+)) 0
sumll' :: Num a => [[a]] -> a
sumll' = foldl (foldl (+)) 0

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir, mediante plegado, la función
--    borra :: Eq a => a -> a -> [a]
-- tal que (borra y xs) es la lista obtenida borrando las ocurrencias de
-- y en xs. Por ejemplo,
--    borra 5 [2,3,5,6]    ==  [2,3,6]
--    borra 5 [2,3,5,6,5]  ==  [2,3,6]
--    borra 7 [2,3,5,6,5]  ==  [2,3,5,6,5]
-- ---------------------------------------------------------------------

-- La definición de borra por recursión es
borraR :: Eq a => a -> [a] -> [a]
borraR z [] = []
borraR z (x:xs) | z == x    = borraR z xs
```

```
                | otherwise = x : borraR z xs

-- La definición por plegado es
borra :: Eq a => a -> [a] -> [a]
borra z = foldr f []
    where f x y | z == x    = y
                | otherwise = x:y

-- La definición por plegado con lambda es es
borra' :: Eq a => a -> [a] -> [a]
borra' z = foldr (\x y -> if z==x then y else x:y) []

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir, mediante plegado, la función
--    diferencia :: Eq a => [a] -> [a] -> [a]
-- tal que (diferencia xs ys) es la diferencia del conjunto xs e ys; es
-- decir el conjunto de los elementos de xs que no pertenecen a ys. Por
-- ejemplo,
--    diferencia [2,3,5,6] [5,2,7]  ==  [3,6]
-- ---------------------------------------------------------------------

-- La definición por recursión es
diferenciaR :: Eq a => [a] -> [a] -> [a]
diferenciaR xs ys = aux xs xs ys
    where aux a xs []     = a
          aux a xs (y:ys) = aux (borra y a) xs ys

-- La definición, para aproximarse al patrón foldr, se puede escribir como
diferenciaR' :: Eq a => [a] -> [a] -> [a]
diferenciaR' xs ys = aux xs xs ys
    where aux a xs []     = a
          aux a xs (y:ys) = aux (flip borra a y) xs ys

-- La definición por plegado es
diferencia :: Eq a => [a] -> [a] -> [a]
diferencia xs ys = foldl (flip borra) xs ys

-- La definición anterior puede simplificarse a
diferencia' :: Eq a => [a] -> [a] -> [a]
diferencia' = foldl (flip borra)
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 11. En el tema se ha definido la función
--    composicionLista :: [a -> a] -> (a -> a)
-- tal que (composicionLista fs) es la composición de la lista de
-- funciones fs. Por ejemplo,
--    composicionLista [(*2),(^2)] 3        ==  18
--    composicionLista [(^2),(*2)] 3        ==  36
--    composicionLista [(/9),(^2),(*2)] 3   ==  4.0
-- La definición es
--    composicionLista = foldr (.) id
--
-- Explicar por qué la siguiente definición no es válida:
--    sumaCuadradosPares =
--        composicionLista [sum, map (^2), filter even]
-- ---------------------------------------------------------------------

-- El argumento de composicionLista tiene que ser una lista de funciones
-- del mismo tipo y las funciones de la lista
--    [sum, map (^2), filter even]
-- no tienen el mismo tipo. En efecto,
--    sum         :: [Int] -> Int
--    map (^2)    :: [Int] -> [Int]
--    filter even :: [Int] -> [Int]

-- ---------------------------------------------------------------------
-- Ejercicio 12. Se define el siguiente patrón
--    unfold :: (a -> Bool) -> (a -> b) -> (a -> a) -> a -> [b]
--    unfold p h t x | p x       = []
--                   | otherwise = h x : unfold p h t (t x)
-- Con el patrón unfold pueden simplificarse algunas definiciones. Por
-- ejemplo, en el tema se ha definido la función
--    int2bin :: Int -> [Int]
-- tal que (int2bin x) es el número binario correspondiente al número
-- decimal x. Por ejemplo,
--    int2bin 13  =>  [1,0,1,1]
-- La definición en el tema es
--    int2bin 0 = []
--    int2bin n = n `mod` 2 : int2bin (n `div` 2)
-- Usando unfold, int2bin puede definirse como
```

```haskell
--    int2bin = unfold (== 0) (`mod` 2) (`div` 2)
-- ---------------------------------------------------------------------

unfold :: (a -> Bool) -> (a -> b) -> (a -> a) -> a -> [b]
unfold p h t x | p x       = []
               | otherwise = h x : unfold p h t (t x)

-- ---------------------------------------------------------------------
-- Ejercicio 13. Redefinir, usando unfold, la función definida en el
-- tema
--    separaOctetos :: [Int] -> [[Int]]
-- tal que (separaOctetos bs) es la lista obtenida separando la lista de
-- bits bs en listas de 8 elementos. Por ejemplo,
--    *Main> separaOctetos [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0]
--    [[1,0,0,0,0,1,1,0],[0,1,0,0,0,1,1,0]]
-- Comprobar con quickCheck la equivalencia de las definiciones.
-- ---------------------------------------------------------------------

-- La definición en el tema es
separaOctetos :: [Int] -> [[Int]]
separaOctetos [] = []
separaOctetos bs = take 8 bs : separaOctetos (drop 8 bs)

-- La definición con unfold es
separaOctetos' :: [Int] -> [[Int]]
separaOctetos' = unfold null (take 8) (drop 8)

-- La propiedad es
prop_separaOctetos xs =
    separaOctetos' xs == separaOctetos xs

-- La comprobación es
--    *Main> quickCheck prop_separaOctetos
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 14. Redefinir, usando unfold, la función map.
-- ---------------------------------------------------------------------

-- La definición es
```

```haskell
map'' :: (a -> b) -> [a] -> [b]

-- ---------------------------------------------------------------------
-- 2. Transmisión de mensajes                                          --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 15. En este ejercicio se va a modificar el programa de
-- transmisión de cadenas para detectar errores de transmisión sencillos
-- usando bits de paridad. Es decir, cada octeto de ceros y unos
-- generado durante la codificación se extiende con un bit de paridad
-- que será un uno si el número contiene un número impar de unos y cero
-- en caso contrario. En la decodificación, en cada número binario de 9
-- cifras debe comprobarse que la paridad es correcta, en cuyo caso se
-- descarta el bit de paridad. En caso contrario, debe generarse un
-- mensaje de error en la paridad.
--
-- Se usarán las siguientes definiciones del tema
-- ---------------------------------------------------------------------

type Bit = Int

bin2int :: [Bit] -> Int
bin2int =  foldr (\x y -> x + 2*y) 0

int2bin :: Int -> [Bit]
int2bin 0 =  []
int2bin n =  n `mod` 2 : int2bin (n `div` 2)

creaOcteto :: [Bit] -> [Bit]
creaOcteto bs =  take 8 (bs ++ repeat 0)

-- La definición anterior puede simplificarse a
creaOcteto' :: [Bit] -> [Bit]
creaOcteto' =  take 8 . (++ repeat 0)

-- ---------------------------------------------------------------------
-- Ejercicio 16. Definir la función
--    paridad :: [Bit] -> Bit
-- tal que (paridad bs) es el bit de paridad de bs; es decir, 1 si bs
```

```haskell
-- contiene un número impar de unos y 0 en caso contrario. Por ejemplo,
--    paridad [0,1,1]      =>  0
--    paridad [0,1,1,0,1]  =>  1
-- ---------------------------------------------------------------------

paridad :: [Bit] -> Bit
paridad bs | odd (sum bs)  = 1
           | otherwise     = 0

-- ---------------------------------------------------------------------
-- Ejercicio 17. Definir la función
--    agregaParidad :: [Bit] -> [Bit]
-- tal que (agregaParidad bs) es la lista obtenida añadiendo al
-- principio de bs su paridad. Por ejemplo,
--    agregaParidad [0,1,1]      =>  [0,0,1,1]
--    agregaParidad [0,1,1,0,1]  =>  [1,0,1,1,0,1]
-- ---------------------------------------------------------------------

agregaParidad :: [Bit] -> [Bit]
agregaParidad bs = (paridad bs) : bs

-- ---------------------------------------------------------------------
-- Ejercicio 18. Definir la función
--    codifica :: String -> [Bit]
-- tal que (codifica c) es la codificación de la cadena c como una lista
-- de bits obtenida convirtiendo cada carácter en un número Unicode,
-- convirtiendo cada uno de dichos números en un octeto con su paridad y
-- concatenando los octetos con paridad para obtener una lista de
-- bits. Por ejemplo,
--    *Main> codifica "abc"
--    [1,1,0,0,0,0,1,1,0,1,0,1,0,0,0,1,1,0,0,1,1,0,0,0,1,1,0]
-- ---------------------------------------------------------------------

codifica :: String -> [Bit]
codifica = concat . map (agregaParidad . creaOcteto . int2bin . ord)

-- ---------------------------------------------------------------------
-- Ejercicio 19. Definir la función
--    separa9 :: [Bit] -> [[Bit]]
-- tal que (separa9 bs)} es la lista obtenida separando la lista de bits
```

```haskell
-- bs en listas de 9 elementos. Por ejemplo,
--    *Main> separa9 [1,1,0,0,0,0,1,1,0,1,0,1,0,0,0,1,1,0,0,1,1,0,0,0,1,1,0]
--    [[1,1,0,0,0,0,1,1,0],[1,0,1,0,0,0,1,1,0],[0,1,1,0,0,0,1,1,0]]
-- ---------------------------------------------------------------------

separa9 :: [Bit] -> [[Bit]]
separa9 []   = []
separa9 bits = take 9 bits : separa9 (drop 9 bits)

-- ---------------------------------------------------------------------
-- Ejercicio 20. Definir la función
--    compruebaParidad :: [Bit] -> [Bit ]
-- tal que (compruebaParidad bs) es el resto de bs si el primer elemento
-- de bs es el bit de paridad del resto de bs y devuelve error de
-- paridad en caso contrario. Por ejemplo,
--    *Main> compruebaParidad [1,1,0,0,0,0,1,1,0]
--    [1,0,0,0,0,1,1,0]
--    *Main> compruebaParidad [0,1,0,0,0,0,1,1,0]
--    *** Exception: paridad erronea
-- Usar la función del preludio
--    error :: String -> a
-- tal que (error c) devuelve la cadena c.
-- ---------------------------------------------------------------------

compruebaParidad :: [Bit] -> [Bit ]
compruebaParidad (b:bs)
    | b == paridad bs = bs
    | otherwise       = error "paridad erronea"

-- ---------------------------------------------------------------------
-- Ejercicio 21. Definir la función
--    descodifica :: [Bit] -> String
-- tal que (descodifica bs) es la cadena correspondiente a la lista de
-- bits con paridad. Para ello, en cada número binario de 9 cifras debe
-- comprobarse que la paridad es correcta, en cuyo caso se descarta el
-- bit de paridad. En caso contrario, debe generarse un mensaje de error
-- en la paridad. Por ejemplo,
--    descodifica [1,1,0,0,0,0,1,1,0,1,0,1,0,0,0,1,1,0,0,1,1,0,0,0,1,1,0]
--    =>  "abc"
--    descodifica [1,0,0,0,0,0,1,1,0,1,0,1,0,0,0,1,1,0,0,1,1,0,0,0,1,1,0]
```

```
--    =>  "*** Exception: paridad erronea
-- ---------------------------------------------------------------------

descodifica :: [Bit] -> String
descodifica = map (chr . bin2int . compruebaParidad) . separa9

-- ---------------------------------------------------------------------
-- Ejercicio 22. Se define la función
transmite :: ([Bit] -> [Bit]) -> String -> String
transmite canal =  descodifica . canal . codifica
-- tal que (transmite c t) es la cadena obtenida transmitiendo la cadena
-- t a través del canal c. Calcular el reultado de trasmitir la cadena
-- "Conocete a ti mismo" por el canal identidad (id) y del canal que
-- olvida el primer bit (tail).
-- ---------------------------------------------------------------------

--    *Main> transmite id "Conocete a ti mismo"
--    "Conocete a ti mismo"
--    *Main> transmite tail "Conocete a ti mismo"
--    "*** Exception: paridad erronea

-- ---------------------------------------------------------------------
-- 3. La identidad de Bezout                                           --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 23. [La identidad de Bezout] Definir la función
--    bezout :: Integer -> Integer -> (Integer, Integer)
-- tal que (bezout a b) es un par de números x e y tal que a*x+b*y es el
-- máximo común divisor de a y b. Por ejemplo,
--    bezout 12 30  ==  (-2,1)
-- Indicación: Se puede usar la función quotRem tal que (quotRem x y) es
-- el par formado por el cociente y el resto de dividir x entre y.
-- ---------------------------------------------------------------------

-- Ejemplo de cálculo
--    a  b   q r
--    12 30  0 12  (-1,1-0*(-1)) = (-1,1)
--    30 12  1 18  (1,0-1*1)     = (1,-1)
--    12 18  0 18  (0,1-0*0)     = (0,1)
```

```
--    18 18  1  0  (1,0)

bezout :: Integer -> Integer -> (Integer, Integer)
bezout _ 0 = (1,0)
bezout _ 1 = (0,1)
bezout a b = (y, x - q*y)
    where (x,y) = bezout b r
          (q,r) = quotRem a b

-- ---------------------------------------------------------------------
-- Ejercicio 24. Comprobar con QuickCheck que si a>0, b>0 y
-- (x,y) es el valor de (bezout a b), entonces a*x+b*y es igual al
-- máximo común divisor de a y b.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_Bezout :: Integer -> Integer -> Property
prop_Bezout a b = a>0 && b>0 ==> a*x+b*y == gcd a b
    where (x,y) = bezout a b

-- La comprobación es
--   Main> quickCheck prop_Bezout
--   OK, passed 100 tests.
```

Relación 13

Resolución de problemas matemáticos

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En esta relación se plantea la resolución de distintos problemas
-- matemáticos. En concreto,
-- * el problema de Ullman sobre la existencia de subconjunto del tamaño
--   dado y con su suma acotada,
-- * las descomposiciones de un número como suma de dos cuadrados,
-- * el problema 145 del proyecto Euler,
-- * el grafo de una función sobre los elementos que cumplen una
--   propiedad,
-- * los números semiperfectos,
-- * el producto, por plegado, de los números que verifican una propiedad,
-- * el carácter funcional de una relación y
-- * las cabezas y las colas de una lista.
--
-- Además, de los 2 primeros se presentan distintas definiciones y se
-- compara su eficiencia.
--
-- Estos ejercicios corresponden a los temas 5, 6 y 7 cuyas
-- transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-5.pdf
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-6.pdf
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-7.pdf
```

```haskell
-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares                                 --
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    ullman :: (Num a, Ord a) => a -> Int -> [a] -> Bool
-- tal que (ullman t k xs) se verifica si xs tiene un subconjunto con k
-- elementos cuya suma sea menor que t. Por ejemplo,
--    ullman 9 3 [1..10] == True
--    ullman 5 3 [1..10] == False
-- ---------------------------------------------------------------------

-- 1ª solución (corta y eficiente)
ullman :: (Ord a, Num a) => a -> Int -> [a] -> Bool
ullman t k xs = sum (take k (sort xs)) < t

-- 2ª solución (larga e ineficiente)
ullman2 :: (Num a, Ord a) => a -> Int -> [a] -> Bool
ullman2 t k xs =
    [ys | ys <- subconjuntos xs, length ys == k, sum ys < t] /= []

-- (subconjuntos xs) es la lista de los subconjuntos de xs. Por
-- ejemplo,
--    subconjuntos "bc"  ==  ["","c","b","bc"]
--    subconjuntos "abc" ==  ["","c","b","bc","a","ac","ab","abc"]
subconjuntos :: [a] -> [[a]]
subconjuntos [] = [[]]
subconjuntos (x:xs) = zss++[x:ys | ys <- zss]
    where zss = subconjuntos xs

-- Los siguientes ejemplos muestran la diferencia en la eficencia:
--    *Main> ullman 9 3 [1..20]
--    True
--    (0.02 secs, 528380 bytes)
--    *Main> ullman2 9 3 [1..20]
```

```
--    True
--    (4.08 secs, 135267904 bytes)
--    *Main> ullman 9 3 [1..100]
--    True
--    (0.02 secs, 526360 bytes)
--    *Main> ullman2 9 3 [1..100]
--      C-c C-cInterrupted.
--    Agotado

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    sumasDe2Cuadrados :: Integer -> [(Integer, Integer)]
-- tal que (sumasDe2Cuadrados n) es la lista de los pares de números
-- tales que la suma de sus cuadrados es n y el primer elemento del par
-- es mayor o igual que el segundo. Por ejemplo,
--    sumasDe2Cuadrados 25  ==  [(5,0),(4,3)]
-- ---------------------------------------------------------------------

-- Primera definición:
sumasDe2Cuadrados_1 :: Integer -> [(Integer, Integer)]
sumasDe2Cuadrados_1 n =
    [(x,y) | x <- [n,n-1..0],
             y <- [0..x],
             x*x+y*y == n]

-- Segunda definición:
sumasDe2Cuadrados_2 :: Integer -> [(Integer, Integer)]
sumasDe2Cuadrados_2 n =
    [(x,y) | x <- [a,a-1..0],
             y <- [0..x],
             x*x+y*y == n]
    where a = ceiling (sqrt (fromIntegral n))

-- Tercera definición:
sumasDe2Cuadrados_3 :: Integer -> [(Integer, Integer)]
sumasDe2Cuadrados_3 n = aux (ceiling (sqrt (fromIntegral n))) 0 where
    aux x y | x < y          = []
            | x*x + y*y <  n = aux x (y+1)
            | x*x + y*y == n = (x,y) : aux (x-1) (y+1)
            | otherwise      = aux (x-1) y
```

```
-- Comparación
--    +----------+---------------+---------------+---------------+
--    | n        | 1ª definición | 2ª definición | 3ª definición |
--    +----------+---------------+---------------+---------------+
--    |      999 | 2.17 segs     |   0.02 segs   | 0.01 segs     |
--    | 48612265 |               | 140.38 segs   | 0.13 segs     |
--    +----------+---------------+---------------+---------------+

-- ---------------------------------------------------------------------
-- Ejercicio 3. (Basado en el problema 145 del Proyecto Euler). Se dice
-- que un número n es reversible si su última cifra es distinta de 0 y
-- la suma de n y el número obtenido escribiendo las cifras de n en
-- orden inverso es un número que tiene todas sus cifras impares. Por
-- ejemplo, 36 es reversible porque 36+63=99 tiene todas sus cifras
-- impares, 409 es reversible porque 409+904=1313 tiene todas sus cifras
-- impares,  243 no es reversible porque 243+342=585 no tiene todas sus
-- cifras impares.
-- Definir la función
--    reversiblesMenores :: Int -> Int
-- tal que (reversiblesMenores n) es la cantidad de números reversibles
-- menores que n. Por ejemplo,
--    reversiblesMenores 10   == 0
--    reversiblesMenores 100  == 20
--    reversiblesMenores 1000 == 120
-- ---------------------------------------------------------------------

reversiblesMenores :: Int -> Int
reversiblesMenores n = length [x | x <- [1..n-1], esReversible x]

-- (esReversible n) se verifica si n es reversible; es decir, si su
-- última cifra es distinta de 0 y la suma de n y el número obtenido
-- escribiendo las cifras de n en orden inverso es un número que tiene
-- todas sus cifras impares. Por ejemplo,
--    esReversible 36  == True
--    esReversible 409 == True
esReversible :: Int -> Bool
esReversible n = rem n 10 /= 0 && impares (cifras (n + (inverso n)))

-- (impares xs) se verifica si xs es una lista de números impares. Por
```

```haskell
-- ejemplo,
--    impares [3,5,1] == True
--    impares [3,4,1] == False
impares :: [Int] -> Bool
impares xs = and [odd x | x <- xs]

-- (inverso n) es el número obtenido escribiendo las cifras de n en
-- orden inverso. Por ejemplo,
--    inverso 3034 == 4303
inverso :: Int -> Int
inverso n = read (reverse (show n))

-- (cifras n) es la lista de las cifras del número n. Por ejemplo,
--    cifras 3034 == [3,0,3,4]
cifras :: Int -> [Int]
cifras n = [read [x] | x <- show n]

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir, usando funciones de orden superior, la función
--    grafoReducido :: Eq a => (a -> b) -> (a -> Bool) -> [a] -> [(a,b)]
-- tal que (grafoReducido f p xs) es la lista (sin repeticiones) de los
-- pares formados por los elementos de xs que verifican el predicado p
-- y sus imágenes. Por ejemplo,
--    grafoReducido (^2) even [1..9]  ==  [(2,4),(4,16),(6,36),(8,64)]
--    grafoReducido (+4) even (replicate 40 1) == []
--    grafoReducido (*5) even (replicate 40 2) == [(2,10)]
-- ---------------------------------------------------------------------

grafoReducido :: Eq a => (a -> b) -> (a -> Bool) -> [a] -> [(a,b)]
grafoReducido f p xs = [(x,f x) | x <- nub xs, p x]

-- ---------------------------------------------------------------------
-- Ejercicio 6.1. Un número natural n se denomina semiperfecto si es la
-- suma de algunos de sus divisores propios. Por ejemplo, 18 es
-- semiperfecto ya que sus divisores son 1, 2, 3, 6, 9 y se cumple que
-- 3+6+9=18.
--
-- Definir la función
--    esSemiPerfecto :: Int -> Bool
-- tal que (esSemiPerfecto n) se verifica si n es semiperfecto. Por
```

```haskell
-- ejemplo,
--    esSemiPerfecto 18 == True
--    esSemiPerfecto 9  == False
--    esSemiPerfecto 24 == True
-- ---------------------------------------------------------------------

esSemiPerfecto :: Int -> Bool
esSemiPerfecto n =
    or [sum ys == n | ys <- subconjuntos (divisores n)]

-- (divisores n) es la lista de los divisores propios de n. Por ejemplo,
--    divisores 18 == [1,2,3,6,9]
divisores :: Int -> [Int]
divisores n = [x | x <- [1..n-1], mod n x == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 6.2. Definir la constante primerSemiPerfecto tal que su
-- valor es el primer número semiperfecto.
-- ---------------------------------------------------------------------

primerSemiPerfecto :: Int
primerSemiPerfecto = head [n | n <- [1..], esSemiPerfecto n]

-- La evaluación es
--    *Main> primerSemiPerfecto
--    6

-- ---------------------------------------------------------------------
-- Ejercicio 6.3. Definir la función
--    semiPerfecto :: Int -> Int
-- tal que (semiPerfecto n) es el n-ésimo número semiperfecto. Por
-- ejemplo,
--    semiPerfecto 1   == 6
--    semiPerfecto 4   == 20
--    semiPerfecto 100 == 414
-- ---------------------------------------------------------------------

semiPerfecto :: Int -> Int
semiPerfecto n = semiPerfectos !! n
```

```haskell
-- semiPerfectos es la lista de los números semiPerfectos. Por ejemplo,
--    take 4 semiPerfectos  ==  [6,12,18,20]
semiPerfectos = [n | n <- [1..], esSemiPerfecto n]

-- ---------------------------------------------------------------------
-- Ejercicio 7.1. Definir mediante plegado la función
--    producto :: Num a => [a] -> a
-- tal que (producto xs) es el producto de los elementos de la lista
-- xs. Por ejemplo,
--    producto [2,1,-3,4,5,-6] == 720
-- ---------------------------------------------------------------------

producto :: Num a => [a] -> a
producto = foldr (*) 1

-- ---------------------------------------------------------------------
-- Ejercicio 7.2. Definir mediante plegado la función
--    productoPred :: Num a => (a -> Bool) -> [a] -> a
-- tal que (productoPred p xs) es el producto de los elementos de la
-- lista xs que verifican el predicado p. Por ejemplo,
--    productoPred even [2,1,-3,4,-5,6] == 48
-- ---------------------------------------------------------------------

productoPred :: Num a => (a -> Bool) -> [a] -> a
productoPred p = foldr (\x y -> if p x then x*y else y) 1

-- ---------------------------------------------------------------------
-- Ejercicio 7.3. Definir la función
--    productoPos :: (Num a, Ord a) => [a] -> a
-- tal que (productoPos xs) esel producto de los elementos estríctamente
-- positivos de la lista xs. Por ejemplo,
--    productoPos [2,1,-3,4,-5,6] == 48
-- ---------------------------------------------------------------------

productoPos :: (Num a, Ord a) => [a] -> a
productoPos = productoPred (>0)

-- ---------------------------------------------------------------------
-- Ejercicio 8. Las relaciones finitas se pueden representar mediante
-- listas de pares. Por ejemplo,
```

```
--    r1, r2, r3 :: [(Int, Int)]
--    r1 = [(1,3), (2,6), (8,9), (2,7)]
--    r2 = [(1,3), (2,6), (8,9), (3,7)]
--    r3 = [(1,3), (2,6), (8,9), (3,6)]
-- Definir la función
--    esFuncion :: (Eq a, Eq b) => [(a,b)] -> Bool
-- tal que (esFuncion r) se verifica si la relación r es una función (es
-- decir, a cada elemento del dominio de la relación r le corresponde un
-- único elemento). Por ejemplo,
--    esFuncion r1 == False
--    esFuncion r2 == True
--    esFuncion r3 == True
-- ---------------------------------------------------------------------

r1, r2, r3 :: [(Int, Int)]
r1 = [(1,3), (2,6), (8,9), (2,7)]
r2 = [(1,3), (2,6), (8,9), (3,7)]
r3 = [(1,3), (2,6), (8,9), (3,6)]

esFuncion :: (Eq a, Eq b) => [(a,b)] -> Bool
esFuncion [] = True
esFuncion ((x,y):r) =
    [y' | (x',y') <- r, x == x', y /= y'] == [] && esFuncion r

-- ---------------------------------------------------------------------
-- Ejercicio 9.1. Se denomina cola de una lista l a una sublista no
-- vacía de l formada por un elemento y los siguientes hasta el
-- final. Por ejemplo, [3,4,5] es una cola de la lista [1,2,3,4,5].
--
-- Definir la función
--    colas :: [a] -> [[a]]
-- tal que (colas xs) es la lista de las colas
-- de la lista xs. Por ejemplo,
--    colas []        == [[]]
--    colas [1,2]     == [[1,2],[2],[]]
--    colas [4,1,2,5] == [[4,1,2,5],[1,2,5],[2,5],[5],[]]
-- ---------------------------------------------------------------------

colas :: [a] -> [[a]]
colas []     = [[]]
```

```haskell
colas (x:xs) = (x:xs) : colas xs

-- ---------------------------------------------------------------------
-- Ejercicio 9.2. Comprobar con QuickCheck que las funciones colas y
-- tails son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_colas :: [Int] -> Bool
prop_colas xs = colas xs == tails xs

-- La comprobación es
--    *Main> quickCheck prop_colas
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 10.1. Se denomina cabeza de una lista l a una sublista no
-- vacía de la formada por el primer elemento y los siguientes hasta uno
-- dado. Por ejemplo, [1,2,3] es una cabeza de [1,2,3,4,5].
--
-- Definir la función
--    cabezas :: [a] -> [[a]]
-- tal que (cabezas xs) es la lista de las cabezas de la lista xs. Por
-- ejemplo,
--    cabezas []          == [[]]
--    cabezas [1,4]       == [[],[1],[1,4]]
--    cabezas [1,4,5,2,3] == [[],[1],[1,4],[1,4,5],[1,4,5,2],[1,4,5,2,3]]
-- ---------------------------------------------------------------------

-- 1. Por recursión
cabezas :: [a] -> [[a]]
cabezas []     = [[]]
cabezas (x:xs) = [] : [x:ys | ys <- cabezas xs]

-- 2. Usando patrones de plegado.
cabezasP :: [a] -> [[a]]
cabezasP = foldr (\x y -> [x]:[x:ys | ys <- y]) []

-- 3. Usando colas y funciones de orden superior.
cabezas3 :: [a] -> [[a]]
```

```haskell
cabezas3 xs = reverse (map reverse (colas (reverse xs)))

-- La anterior definición puede escribirse sin argumentos como
cabezas3' :: [a] -> [[a]]
cabezas3' = reverse . map reverse . (colas . reverse)

-- ---------------------------------------------------------------------
-- Ejercicio 10.2. Comprobar con QuickCheck que las funciones cabezas y
-- inits son equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_cabezas :: [Int] -> Bool
prop_cabezas xs = cabezas xs == inits xs

-- La comprobación es
--    *Main> quickCheck prop_cabezas
--    +++ OK, passed 100 tests.
```

Relación 14

El 2011 y los números primos

```haskell
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- Cada comienzo de año se suelen buscar propiedades numéricas del
-- número del año. En el 2011 se han buscado propiedades que relacionan
-- el 2011 y los números primos. En este ejercicio vamos a realizar la
-- búsqueda de dichas propiedades con Haskell.

import Data.List (sort)

-- La criba de Erastótenes es un método para calcular números primos. Se
-- comienza escribiendo todos los números desde 2 hasta (supongamos)
-- 100. El primer número (el 2) es primo. Ahora eliminamos todos los
-- múltiplos de 2. El primero de los números restantes (el 3) también es
-- primo. Ahora eliminamos todos los múltiplos de 3. El primero de los
-- números restantes (el 5) también es primo ... y así
-- sucesivamente. Cuando no quedan números, se han encontrado todos los
-- números primos en el rango fijado.

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    elimina :: Int -> [Int] -> [Int]
-- tal que (elimina n xs) es la lista obtenida eliminando en la lista xs
-- los múltiplos de n. Por ejemplo,
--    elimina 3 [2,3,8,9,5,6,7]  ==  [2,8,5,7]
-- ---------------------------------------------------------------------
```

```haskell
-- Por comprensión:
elimina :: Int -> [Int] -> [Int]
elimina n xs = [ x | x <- xs, x `mod` n /= 0 ]

-- Por recursión:
eliminaR :: Int -> [Int] -> [Int]
eliminaR n [] = []
eliminaR n (x:xs) | mod x n == 0 = eliminaR n xs
                  | otherwise    = x : eliminaR n xs

-- Por plegado:
eliminaP :: Int -> [Int] -> [Int]
eliminaP n = foldr f []
             where f x y | mod x n == 0 = y
                         | otherwise    = x:y

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    criba :: [Int] -> [Int]
-- tal que (criba xs) es la lista obtenida cribando la lista xs con el
-- método descrito anteriormente. Por ejemplo,
--    criba [2..20]          ==  [2,3,5,7,11,13,17,19]
--    take 10 (criba [2..])  ==  [2,3,5,7,11,13,17,19,23,29]
-- ---------------------------------------------------------------------

criba :: [Int] -> [Int]
criba []     = []
criba (n:ns) = n : criba (elimina n ns)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    primos :: [Int]
-- cuyo valor es la lista de los números primos. Por ejemplo,
--    take 10 primos  ==  [2,3,5,7,11,13,17,19,23,29]
-- ---------------------------------------------------------------------

primos :: [Int]
primos = criba [2..]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    esPrimo :: Int -> Bool
-- tal que (esPrimo n) se verifica si n es primo. Por ejemplo,
--    esPrimo 7  ==  True
--    esPrimo 9  ==  False
-- ---------------------------------------------------------------------

esPrimo :: Int -> Bool
esPrimo n = head (dropWhile (<n) primos) == n

-- ---------------------------------------------------------------------
-- Ejercicio 5. Comprobar que 2011 es primo.
-- ---------------------------------------------------------------------

-- La comprobación es
--     ghci> esPrimo 2011
--     True

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    prefijosConSuma :: [Int] -> Int -> [[Int]]
-- tal que (prefijosConSuma xs n) es la lista de los prefijos de xs cuya
-- suma es n. Por ejemplo,
--    prefijosConSuma [1..10] 3  == [[1,2]]
--    prefijosConSuma [1..10] 4  == []
-- ---------------------------------------------------------------------

prefijosConSuma :: [Int] -> Int -> [[Int]]
prefijosConSuma [] 0 = [[]]
prefijosConSuma [] n = []
prefijosConSuma (x:xs) n
    | x < n  = [x:ys | ys <- prefijosConSuma xs (n-x)]
    | x == n = [[x]]
    | x > n  = []

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    consecutivosConSuma :: [Int] -> Int -> [[Int]]
-- (consecutivosConSuma xs n) es la lista de los elementos consecutivos
```

```haskell
-- de xs cuya suma es n. Por ejemplo,
--    consecutivosConSuma [1..10] 9  == [[2,3,4],[4,5],[9]]
-- ---------------------------------------------------------------------

consecutivosConSuma :: [Int] -> Int -> [[Int]]
consecutivosConSuma [] 0 = [[]]
consecutivosConSuma [] n = []
consecutivosConSuma (x:xs) n =
    (prefijosConSuma (x:xs) n) ++ (consecutivosConSuma xs n)

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    primosConsecutivosConSuma :: Int -> [[Int]]
-- tal que (primosConsecutivosConSuma n) es la lista de los números
-- primos consecutivos cuya suma es n. Por ejemplo,
--    ghci> primosConsecutivosConSuma 41
--    [[2,3,5,7,11,13],[11,13,17],[41]]
-- ---------------------------------------------------------------------

primosConsecutivosConSuma :: Int -> [[Int]]
primosConsecutivosConSuma n =
    consecutivosConSuma (takeWhile (<=n) primos) n

-- ---------------------------------------------------------------------
-- Ejercicio 9. Calcular las descomposiciones de 2011 como sumas de
-- primos consecutivos.
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> primosConsecutivosConSuma 2011
--    [[157,163,167,173,179,181,191,193,197,199,211],[661,673,677],[2011]]

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    propiedad1 :: Int -> Bool
-- tal que (propiedad1 n) se verifica si n sólo se puede expresar como
-- sumas de 1, 3 y 11 primos consecutivos. Por ejemplo,
--    propiedad1 2011  ==  True
--    propiedad1 2010  ==  False
-- ---------------------------------------------------------------------
```

```haskell
propiedad1 :: Int -> Bool
propiedad1 n =
    sort (map length (primosConsecutivosConSuma n)) == [1,3,11]

-- ---------------------------------------------------------------------
-- Ejercicio 11. Calcular los años hasta el 3000 que cumplen la
-- propiedad1.
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> [n | n <- [1..3000], propiedad1 n]
--    [883,2011]

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    sumaCifras :: Int -> Int
-- tal que (sumaCifras x) es la suma de las cifras del número x. Por
-- ejemplo,
--    sumaCifras 254  ==  11
-- ---------------------------------------------------------------------

sumaCifras :: Int -> Int
sumaCifras x = sum [read [y] | y <- show x]

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la función
--    sumaCifrasLista :: [Int] -> Int
-- tal que (sumaCifrasLista xs) es la suma de las cifras de la lista de
-- números xs. Por ejemplo,
--    sumaCifrasLista [254, 61]  ==  18
-- ---------------------------------------------------------------------

-- Por comprensión:
sumaCifrasLista :: [Int] -> Int
sumaCifrasLista xs = sum [sumaCifras y | y <- xs]

-- Por recursión:
sumaCifrasListaR :: [Int] -> Int
sumaCifrasListaR [] = 0
```

```haskell
sumaCifrasListaR (x:xs) = sumaCifras x + sumaCifrasListaR xs

-- Por plegado:
sumaCifrasListaP :: [Int] -> Int
sumaCifrasListaP = foldr f 0
                   where f x y = sumaCifras x + y

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función
--    propiedad2 :: Int -> Bool
-- tal que (propiedad2 n) se verifica si n puede expresarse como suma de
-- 11 primos consecutivos y la suma de las cifras de los 11 sumandos es
-- un número primo. Por ejemplo,
--    propiedad2 2011  ==  True
--    propiedad2 2000  ==  False
-- ---------------------------------------------------------------------

propiedad2 :: Int -> Bool
propiedad2 n = [xs | xs <- primosConsecutivosConSuma n,
                     length xs == 11,
                     esPrimo (sumaCifrasLista xs)]
               /= []

-- ---------------------------------------------------------------------
-- Ejercicio 15. Calcular el primer año que cumple la propiedad1 y la
-- propiedad2.
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> head [n | n <- [1..], propiedad1 n, propiedad2 n]
--    2011

-- ---------------------------------------------------------------------
-- Ejercicio 16. Definir la función
--    propiedad3 :: Int -> Bool
-- tal que (propiedad3 n) se verifica si n puede expresarse como suma de
-- tantos números consecutivos como indican sus dos últimas cifras. Por
-- ejemplo,
--    propiedad3 2011  ==  True
--    propiedad3 2000  ==  False
```

```haskell
-- ---------------------------------------------------------------------

propiedad3 :: Int -> Bool
propiedad3 n = [xs | xs <- primosConsecutivosConSuma n,
                     length xs == a]
               /= []
    where a = mod n 100

-- ---------------------------------------------------------------------
-- Ejercicio 17. Calcular el primer año que cumple la propiedad1 y la
-- propiedad3.
-- ---------------------------------------------------------------------

-- El cálculo es
--    ghci> head [n | n <- [1..], propiedad1 n, propiedad3 n]
--    2011

-- Hemos comprobado que 2011 es el menor número que cumple las
-- propiedades 1 y 1 y también es el menor número que cumple las
-- propiedades 1 y 3.
```

Relación 15

Listas infinitas

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En esta relación se presentan ejercicios con listas infinitas y
-- evaluación perezosa. En concreto, se estudian funciones para calcular
-- * la lista de las potencias de un número menores que otro dado,
-- * la lista obtenida repitiendo un elemento infinitas veces,
-- * la lista obtenida repitiendo un elemento un número finito de veces,
-- * la cadena obtenida cada elemento tantas veces como indica su
--   posición,
-- * la aplicación iterada de una función a un elemento,
-- * la lista de las sublistas de longitud dada y
-- * la sucesión de Collatz.
--
-- Estos ejercicios corresponden al tema 10 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-10.pdf

-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir, usando takeWhile y map, la función
```

```haskell
--    potenciasMenores :: Int -> Int -> [Int]
-- tal que (potenciasMenores x y) es la lista de las potencias de x
-- menores que y. Por ejemplo,
--    potenciasMenores 2 1000  ==  [2,4,8,16,32,64,128,256,512]
-- ---------------------------------------------------------------------

potenciasMenores :: Int -> Int -> [Int]
potenciasMenores x y = takeWhile (<y) (map (x^) [1..])

-- Otra definición:
potenciasMenores' :: Int -> Int -> [Int]
potenciasMenores' x y = takeWhile (<y) (map (x^) [1..])

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir, por recursión y comprensión, la función
--    repite :: a -> [a]
-- tal que (repite x) es la lista infinita cuyos elementos son x. Por
-- ejemplo,
--    repite 5           == [5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,...
--    take 3 (repite 5)  ==  [5,5,5]
-- Nota: La función repite es equivalente a la función repeat definida en
-- el preludio de Haskell.
-- ---------------------------------------------------------------------

-- Por recursión:
repite :: a -> [a]
repite x = x : repite x

-- Por comprensión:
repite' x = [x | _ <- [1..]]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir, por recursión y por comprensión, la función
--    repiteFinita :: Int-> a -> [a]
-- tal que (repite n x) es la lista con n elementos iguales a x. Por
-- ejemplo,
--    repiteFinita 3 5  ==  [5,5,5]
-- Nota: La función repite es equivalente a la función replicate definida
-- en el preludio de Haskell.
-- ---------------------------------------------------------------------
```

```haskell
-- Por recursión:
repiteFinita :: Int -> a -> [a]
repiteFinita n x = take n (repite x)

-- Por comprensión:
repiteFinita' :: Int -> a -> [a]
repiteFinita' n x = [x | _ <- [1..n]]

-- También se puede definir usando repite
repiteFinita2 :: Int -> a -> [a]
repiteFinita2 n x = take n (repite x)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Se considera la función
--    eco :: String -> String
-- tal que (eco xs) es la cadena obtenida a partir de la cadena xs
-- repitiendo cada elemento tantas veces como indica su posición: el
-- primer elemento se repite 1 vez, el segundo 2 veces y así
-- sucesivamente. Por ejemplo,
--    eco "abcd"  ==  "abbcccdddd"
-- 1. Escribir una definición 'no recursiva' de la función eco.
-- 2. Escribir una definición 'recursiva' de la función eco.
-- ---------------------------------------------------------------------

-- Una definición no recursiva es
ecoNR :: String -> String
ecoNR xs = concat [replicate i x | (i,x) <- zip [1..] xs]

-- Una definición recursiva es
ecoR :: String -> String
ecoR xs =
    ecoRaux 1 0 xs
    where
      ecoRaux i n []  =  []
      ecoRaux i n (c:cs) | i <= n     =  ecoRaux (i+1) 0 cs
                         | otherwise  =  c : (ecoRaux i (n+1) (c:cs))

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir, por recursión, la función
```

```haskell
--    itera :: (a -> a) -> a -> [a]
-- tal que (itera f x) es la lista cuyo primer elemento es x y los
-- siguientes elementos se calculan aplicando la función f al elemento
-- anterior. Por ejemplo,
--    Main> itera (+1) 3
--    [3,4,5,6,7,8,9,10,11,12,{Interrupted!}
--    Main> itera (*2) 1
--    [1,2,4,8,16,32,64,{Interrupted!}
--    Main> itera (`div` 10) 1972
--    [1972,197,19,1,0,0,0,0,0,0,{Interrupted!}
-- Nota: La función repite es equivalente a la función iterate definida
-- en el preludio de Haskell.
-- ---------------------------------------------------------------------

itera :: (a -> a) -> a -> [a]
itera f x = x : itera f (f x)

-- ---------------------------------------------------------------------------
-- Ejercicio 6, Definir la función
--    agrupa :: Int -> [a] -> [[a]]
-- tal que (agrupa n xs) es la lista de las sublistas de longitud n de
-- la lista xs. Por ejemplo,
--    Main> agrupa 2 [3,1,5,8,2,7]
--    [[3,1],[5,8],[2,7]]
--    Main> agrupa 2 [3,1,5,8,2,7,9]
--    [[3,1],[5,8],[2,7],[9]]
--    Main> agrupa 5 "todo necio confunde valor y precio"
--    ["todo ","necio"," conf","unde ","valor"," y pr","ecio"]
-- ---------------------------------------------------------------------------

-- Una definición no recursiva es
agrupa :: Int -> [a] -> [[a]]
agrupa n = takeWhile (not . null)
         . map (take n)
         . iterate (drop n)

-- Puede verse su funcionamiento en el siguiente ejemplo,
--    iterate (drop 2) [5..10]
--    ==> [[5,6,7,8,9,10],[7,8,9,10],[9,10],[],[],...
--    map (take 2) (iterate (drop 2) [5..10])
```

```haskell
--    ==> [[5,6],[7,8],[9,10],[],[],[],[],...
--    takeWhile (not . null) (map (take 2) (iterate (drop 2) [5..10]))
--    ==> [[5,6],[7,8],[9,10]]

-- Una definición recursiva de agrupa es
agrupa' :: Int -> [a] -> [[a]]
agrupa' n [] = []
agrupa' n xs = take n xs : agrupa' n (drop n xs)

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir, y comprobar, con QuickCheck las dos propiedades
-- que caracterizan a la función agrupa:
-- * todos los grupos tienen que tener la longitud determinada (salvo el
--   último que puede tener una longitud menor) y
-- * combinando todos los grupos se obtiene la lista inicial.
-- ---------------------------------------------------------------------

-- La primera propiedad es
prop_AgrupaLongitud :: Int -> [Int] -> Property
prop_AgrupaLongitud n xs =
    n > 0 && not (null gs) ==>
      and [length g == n | g <- init gs] &&
      0 < length (last gs) && length (last gs) <= n
    where
      gs = agrupa n xs

-- La comprobación es
--    Main> quickCheck prop_AgrupaLongitud
--    OK, passed 100 tests.

-- La segunda propiedad es
prop_AgrupaCombina :: Int -> [Int] -> Property
prop_AgrupaCombina n xs =
    n > 0 ==>
      concat (agrupa n xs) == xs

-- La comprobación es
--    Main> quickCheck prop_AgrupaCombina
--    OK, passed 100 tests.
```

```haskell
-- ---------------------------------------------------------------------
-- Sea la siguiente operación, aplicable a cualquier número entero
-- positivo:
--    * Si el número es par, se divide entre 2.
--    * Si el número es impar, se multiplica por 3 y se suma 1.
-- Dado un número cualquiera, podemos considerar su órbita, es decir,
-- las imágenes sucesivas al iterar la función. Por ejemplo, la órbita
-- de 13 es
--    13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1,...
-- Si observamos este ejemplo, la órbita de 13 es periódica, es decir,
-- se repite indefinidamente a partir de un momento dado). La conjetura
-- de Collatz dice que siempre alcanzaremos el 1 para cualquier número
-- con el que comencemos. Ejemplos:
--    * Empezando en n = 6 se obtiene 6, 3, 10, 5, 16, 8, 4, 2, 1.
--    * Empezando en n = 11 se obtiene: 11, 34, 17, 52, 26, 13, 40, 20,
--      10, 5, 16, 8, 4, 2, 1.
--    * Empezando en n = 27, la sucesión tiene 112 pasos, llegando hasta
--      9232 antes de descender a 1:  27, 82, 41, 124, 62, 31, 94, 47,
--      142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274,
--      137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263,
--      790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502,
--      251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958,
--      479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644,
--      1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308,
--      1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122,
--      61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5,
--      16, 8, 4, 2, 1.

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    siguiente :: Integer -> Integer
-- tal que (siguiente n) es el siguiente de n en la sucesión de
-- Collatz. Por ejemplo,
--    siguiente 13  ==  40
--    siguiente 40  ==  20
-- ---------------------------------------------------------------------

siguiente n | even n    = n `div` 2
            | otherwise = 3*n+1
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir, por recursión, la función
--    collatz :: Integer -> [Integer]
-- tal que (collatz n) es la órbita de Collatz d n hasta alcanzar el
-- 1. Por ejemplo,
--    collatz 13  ==  [13,40,20,10,5,16,8,4,2,1]
-- ---------------------------------------------------------------------

collatz :: Integer -> [Integer]
collatz 1 = [1]
collatz n = n : collatz (siguiente n)

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir, sin recursión, la función
--    collatz' :: Integer -> [Integer]
-- tal que (collatz' n) es la órbita de Collatz d n hasta alcanzar el
-- 1. Por ejemplo,
--    collatz' 13  ==  [13,40,20,10,5,16,8,4,2,1]
-- Indicación: Usar takeWhile e iterate.
-- ---------------------------------------------------------------------

collatz' :: Integer -> [Integer]
collatz' n = (takeWhile (/=1) (iterate siguiente n)) ++ [1]

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--    menorCollatzMayor :: Int -> Integer
-- tal que (menorCollatzMayor x) es el menor número cuya órbita de
-- Collatz tiene más de x elementos. Por ejemplo,
--    menorCollatzMayor 100  ==  27
-- ---------------------------------------------------------------------

menorCollatzMayor :: Int -> Integer
menorCollatzMayor x = head [y | y <- [1..], length (collatz y) > x]

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    menorCollatzSupera :: Integer -> Integer
-- tal que (menorCollatzSupera x) es el menor número cuya órbita de
-- Collatz tiene algún elemento mayor que x. Por ejemplo,
```

```haskell
--    menorCollatzSupera 100  ==  15
-- ---------------------------------------------------------------------

menorCollatzSupera :: Integer -> Integer
menorCollatzSupera x =
    head [y | y <- [1..], maximum (collatz y) > x]

-- Otra definición alternativa es
menorCollatzSupera' :: Integer -> Integer
menorCollatzSupera' x = head [n | n <- [1..], t <- collatz' n, t > x]
```

Relación 16

Tipos de datos algebraicos. Misceláneas

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En esta relación se presenta ejercicios sobre tipos de datos
-- algebraicos y una miscelánea de ejercicios. Se consideran dos tipos
-- de datos algebraicos: los números naturales (para los que se define
-- su producto) y los árboles binarios, para los que se definen
-- funciones para calcular:
--   * la ocurrencia de un elemento en el árbol,
--   * el número de hojas
--   * el carácter balanceado de un árbol,
--   * el árbol balanceado correspondiente a una lista,
--
-- En la miscelánea, se plantean
-- * problemas matemáticos para calcular
--   * la suma de los múltiplos de 3 ó 5 menores que un número,
--   * el límite de una sucesión,
-- * problemas sobre listas de listas para calcular
--   * el producto cartesiano de una lista de conjuntos,
--   * la verificación de una propiedad por todos los elemento de una
--     lista de listas verifican una propiedad.
--   * la lista de las listas de mayor suma,
-- * problemas sobre listas infinitas para calcular
--   * la pertenencia de un elemento a una lista creciente,
```

```
--   * la lista ordenada de los pares de números naturales,
--
-- Los ejercicios de la primera parte corresponden al tema 9 cuyas
-- transparencias se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-9.pdf

-- ---------------------------------------------------------------------
-- 1. Tipos de datos algebraicos                                       --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Usando el tipo de dato Nat y la función suma definidas
-- en las transparencias del tema 9, definir la función
--    producto :: Nat -> Nat -> Nat
-- tal que (producto m n) es el producto de los números naturales m y
-- n. Por ejemplo,
--    *Main> producto (Suc (Suc Cero)) (Suc (Suc (Suc Cero)))
--    Suc (Suc (Suc (Suc (Suc (Suc Cero)))))
-- ---------------------------------------------------------------------

data Nat = Cero | Suc Nat
           deriving (Eq, Show)

suma :: Nat -> Nat -> Nat
suma Cero    n = n
suma (Suc m) n = Suc (suma m n)

producto :: Nat -> Nat -> Nat
producto Cero _    = Cero
producto (Suc m) n = suma n (producto m n)

-- ---------------------------------------------------------------------
-- Nota. En los siguientes ejercicios se trabajará con árboles binarios
-- definidos como sigue
--    data Arbol = Hoja Int
--               | Nodo Arbol Int Arbol
--               deriving (Show, Eq)
-- Por ejemplo, el árbol
--         5
--        / \
```

```
--       /   \
--      3     7
--     / \   / \
--    1   4 6   9
-- se representa por
--    Nodo (Nodo (Hoja 1) 3 (Hoja 4))
--         5
--         (Nodo (Hoja 6) 7 (Hoja 9))
-- ----------------------------------------------------------------------

data Arbol = Hoja Int
           | Nodo Arbol Int Arbol
           deriving (Show, Eq)

ejArbol :: Arbol
ejArbol = Nodo (Nodo (Hoja 1) 3 (Hoja 4))
               5
               (Nodo (Hoja 6) 7 (Hoja 9))

-- ----------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    ocurre :: Int -> Arbol -> Bool
-- tal que (ocurre x a) se verifica si x ocurre en el árbol a como valor
-- de un nodo o de una hoja. Por ejemplo,
--    ocurre  4 ejArbol  ==  True
--    ocurre 10 ejArbol  ==  False
-- ----------------------------------------------------------------------

ocurre :: Int -> Arbol -> Bool
ocurre m (Hoja n)     = m == n
ocurre m (Nodo i n d) = m == n || ocurre m i || ocurre m d

-- ----------------------------------------------------------------------
-- Ejercicio 3. En el preludio está definido el tipo de datos
--    data Ordering = LT | EQ | GT
-- junto con la función
--    compare :: Ord a => a -> a -> Ordering
-- que decide si un valor en un tipo ordenado es menor (LT), igual (EQ)
-- o mayor (GT) que otro.
--
```

```haskell
-- Usando esta función, redefinir la función
--    ocurre :: Int -> Arbol -> Bool
-- del ejercicio anterior.
-- ---------------------------------------------------------------------

ocurre' :: Int -> Arbol -> Bool
ocurre' m (Hoja n)     = m == n
ocurre' m (Nodo i n d) = case compare m n of
                           LT -> ocurre' m i
                           EQ -> True
                           GT -> ocurre' m d

-- ---------------------------------------------------------------------
-- Ejercicio 4. ¿Porqué la segunda definición de ocurre es más eficiente
-- que la primera?
-- ---------------------------------------------------------------------

-- La nueva definición es más eficiente porque sólo necesita una
-- comparación por nodo, mientras que la definición de las
-- transparencias necesita dos comparaciones por nodo.

-- ---------------------------------------------------------------------
-- Nota. En los siguientes ejercicios se trabajará con árboles binarios
-- definidos como sigue
--    type ArbolB = HojaB Int
--                | NodoB ArbolB ArbolB
--                deriving Show
-- Por ejemplo, el árbol
--         .
--        / \
--       /   \
--      .     .
--     / \   / \
--    1   4 6   9
-- se representa por
--    NodoB (NodoB (HojaB 1) (HojaB 4))
--          (NodoB (HojaB 6) (HojaB 9))
-- ---------------------------------------------------------------------

data ArbolB = HojaB Int
```

```
            | NodoB ArbolB ArbolB
            deriving Show

ejArbolB :: ArbolB
ejArbolB = NodoB (NodoB (HojaB 1) (HojaB 4))
                 (NodoB (HojaB 6) (HojaB 9))

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    nHojas :: ArbolB -> Int
-- tal que (nHojas a) es el número de hojas del árbol a. Por ejemplo,
--    nHojas (NodoB (HojaB 5) (NodoB (HojaB 3) (HojaB 7)))  ==  3
--    nHojas ejArbolB ==  4
-- ---------------------------------------------------------------------

nHojas :: ArbolB -> Int
nHojas (HojaB _)     = 1
nHojas (NodoB a1 a2) = nHojas a1 + nHojas a2

-- ---------------------------------------------------------------------
-- Ejercicio 6. Se dice que un árbol de este tipo es balanceado si es
-- una hoja o bien si para cada nodo se tiene que el número de hojas en
-- cada uno de sus subárboles difiere como máximo en uno y sus
-- subárboles son balanceados. Definir la función
--    balanceado :: ArbolB -> BoolB
-- tal que (balanceado a) se verifica si a es un árbol balanceado. Por
-- ejemplo,
--    bakaceado ejArbolB
--    ==> True
--    balanceado (NodoB (HojaB 5) (NodoB (HojaB 3) (HojaB 7)))
--    ==> True
--    balanceado (NodoB (HojaB 5) (NodoB (HojaB 3) (NodoB (HojaB 5) (HojaB 7))))
--    ==> False
-- ---------------------------------------------------------------------

balanceado :: ArbolB -> Bool
balanceado (HojaB _)     = True
balanceado (NodoB a1 a2) = abs (nHojas a1 - nHojas a2) <= 1 &&
                           balanceado a1 &&
                           balanceado a2
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    mitades :: [a] -> ([a],[a])
-- tal que (mitades xs) es un par de listas que se obtiene al dividir xs
-- en dos mitades cuya longitud difiere como máximo en uno. Por ejemplo,
--    mitades [2,3,5,1,4,7]    ==  ([2,3,5],[1,4,7])
--    mitades [2,3,5,1,4,7,9]  ==  ([2,3,5],[1,4,7,9])
-- ---------------------------------------------------------------------

mitades :: [a] -> ([a],[a])
mitades xs = splitAt (length xs `div` 2) xs

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    arbolBalanceado :: [Int] -> ArbolB
-- tal que (arbolBalanceado xs) es el árbol balanceado correspondiente
-- a la lista xs. Por ejemplo,
--    *Main> arbolBalanceado [2,5,3]
--    NodoB (HojaB 2) (NodoB (HojaB 5) (HojaB 3))
--    *Main> arbolBalanceado [2,5,3,7]
--    NodoB (NodoB (HojaB 2) (HojaB 5)) (NodoB (HojaB 3) (HojaB 7))
-- ---------------------------------------------------------------------

arbolBalanceado :: [Int] -> ArbolB
arbolBalanceado [x] = HojaB x
arbolBalanceado xs  = NodoB (arbolBalanceado ys) (arbolBalanceado zs)
                      where (ys,zs) = mitades xs

-- ---------------------------------------------------------------------
-- 2. Miscelánea                                                       --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 9. Los números naturales menores que 10 que son múltiplos
-- de 3 ó 5 son 3, 5, 6 y 9. La suma de estos múltiplos es 23. Definir
-- la función
--    sumaMultiplosMenores :: Integer -> Integer
-- tal que (sumaMultiplosMenores n) es la suma de todos los múltiplos de
-- 3 ó 5 menores que n. Por ejemplo,
```

```haskell
--    sumaMultiplosMenores 10  ==  23
-- Calcular la suma de todos los múltiplos de 3 ó 5 menores que 1000,
-- indicando el tiempo y el espacio empleado.
-- ---------------------------------------------------------------------

sumaMultiplosMenores :: Integer -> Integer
sumaMultiplosMenores n =
    sum [x | x <- [1..n-1], multiplo x 3 || multiplo x 5]
    where multiplo x y = mod x y == 0

-- Cálculo:
--    *Main> :set +s
--    *Main> sumaMultiplosMenores 1000
--    233168
--    (0.03 secs, 1053460 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    productoCartesiano :: [[a]] -> [[a]]
-- tal que (productoCartesiano xss) es el producto cartesiano de los conjuntos
-- xss. Por ejemplo,
--    *Main> productoCartesiano [[1,3],[2,5]]
--    [[1,2],[1,5],[3,2],[3,5]]
--    *Main> productoCartesiano [[1,3],[2,5],[6,4]]
--    [[1,2,6],[1,2,4],[1,5,6],[1,5,4],[3,2,6],[3,2,4],[3,5,6],[3,5,4]]
--    *Main> productoCartesiano [[1,3,5],[2,4]]
--    [[1,2],[1,4],[3,2],[3,4],[5,2],[5,4]]
--    *Main> productoCartesiano []
--    [[]]
-- ---------------------------------------------------------------------

productoCartesiano :: [[a]] -> [[a]]
productoCartesiano []       = [[]]
productoCartesiano (xs:xss) =
    [x:ys | x <- xs, ys <- productoCartesiano xss]

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir (por recursión, plegado y comprensión)  el
-- predicado
--    comprueba :: [[Int]] -> Bool
```

```haskell
-- tal que tal que (comprueba xss) se verifica si cada elemento de la
-- lista de listas xss contiene algún número par. Por ejemplo,
--    comprueba [[1,2],[3,4,5],[8]]  ==  True
--    comprueba [[1,2],[3,5]]        ==  False
-- ---------------------------------------------------------------------

-- La definición por recursión es
compruebaR :: [[Int]] -> Bool
compruebaR [] = True
compruebaR (xs:xss) = tienePar xs && compruebaR xss

-- (tienePar xs) se verifica si xs contiene algún número par.
tienePar  :: [Int] -> Bool
tienePar []     = False
tienePar (x:xs) = even x || tienePar xs

-- La definición por plegado es
compruebaP :: [[Int]] -> Bool
compruebaP = foldr f True
    where f x y = tienePar x && y

-- La definición por comprensión es
compruebaC :: [[Int]] -> Bool
compruebaC xss = and [or [even x | x <- xs] | xs <- xss]

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    pertenece :: Ord a => a -> [a] -> Bool
-- tal que (pertenece x ys) se verifica si x pertenece a la lista
-- ordenada creciente, finita o infinita, ys. Por ejemplo,
--    pertenece 22 [1,3,22,34]  ==>  True
--    pertenece 22 [1,3,34]     ==>  False
--    pertenece 23 [1,3..]      ==>  True
--    pertenece 22 [1,3..]      ==>  False
-- ---------------------------------------------------------------------

pertenece :: Ord a => a -> [a] -> Bool
pertenece _ [] = False
pertenece x (y:ys) | x >  y    = pertenece x ys
                   | x == y    = True
```

```
                  | otherwise = False

-- La definición de pertenece puede simplificarse
pertenece' :: Ord a => a -> [a] -> Bool
pertenece' x ys = elem x (takeWhile (<= x) ys)

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la función
--    listasMayores :: [[Int]] -> [[Int]]
-- tal que (listasMayores xss) es la lista de las listas de xss de mayor
-- suma. Por ejemplo,
--    *Main> listasMayores [[1,3,5],[2,7],[1,1,2],[3],[5]]
--    [[1,3,5],[2,7]]
-- ---------------------------------------------------------------------

listasMayores :: [[Int]] -> [[Int]]
listasMayores xss = [xs | xs <- xss, sum xs == m]
    where m = maximum [sum xs | xs <- xss]

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la constante
--    pares :: Int
-- tal que pares es la lista de todos los pares de números enteros
-- positivos ordenada según la suma de sus componentes y el valor de la
-- primera componente. Por ejemplo,
--    *Main> take 11 pares
--    [(1,1),(1,2),(2,1),(1,3),(2,2),(3,1),(1,4),(2,3),(3,2),(4,1),(1,5)]
-- ---------------------------------------------------------------------

pares :: [(Integer,Integer)]
pares = [(x,z-x) | z <- [1..], x <- [1..z-1]]

-- ---------------------------------------------------------------------
-- Ejercicio 15. Definir la constante
--    paresDestacados :: [(Integer,Integer)]
-- tal que paresDestadados es la lista de pares de números enteros (x,y)
-- tales que 11 divide a x+13y y 13 divide a x+11y.
-- ---------------------------------------------------------------------

paresDestacados :: [(Integer,Integer)]
```

```haskell
paresDestacados = [(x,y) | (x,y) <- pares,
                           mod (x+13*y) 11 == 0,
                           mod (x+11*y) 13 == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 16. Definir la constante
--    parDestacadoConMenorSuma :: Integer
-- tal que pardestacadoconmenorsuma es el par destacado con menor suma y
-- calcular su valor y su posición en la lista pares.
-- ---------------------------------------------------------------------

-- La definición es
parDestacadoConMenorSuma :: (Integer,Integer)
parDestacadoConMenorSuma = head paresDestacados

-- El valor es
--   *Main> parDestacadoConMenorSuma
--   (23,5)

-- La posición es
--    *Main> 1 + length (takeWhile (/=parDestacadoConMenorSuma) pares)
--    374

-- ---------------------------------------------------------------------
-- Ejercicio 17. Definir la función
--    limite :: (Num a, Enum a, Num b, Ord b) => (a -> b) -> b -> b
-- tal que (limite f a) es el valor de f en el primer término x tal que
-- para todo y entre x+1 y x+100, el valor absoluto de f(y)-f(x) es
-- menor que a. Por ejemplo,
--    limite (\n -> (2*n+1)/(n+5)) 0.001  ==  1.9900110987791344
--    limite (\n -> (1+1/n)**n) 0.001     ==  2.714072874546881
-- ---------------------------------------------------------------------

limite :: (Num a, Enum a, Num b, Ord b) => (a -> b) -> b -> b
limite f a =
    head [f x | x <- [1..],
                maximum [abs(f y - f x) | y <- [x+1..x+100]] < a]

-- ---------------------------------------------------------------------
-- Ejercicio 18. Definir la función
```

```haskell
--    esLimite :: (Num a, Enum a, Num b, Ord b) => (a -> b) -> b -> b -> Bool
-- tal que (esLimite f b a) se verifica si existe un x tal que para todo
-- y entre x+1 y x+100, el valor absoluto de f(y)-b es menor que a. Por
-- ejemplo,
--    esLimite (\n -> (2*n+1)/(n+5)) 2 0.01     ==  True
--    esLimite (\n -> (1+1/n)**n) (exp 1) 0.01  ==  True
-- ------------------------------------------------------------------

esLimite :: (Num a, Enum a, Num b, Ord b) => (a -> b) -> b -> b -> Bool
esLimite f b a =
    not (null [x | x <- [1..],
                   maximum [abs(f y - b) | y <- [x+1..x+100]] < a])
```

Relación 17

Tipos de datos algebraicos: árboles binarios y fórmulas

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En esta relación se plantean ejercicios sobre tipos de datos
-- algebraicos. En concreto, sobre árboles binarios y sobre fórmulas
-- proposicionales. En el caso de los árboles, se extienden funciones de
-- listas a árboles. En el caso de las fórmulas, se extiende el
-- demostrador proposicional para incluir disyunciones y equivalencias.
--
-- Estos ejercicios corresponden al tema 9 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-

-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- 1. Árboles binarios                                                 --
-- ---------------------------------------------------------------------
```

```haskell
-- ---------------------------------------------------------------------
-- Nota. En los siguientes ejercicios se pide adaptar funciones sobre
-- lista a funciones sobre árboles definidos por
--    data Arbol a = Hoja | Nodo (Arbol a) a (Arbol a) deriving Show
-- ---------------------------------------------------------------------

data Arbol a = Hoja
             | Nodo (Arbol a) a (Arbol a)
             deriving Show

-- ---------------------------------------------------------------------
-- Ejercicio 1. La función take está definida por
--    take :: Int -> [a] -> [a]
--    take 0            = []
--    take (n+1) []     = []
--    take (n+1) (x:xs) = x : take n xs
-- Definir la función
--    takeArbol ::  Int -> Arbol a -> Arbol a
-- tal que (takeArbol n t) es el subárbol de t de profundidad n. Por
-- ejemplo,
--    *Main> takeArbol 0 (Nodo Hoja 6 (Nodo (Nodo Hoja 5 Hoja) 7 Hoja))
--    Hoja
--    *Main> takeArbol 1 (Nodo Hoja 6 (Nodo (Nodo Hoja 5 Hoja) 7 Hoja))
--    Nodo Hoja 6 Hoja
--    *Main> takeArbol 2 (Nodo Hoja 6 (Nodo (Nodo Hoja 5 Hoja) 7 Hoja))
--    Nodo Hoja 6 (Nodo Hoja 7 Hoja)
--    *Main> takeArbol 3 (Nodo Hoja 6 (Nodo (Nodo Hoja 5 Hoja) 7 Hoja))
--    Nodo Hoja 6 (Nodo (Nodo Hoja 5 Hoja) 7 Hoja)
--    *Main> takeArbol 4 (Nodo Hoja 6 (Nodo (Nodo Hoja 5 Hoja) 7 Hoja))
--    Nodo Hoja 6 (Nodo (Nodo Hoja 5 Hoja) 7 Hoja)
-- ---------------------------------------------------------------------

takeArbol 0     _            = Hoja
takeArbol (n+1) Hoja         = Hoja
takeArbol (n+1) (Nodo l x r) = Nodo (takeArbol n l) x (takeArbol n r)

-- ---------------------------------------------------------------------
-- Ejercicio 2. La función
--    repeat :: a -> [a]
-- está definida de forma que (repeat x) es la lista formada por
```

```haskell
-- infinitos elementos x. Por ejemplo,
--    repeat 3  ==>  [3,3,3,3,3,3,3,3,3,3,3,3,...
-- La definición de repeat es
--    repeat x = xs where xs = x:xs
-- Definir la función
--    repeatArbol :: a -> Arbol a
-- tal que (repeatArbol x) es es árbol con infinitos nodos x. Por
-- ejemplo,
--    Main> takeArbol 0 (repeatArbol 3)
--    Hoja
--    *Main> takeArbol 1 (repeatArbol 3)
--    Nodo Hoja 3 Hoja
--    *Main> takeArbol 2 (repeatArbol 3)
--    Nodo (Nodo Hoja 3 Hoja) 3 (Nodo Hoja 3 Hoja)
-- ---------------------------------------------------------------------

repeatArbol :: a -> Arbol a
repeatArbol x = Nodo t x t
                where t = repeatArbol x

-- ---------------------------------------------------------------------
-- Ejercicio 3. La función
--    replicate :: Int -> a -> [a]
-- está definida por
--    replicate n = take n . repeat
-- es tal que (replicate n x) es la lista de longitud n cuyos elementos
-- son x. Por ejemplo,
--    replicate 3 5  ==>  [5,5,5]
-- Definir la función
--    replicateArbol :: Int -> a -> Arbol a
-- tal que (replicate n x) es el árbol de profundidad n cuyos nodos son
-- x. Por ejemplo,
--    *Main> replicateArbol 0 5
--    Hoja
--    *Main> replicateArbol 1 5
--    Nodo Hoja 5 Hoja
--    *Main> replicateArbol 2 5
--    Nodo (Nodo Hoja 5 Hoja) 5 (Nodo Hoja 5 Hoja)
-- ---------------------------------------------------------------------
```

```haskell
replicateArbol :: Int -> a -> Arbol a
replicateArbol n = takeArbol n . repeatArbol

-- ---------------------------------------------------------------------
-- 2. Fórmulas proposicionales                                         --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 4. Extender el procedimiento de decisión de tautologías
-- para incluir las disyunciones (Disj) y las equivalencias (Equi). Por
-- ejemplo,
--    *Main> esTautologia (Equi (Var 'A') (Disj (Var 'A') (Var 'A')))
--    True
--    *Main> esTautologia (Equi (Var 'A') (Disj (Var 'A') (Var 'B')))
--    False
-- Se incluye el código del procedimiento visto en clase para que se
-- extienda de manera adecuada.
-- ---------------------------------------------------------------------

data FProp = Const Bool
           | Var Char
           | Neg FProp
           | Conj FProp FProp
           | Disj FProp FProp   -- Añadido
           | Impl FProp FProp
           | Equi FProp FProp   -- Añadido
           deriving Show

type Interpretacion = [(Char, Bool)]

valor :: Interpretacion -> FProp -> Bool
valor _ (Const b)  = b
valor i (Var x)    = busca x i
valor i (Neg p)    = not (valor i p)
valor i (Conj p q) = valor i p && valor i q
valor i (Disj p q) = valor i p || valor i q    -- Añadido
valor i (Impl p q) = valor i p <= valor i q
valor i (Equi p q) = valor i p == valor i q    -- Añadido

busca :: Eq c => c -> [(c,v)] -> v
```

```haskell
busca c t = head [v | (c',v) <- t, c == c']

variables :: FProp -> [Char]
variables (Const _)  = []
variables (Var x)    = [x]
variables (Neg p)    = variables p
variables (Conj p q) = variables p ++ variables q
variables (Disj p q) = variables p ++ variables q   -- Añadido
variables (Impl p q) = variables p ++ variables q
variables (Equi p q) = variables p ++ variables q   -- Añadido

interpretacionesVar :: Int -> [[Bool]]
interpretacionesVar 0     = [[]]
interpretacionesVar (n+1) =
    map (False:) bss ++ map (True:) bss
    where bss = interpretacionesVar n

interpretaciones :: FProp -> [Interpretacion]
interpretaciones p =
    map (zip vs) (interpretacionesVar (length vs))
    where vs = nub (variables p)

esTautologia :: FProp -> Bool
esTautologia p =
    and [valor i p | i <- interpretaciones p]

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    interpretacionesVar' :: Int -> [[Bool]]
-- que sea equivalente a interpretacionesVar pero que en su definición
-- use listas de comprensión en lugar de map. Por ejemplo,
--    *Main> interpretacionesVar' 2
--    [[False,False],[False,True],[True,False],[True,True]]
-- ---------------------------------------------------------------------

interpretacionesVar' :: Int -> [[Bool]]
interpretacionesVar' 0     = [[]]
interpretacionesVar' (n+1) =
    [False:bs | bs <- bss] ++ [True:bs | bs <- bss]
    where bss = interpretacionesVar' n
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    interpretaciones' :: FProp -> [Interpretacion]
-- que sea equivalente a interpretaciones pero que en su definición
-- use listas de comprensión en lugar de map. Por ejemplo,
--    *Main> interpretaciones' (Impl (Var 'A') (Conj (Var 'A') (Var 'B')))
--    [[('A',False),('B',False)],
--     [('A',False),('B',True)],
--     [('A',True),('B',False)],
--     [('A',True),('B',True)]]
-- ---------------------------------------------------------------------

interpretaciones' :: FProp -> [Interpretacion]
interpretaciones' p =
    [zip vs i | i <- is]
    where vs = nub (variables p)
          is = interpretacionesVar (length vs)
```

Relación 18

Tipos de datos algebraicos: Modelización de juego de cartas

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En esta relación se estudia la modelización de un juego de cartas
-- como aplicación de los tipos de datos algebraicos. Además, se definen
-- los generadores correspondientes para comprobar las propiedades con
-- QuickCheck.
--
-- Estos ejercicios corresponden al tema 9 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-

-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares                                 --
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.Char
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio resuelto. Definir el tipo de datos Palo para representar
-- los cuatro palos de la baraja: picas, corazones, diamantes y
-- tréboles. Hacer que Palo sea instancia de Eq y Show.
```

```haskell
-- ---------------------------------------------------------------------

-- La definición es
data Palo = Picas | Corazones | Diamantes | Treboles
            deriving (Eq, Show)

-- ---------------------------------------------------------------------
-- Nota: Para que QuickCheck pueda generar elementos del tipo Palo se
-- usa la siguiente función.
-- ---------------------------------------------------------------------

instance Arbitrary Palo where
    arbitrary = elements [Picas, Corazones, Diamantes, Treboles]

-- ---------------------------------------------------------------------
-- Ejercicio resuelto. Definir el tipo de dato Color para representar los
-- colores de las cartas: rojo y negro. Hacer que Color sea instancia de
-- Show.
-- ---------------------------------------------------------------------

data Color = Rojo | Negro
             deriving Show

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    color :: Palo -> Color
-- tal que (color p) es el color del palo p. Por ejemplo,
--    color Corazones ==>  Rojo
-- Nota: Los corazones y los diamantes son rojos. Las picas y los
-- tréboles son negros.
-- ---------------------------------------------------------------------

color :: Palo -> Color
color Picas     = Negro
color Corazones = Rojo
color Diamantes = Rojo
color Treboles  = Negro

-- ---------------------------------------------------------------------
-- Ejercicio resuelto. Los valores de las cartas se dividen en los
```

```haskell
-- numéricos (del 2 al 10) y las figuras (sota, reina, rey y
-- as). Definir el tipo -- de datos Valor para representar los valores
-- de las cartas. Hacer que Valor sea instancia de Eq y Show.
--    Main> :type Sota
--    Sota :: Valor
--    Main> :type Reina
--    Reina :: Valor
--    Main> :type Rey
--    Rey :: Valor
--    Main> :type As
--    As :: Valor
--    Main> :type Numerico 3
--    Numerico 3 :: Valor
-- ---------------------------------------------------------------------

data Valor = Numerico Int | Sota | Reina | Rey | As
             deriving (Eq, Show)

-- ---------------------------------------------------------------------
-- Nota: Para que QuickCheck pueda generar elementos del tipo Valor se
-- usa la siguiente función.
-- ---------------------------------------------------------------------

instance Arbitrary Valor where
  arbitrary =
    oneof $
      [ do return c
      | c <- [Sota,Reina,Rey,As]
      ] ++
      [ do n <- choose (2,10)
           return (Numerico n)
      ]

-- ---------------------------------------------------------------------
-- Ejercicio 2. El orden de valor de las cartas (de mayor a menor) es
-- as, rey, reina, sota y las numéricas según su valor. Definir la
-- función
--    mayor :: Valor -> Valor -> Bool
-- tal que (mayor x y) se verifica si la carta x es de mayor valor que
-- la carta y. Por ejemplo,
```

```haskell
--    mayor Sota (Numerico 7)    ==>  True
--    mayor (Numerico 10) Reina  ==>  False
-- ---------------------------------------------------------------------

mayor :: Valor -> Valor -> Bool
mayor _            As          = False
mayor As           _           = True
mayor _            Rey         = False
mayor Rey          _           = True
mayor _            Reina       = False
mayor Reina        _           = True
mayor _            Sota        = False
mayor Sota         _           = True
mayor (Numerico m) (Numerico n) = m > n

-- ---------------------------------------------------------------------
-- Ejercicio 3. Comprobar con QuickCheck si dadas dos cartas, una
-- siempre tiene mayor valor que la otra. En caso de que no se verifique,
-- añadir la menor precondición para que lo haga.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_MayorValor1 a b =
    mayor a b || mayor b a

-- La comprobación es
--    Main> quickCheck prop_MayorValor1
--    Falsifiable, after 2 tests:
--    Sota
--    Sota
-- que indica que la propiedad es falsa porque la sota no tiene mayor
-- valor que la sota.

-- La precondición es que las cartas sean distintas:
prop_MayorValor a b =
    a /= b ==> mayor a b || mayor b a

-- La comprobación es
--    Main> quickCheck prop_MayorValor
--    OK, passed 100 tests.
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio resuelto. Definir el tipo de datos Carta para representar
-- las cartas mediante un valor y un palo. Hacer que Carta sea instancia
-- de Eq y Show. Por ejemplo,
--    Main> :type Carta Rey Corazones
--    Carta Rey Corazones :: Carta
--    Main> :type Carta (Numerico 4) Corazones
--    Carta (Numerico 4) Corazones :: Carta
-- ---------------------------------------------------------------------

data Carta = Carta Valor Palo
             deriving (Eq, Show)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    valor :: Carta -> Valor
-- tal que (valor c) es el valor de la carta c. Por ejemplo,
--    valor (Carta Rey Corazones)  ==>  Rey
-- ---------------------------------------------------------------------

valor :: Carta -> Valor
valor (Carta v p) = v

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    palo :: Carta -> Valor
-- tal que (palo c) es el palo de la carta c. Por ejemplo,
--    palo (Carta Rey Corazones)  ==>  Corazones
-- ---------------------------------------------------------------------

palo :: Carta -> Palo
palo (Carta v p) = p

-- ---------------------------------------------------------------------
-- Nota: Para que QuickCheck pueda generar elementos del tipo Carta se
-- usa la siguiente función.
-- ---------------------------------------------------------------------

instance Arbitrary Carta where
```

```
    arbitrary =
        do v <- arbitrary
           p <- arbitrary
           return (Carta v p)

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    ganaCarta :: Palo -> Carta -> Carta -> Bool
-- tal que (ganaCarta p c1 c2) se verifica si la carta c1 le gana a la
-- carta c2 cuando el palo de triunfo es p (es decir, las cartas son del
-- mismo palo y el valor de c1 es mayor que el de c2 o c1 es del palo de
-- triunfo). Por ejemplo,
--    ganaCarta Corazones (Carta Sota Picas) (Carta (Numerico 5) Picas)
--    ==> True
--    ganaCarta Corazones (Carta (Numerico 3) Picas) (Carta Sota Picas)
--    ==> False
--    ganaCarta Corazones (Carta (Numerico 3) Corazones) (Carta Sota Picas)
--    ==> True
--    ganaCarta Treboles (Carta (Numerico 3) Corazones) (Carta Sota Picas)
--    ==> False
-- ---------------------------------------------------------------------

ganaCarta :: Palo -> Carta -> Carta -> Bool
ganaCarta triunfo c c'
    | palo c == palo c' = mayor (valor c) (valor c')
    | palo c == triunfo = True
    | otherwise         = False

-- ---------------------------------------------------------------------
-- Ejercicio 7. Comprobar con QuickCheck si dadas dos cartas, una
-- siempre gana a la otra.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_GanaCarta t c1 c2 =
    ganaCarta t c1 c2 || ganaCarta t c2 c1

-- La comprobación es
--    Main> quickCheck prop_GanaCarta
--    Falsifiable, after 0 tests:
```

```
--    Diamantes
--    Carta Rey Corazones
--    Carta As Treboles
-- que indica que la propiedad no se verifica ya que cuando el triunfo
-- es diamantes, ni el rey de corazones le gana al as de tréboles ni el
-- as de tréboles le gana al rey de corazones.

-- ---------------------------------------------------------------------
-- Ejercicio resuelto. Definir el tipo de datos Mano para representar
-- una  mano en el juego de cartas. Una mano es vacía o se obtiene
-- agregando una carta a una mano. Hacer Mano instancia de Eq y
-- Show. Por ejemplo,
--    Main> :type Agrega (Carta Rey Corazones) Vacia
--    Agrega (Carta Rey Corazones) Vacia :: Mano
-- ---------------------------------------------------------------------

data Mano = Vacia | Agrega Carta Mano
            deriving (Eq, Show)

-- ---------------------------------------------------------------------
-- Nota: Para que QuickCheck pueda generar elementos del tipo Mano se
-- usa la siguiente función.
-- ---------------------------------------------------------------------

instance Arbitrary Mano where
    arbitrary =
        do cs <- arbitrary
           let mano []     = Vacia
               mano (c:cs) = Agrega c (mano cs)
           return (mano cs)

-- ---------------------------------------------------------------------
-- Ejercicio 8. Una mano gana a una carta c si alguna carta de la mano
-- le gana a c. Definir la función
--    ganaMano :: Palo -> Mano -> Carta -> Bool
-- tal que (gana t m c) se verifica si la mano m le gana a la carta c
-- cuando el triunfo es t. Por ejemplo,
--    ganaMano Picas (Agrega (Carta Sota Picas) Vacia) (Carta Rey Corazones)
--    ==>  True
--    ganaMano Picas (Agrega (Carta Sota Picas) Vacia) (Carta Rey Picas)
```

```
--    ==>  False
-- ---------------------------------------------------------------------

ganaMano :: Palo -> Mano -> Carta -> Bool
ganaMano triunfo Vacia        c' = False
ganaMano triunfo (Agrega c m) c' = ganaCarta triunfo c c' ||
                                   ganaMano triunfo m c'

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    eligeCarta :: Palo -> Carta -> Mano -> Carta
-- tal que (eligeCarta t c1 m) es la mejor carta de la mano m frente a
-- la carta c cuando el triunfo es t. La estrategia para elegir la mejor
-- carta es la siguiente:
-- * Si la mano sólo tiene una carta, se elige dicha carta.
-- * Si la primera carta de la mano es del palo de c1 y la mejor del
--   resto no es del palo de c1, se elige la primera de la mano,
-- * Si la primera carta de la mano no es del palo de c1 y la mejor
--   del resto es del palo de c1, se elige la mejor del resto.
-- * Si la primera carta de la mano le gana a c1 y la mejor del
--   resto no le gana a c1, se elige la primera de la mano,
-- * Si la mejor del resto le gana a c1 y la primera carta de la mano
--   no le gana a c1, se elige la mejor del resto.
-- * Si el valor de la primera carta es mayor que el de la mejor del
--   resto, se elige la mejor del resto.
-- * Si el valor de la primera carta no es mayor que el de la mejor
--   del resto, se elige la primera carta.
-- ---------------------------------------------------------------------

eligeCarta :: Palo -> Carta -> Mano -> Carta
eligeCarta triunfo c1 (Agrega c Vacia) = c                          -- 1
eligeCarta triunfo c1 (Agrega c resto)
  | palo c == palo c1 && palo c' /= palo c1                  = c    -- 2
  | palo c /= palo c1 && palo c' == palo c1                  = c'   -- 3
  | ganaCarta triunfo c  c1 && not (ganaCarta triunfo c' c1) = c    -- 4
  | ganaCarta triunfo c' c1 && not (ganaCarta triunfo c c1)  = c'   -- 5
  | mayor (valor c) (valor c')                               = c'   -- 6
  | otherwise                                                = c    -- 7
 where
  c' = eligeCarta triunfo c1 resto
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 10. Comprobar con QuickCheck que si una mano es ganadora,
-- entonces la carta elegida es ganadora.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_eligeCartaGanaSiEsPosible triunfo c m =
    m /= Vacia ==>
    ganaMano triunfo m c == ganaCarta triunfo (eligeCarta triunfo c m) c

-- La comprobación es
--    Main> quickCheck prop_eligeCartaGanaSiEsPosible
--    Falsifiable, after 12 tests:
--    Corazones
--    Carta Rey Treboles
--    Agrega (Carta (Numerico 6) Diamantes)
--           (Agrega (Carta Sota Picas)
--            (Agrega (Carta Rey Corazones)
--             (Agrega (Carta (Numerico 10) Treboles)
--              Vacia)))
-- La carta elegida es el 10 de tréboles (porque tiene que ser del mismo
-- palo), aunque el mano hay una carta (el rey de corazones) que gana.
```

Relación 19

Cálculo numérico

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En esta relación se definen funciones para resolver los siguientes
-- problemas de cálculo numérico:
-- * diferenciación numérica,
-- * cálculo de la raíz cuadrada mediante el método de Herón,
-- * cálculo de los ceros de una función por el método de Newton y
-- * cálculo de funciones inversas.

-- ---------------------------------------------------------------------
-- Importación de librerías                                            --
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Diferenciación numérica                                             --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    derivada :: Double -> (Double -> Double) -> Double -> Double
-- tal que (derivada a f x) es el valor de la derivada de la función f
-- en el punto x con aproximación a. Por ejemplo,
--    derivada 0.001 sin pi  ==  -0.9999998333332315
```

```
--    derivada 0.001 cos pi  ==  4.999999583255033e-4
-- ---------------------------------------------------------------------

derivada :: Double -> (Double -> Double) -> Double -> Double
derivada a f x = (f(x+a)-f(x))/a

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir las funciones
--    derivadaBurda :: (Double -> Double) -> Double -> Double
--    derivadaFina  :: (Double -> Double) -> Double -> Double
--    derivadaSuper :: (Double -> Double) -> Double -> Double
-- tales que
--    * (derivadaBurda f x) es el valor de la derivada de la función f
--      en el punto x con aproximación 0.01,
--    * (derivadaFina f x) es el valor de la derivada de la función f
--      en el punto x con aproximación 0.0001.
--    * (derivadauperBurda f x) es el valor de la derivada de la función f
--      en el punto x con aproximación 0.000001.
-- Por ejemplo,
--    derivadaBurda cos pi  ==  4.999958333473664e-3
--    derivadaFina  cos pi  ==  4.999999969612645e-5
--    derivadaSuper cos pi  ==  5.000444502911705e-7
-- ---------------------------------------------------------------------

derivadaBurda :: (Double -> Double) -> Double -> Double
derivadaBurda = derivada 0.01

derivadaFina :: (Double -> Double) -> Double -> Double
derivadaFina  = derivada 0.0001

derivadaSuper :: (Double -> Double) -> Double -> Double
derivadaSuper = derivada 0.000001

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    derivadaFinaDelSeno :: Double -> Double
-- tal que (derivadaFinaDelSeno x) es el valor de la derivada fina del
-- seno en x. Por ejemplo,
--    derivadaFinaDelSeno pi  ==  -0.9999999983354436
-- ---------------------------------------------------------------------
```

```haskell
derivadaFinaDelSeno :: Double -> Double
derivadaFinaDelSeno = derivadaFina sin

-- ---------------------------------------------------------------------
-- Cálculo de la raíz cuadrada                                         --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. En los siguientes apartados de este ejercicio se va a
-- calcular la raíz cuadrada de un número basándose en las siguientes
-- propiedades:
-- * Si y es una aproximación de la raíz cuadrada de x, entonces
--   (y+x/y)/2 es una aproximación mejor.
-- * El límite de la sucesión definida por
--       x_0     = 1
--       x_{n+1} = (x_n+x/x_n)/2
--   es la raíz cuadrada de x.
--
-- Definir, por iteración con until, la función
--    raiz :: Double -> Double
-- tal que (raiz x) es la raíz cuadrada de x calculada usando la
-- propiedad anterior con una aproximación de 0.00001 y tomando como
-- v. Por ejemplo,
--    raiz 9  ==  3.000000001396984
-- ---------------------------------------------------------------------

raiz :: Double -> Double
raiz x = raiz' 1
    where raiz' y | aceptable y = y
                  | otherwise   = raiz' (mejora y)
          mejora y    = 0.5*(y+x/y)
          aceptable y = abs(y*y-x) < 0.00001

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir el operador
--    (~=) :: Double -> Double -> Bool
-- tal que (x ~= y) si |x-y| < 0.001. Por ejemplo,
--    3.05 ~= 3.07        ==  False
--    3.00005 ~= 3.00007  == True
```

```haskell
-- ---------------------------------------------------------------------

infix 5 ~=
(~=) :: Double -> Double -> Bool
x ~= y = abs(x-y) < 0.001

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Comprobar con QuickCheck que si x es positivo,
-- entonces
--    (raiz x)^2 ~= x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_raiz :: Double -> Bool
prop_raiz x =
    (raiz x')^2 ~= x'
    where x' = abs x

-- La comprobación es
--    *Main> quickCheck prop_raiz
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.4. Definir por recursión la función
--    until' :: (a -> Bool) -> (a -> a) -> a -> a
-- tal que (until' p f x) es el resultado de aplicar la función f a x el
-- menor número posible de veces, hasta alcanzar un valor que satisface
-- el predicado p. Por ejemplo,
--    until' (>1000) (2*) 1  ==  1024
-- Nota: until' es equivalente a la predefinida until.
-- ---------------------------------------------------------------------

until' :: (a -> Bool) -> (a -> a) -> a -> a
until' p f x  | p x       = x
              | otherwise = until' p f (f x)

-- ---------------------------------------------------------------------
-- Ejercicio 3.5. Definir, por iteración con until, la función
--    raizI :: Double -> Double
-- tal que (raizI x) es la raíz cuadrada de x calculada usando la
```

```haskell
-- propiedad anterior. Por ejemplo,
--    raizI 9  ==  3.000000001396984
-- ---------------------------------------------------------------------

raizI :: Double -> Double
raizI x = until aceptable mejora 1
          where mejora y    = 0.5*(y+x/y)
                aceptable y = abs(y*y-x) < 0.00001

-- ---------------------------------------------------------------------
-- Ejercicio 3.6. Comprobar con QuickCheck que si x es positivo,
-- entonces
--    (raizI x)^2 ~= x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_raizI :: Double -> Bool
prop_raizI x =
    (raizI x')^2 ~= x'
    where x' = abs x

-- La comprobación es
--    *Main> quickCheck prop_raizI
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ceros de una función                                                --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 4. Los ceros de una función pueden calcularse mediante el
-- método de Newton basándose en las siguientes propiedades:
-- * Si b es una aproximación para el punto cero de f, entonces
--   b-f(b)/f'(b) es una mejor aproximación.
-- * el límite de la sucesión x_n definida por
--      x_0     = 1
--      x_{n+1} = x_n-f(x_n)/f'(x_n)
--   es un cero de f.
-- ---------------------------------------------------------------------
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir por recursión la función
--    puntoCero :: (Double -> Double) -> Double
-- tal que (puntoCero f) es un cero de la función f calculado usando la
-- propiedad anterior. Por ejemplo,
--    puntoCero cos  ==  1.5707963267949576
-- ---------------------------------------------------------------------

puntoCero :: (Double -> Double) -> Double
puntoCero f = puntoCero' f 1
    where puntoCero' f x | aceptable x = x
                         | otherwise   = puntoCero' f (mejora x)
          mejora b    = b - f b / derivadaFina f b
          aceptable b = abs (f b) < 0.00001

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir, por iteración con until, la función
--    puntoCeroI :: (Double -> Double) -> Double
-- tal que (puntoCeroI f) es un cero de la función f calculado usando la
-- propiedad anterior. Por ejemplo,
--    puntoCeroI cos  ==  1.5707963267949576
-- ---------------------------------------------------------------------

puntoCeroI :: (Double -> Double) -> Double
puntoCeroI f = until aceptable mejora 1
    where mejora b    = b - f b / derivadaFina f b
          aceptable b = abs (f b) < 0.00001

-- ---------------------------------------------------------------------
-- Funciones inversas                                                  --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 5. En este ejercicio se usará la función puntoCero para
-- definir la inversa de distintas funciones.
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Definir, usando puntoCero, la función
--    raizCuadrada :: Double -> Double
```

```haskell
-- tal que (raizCuadrada x) es la raíz cuadrada de x. Por ejemplo,
--    raizCuadrada 9  ==  3.000000002941184
-- ---------------------------------------------------------------------

raizCuadrada :: Double -> Double
raizCuadrada a = puntoCero f
    where f x = x*x-a

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Comprobar con QuickCheck que si x es positivo,
-- entonces
--    (raizCuadrada x)^2 ~= x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_raizCuadrada :: Double -> Bool
prop_raizCuadrada x =
    (raizCuadrada x')^2 ~= x'
    where x' = abs x

-- La comprobación es
--    *Main> quickCheck prop_raizCuadrada
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Definir, usando puntoCero, la función
--    raizCubica :: Double -> Double
-- tal que (raizCubica x) es la raíz cuadrada de x. Por ejemplo,
--    raizCubica 27  ==  3.0000000000196048
-- ---------------------------------------------------------------------

raizCubica :: Double -> Double
raizCubica a = puntoCero f
    where f x = x*x*x-a

-- ---------------------------------------------------------------------
-- Ejercicio 5.4. Comprobar con QuickCheck que si x es positivo,
-- entonces
--    (raizCubica x)^3 ~= x
-- ---------------------------------------------------------------------
```

```haskell
-- La propiedad es
prop_raizCubica :: Double -> Bool
prop_raizCubica x =
    (raizCubica x)^3 ~= x
    where x' = abs x

-- La comprobación es
--    *Main> quickCheck prop_raizCubica
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.5. Definir, usando puntoCero, la función
--    arcoseno :: Double -> Double
-- tal que (arcoseno x) es el arcoseno de x. Por ejemplo,
--    arcoseno 1  == 1.5665489428306574
-- ---------------------------------------------------------------------

arcoseno :: Double -> Double
arcoseno a = puntoCero f
    where f x = sin x - a

-- ---------------------------------------------------------------------
-- Ejercicio 5.6. Comprobar con QuickCheck que si x está entre 0 y 1,
-- entonces
--    sin (arcoseno x) ~= x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_arcoseno :: Double -> Bool
prop_arcoseno x =
    sin (arcoseno x') ~= x'
    where x' = abs (x - fromIntegral (truncate x))

-- La comprobación es
--    *Main> quickCheck prop_arcoseno
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.7. Definir, usando puntoCero, la función
```

```haskell
--    arcocoseno :: Double -> Double
-- tal que (arcoseno x) es el arcoseno de x. Por ejemplo,
--    arcocoseno 0  == 1.5707963267949576
-- ---------------------------------------------------------------------

arcocoseno :: Double -> Double
arcocoseno a = puntoCero f
    where f x = cos x - a

-- ---------------------------------------------------------------------
-- Ejercicio 5.8. Comprobar con QuickCheck que si x está entre 0 y 1,
-- entonces
--    cos (arcocoseno x) ~= x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_arcocoseno :: Double -> Bool
prop_arcocoseno x =
    cos (arcocoseno x') ~= x'
    where x' = abs (x - fromIntegral (truncate x))

-- La comprobación es
--    *Main> quickCheck prop_arcocoseno
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.7. Definir, usando puntoCero, la función
--    inversa :: (Double -> Double) -> Double -> Double
-- tal que (inversa g x) es el valor de la inversa de g en x. Por
-- ejemplo,
--    inversa (^2) 9  ==  3.000000002941184
-- ---------------------------------------------------------------------

inversa :: (Double -> Double) -> Double -> Double
inversa g a = puntoCero f
    where f x = g x - a

-- ---------------------------------------------------------------------
-- Ejercicio 5.8. Redefinir, usando inversa, las funciones raizCuadrada,
-- raizCubica, arcoseno y arcocoseno.
```

```haskell
-- ---------------------------------------------------------------------

raizCuadrada' = inversa (^2)
raizCubica'   = inversa (^3)
arcoseno'     = inversa sin
arcocoseno'   = inversa cos
```

Relación 20

Demostración de propiedades por inducción

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En esta relación se plantean ejercicios de demostración por inducción
-- de propiedades de programas. En concreto,
--   * la suma de los n primeros impares es n^2,
--   * 1 + 2^0 + 2^1 + 2^2 + ... + 2^n = 2^(n+1),
--   * todos los elementos de (copia n x) son iguales a x,
-- Además, se plantea la definición de la traspuesta de una matriz.
--
-- Estos ejercicios corresponden al tema 8 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-8.pdf

-- ---------------------------------------------------------------------
-- Importación de librerías                                            --
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1a. Definir por recursión la función
--    sumaImpares :: Int -> Int
-- tal que (sumaImpares n) es la suma de los n primeros números
```

```haskell
-- impares. Por ejemplo,
--    sumaImpares 5  ==  25
-- ---------------------------------------------------------------------

sumaImpares :: Int -> Int
sumaImpares 0     = 0
sumaImpares (n+1) = sumaImpares n + (2*n+1)

-- ---------------------------------------------------------------------
-- Ejercicio 1b. Definir, sin usar recursión, la función
--    sumaImpares' :: Int -> Int
-- tal que (sumaImpares' n) es la suma de los n primeros números
-- impares. Por ejemplo,
--    *Main> sumaImpares' 5  ==  25
-- ---------------------------------------------------------------------

sumaImpares' :: Int -> Int
sumaImpares' n = sum [1,3..(2*n-1)]

-- ---------------------------------------------------------------------
-- Ejercicio 1c. Definir la función
--    sumaImparesIguales :: Int -> Int -> Bool
-- tal que (sumaImparesIguales m n) se verifica si para todo x entre m y
-- n se tiene que (sumaImpares x) y (sumaImpares' x) son iguales.
--
-- Comprobar que (sumaImpares x) y (sumaImpares' x) son iguales para
-- todos los números x entre 1 y 100.
-- ---------------------------------------------------------------------

-- La definición es
sumaImparesIguales :: Int -> Int -> Bool
sumaImparesIguales m n =
    and [sumaImpares x == sumaImpares' x | x <- [m..n]]

-- La comprobación es
--    *Main>  sumaImparesIguales 1 100
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 1d. Definir la función
```

```
--    grafoSumaImpares :: Int -> Int -> [(Int,Int)]
-- tal que (grafoSumaImpares m n) es la lista formadas por los números x
-- entre m y n y los valores de (sumaImpares x).
--
-- Calcular (grafoSumaImpares 1 9).
-- ------------------------------------------------------------------

-- La definición es
grafoSumaImpares :: Int -> Int -> [(Int,Int)]
grafoSumaImpares m n =
    [(x,sumaImpares x) | x <- [m..n]]

-- El cálculo es
--    *Main> grafoSumaImpares 1 9
--    [(1,1),(2,4),(3,9),(4,16),(5,25),(6,36),(7,49),(8,64),(9,81)]

-- ------------------------------------------------------------------
-- Ejercicio 1e. Demostrar por inducción que para todo n,
-- (sumaImpares n) es igual a n^2.
-- ------------------------------------------------------------------

{-
 Caso base: Hay que demostrar que
    sumaImpares 0 = 0^2
 En efecto,
    sumaImpares 0   [por hipótesis]
    = 0               [por sumaImpares.1]
    = 0^2             [por aritmética]

  Caso inductivo: Se supone la hipótesis de inducción (H.I.)
     sumaImpares n = n^2
  Hay que demostrar que
     sumaImpares (n+1) = (n+1)^2
  En efecto,
     sumaImpares (n+1) =
     = (sumaImpares n) + (2*n+1    )    [por sumaImpares.2]
     = n^2 + (2*n+1)                   [por H.I.]
     = (n+1)^2                         [por álgebra]
-}
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2a. Definir por recursión la función
--    sumaPotenciasDeDosMasUno :: Int -> Int
-- tal que
--    (sumaPotenciasDeDosMasUno n) = 1 + 2^0 + 2^1 + 2^2 + ... + 2^n.
-- Por ejemplo,
--    sumaPotenciasDeDosMasUno 3  ==  16
-- ---------------------------------------------------------------------

sumaPotenciasDeDosMasUno :: Int -> Int
sumaPotenciasDeDosMasUno 0     = 2
sumaPotenciasDeDosMasUno (n+1) = sumaPotenciasDeDosMasUno n + 2^(n+1)

-- ---------------------------------------------------------------------
-- Ejercicio 2b. Definir por comprensión la función
--    sumaPotenciasDeDosMasUno' :: Int -> Int
-- tal que
--    (sumaPotenciasDeDosMasUno' n) = 1 + 2^0 + 2^1 + 2^2 + ... + 2^n.
-- Por ejemplo,
--    sumaPotenciasDeDosMasUno' 3  ==  16
-- ---------------------------------------------------------------------

sumaPotenciasDeDosMasUno' :: Int -> Int
sumaPotenciasDeDosMasUno' n = 1 + sum [2^x | x <- [0..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 2c. Demostrar por inducción que
--    sumaPotenciasDeDosMasUno n = 2^(n+1)
-- ---------------------------------------------------------------------

{-
  Caso base: Hay que demostrar que
     sumaPotenciasDeDosMasUno 0 = 2^(0+1)
  En efecto,
       sumaPotenciasDeDosMasUno 0
     = 2                              [por sumaPotenciasDeDosMasUno.1]
     = 2^(0+1)                        [por aritmética]

  Caso inductivo: Se supone la hipótesis de inducción (H.I.)
     sumaPotenciasDeDosMasUno n = 2^(n+1)
```

```
  Hay que demostrar que
     sumaPotenciasDeDosMasUno (n+1) = 2^((n+1)+1)
  En efecto,
       sumaPotenciasDeDosMasUno (n+1)
     = (sumaPotenciasDeDosMasUno n) + 2^(n+1)  [por sumaPotenciasDeDosMasUno.2]
     = 2^(n+1) + 2^(n+1)                       [por H.I.]
     = 2^((n+1)+1)                             [por aritmética]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 3a. Definir por recursión la función
--    copia :: Int -> a -> [a]
-- tal que (copia n x) es la lista formado por n copias del elemento
-- x. Por ejemplo,
--    copia 3 2  ==  [2,2,2]
-- ---------------------------------------------------------------------

copia :: Int -> a -> [a]
copia 0 _     = []              -- copia.1
copia (n+1) x = x : copia n x   -- copia.2

-- ---------------------------------------------------------------------
-- Ejercicio 3b. Definir por recursión la función
--    todos :: (a -> Bool) -> [a] -> Bool
-- tal que (todos p xs) se verifica si todos los elementos de xs cumplen
-- la propiedad p. Por ejemplo,
--    todos even [2,6,4]  ==  True
--    todos even [2,5,4]  ==  False
-- ---------------------------------------------------------------------

todos :: (a -> Bool) -> [a] -> Bool
todos p []       = True                -- todos.1
todos p (x : xs) = p x && todos p xs   -- todos.2

-- ---------------------------------------------------------------------
-- Ejercicio 3c. Comprobar con QuickCheck que todos los elementos de
-- (copia n x) son iguales a x.
-- ---------------------------------------------------------------------

-- La propiedad es
```

```haskell
prop_copia :: Eq a => Int -> a -> Bool
prop_copia n x =
    todos (==x) (copia n' x)
    where n' = abs n

-- La comprobación es
--    *Main> quickCheck prop_copia
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3d. Demostrar, por inducción en n, que todos los elementos
-- de (copia n x) son iguales a x.
-- ---------------------------------------------------------------------

{-
  Hay que demostrar que para todo n y todo x,
     todos (==x) (copia n x)

  Caso base: Hay que demostrar que
     todos (==x) (copia 0 x) = True
  En efecto,
       todos (== x) (copia 0 x)
     = todos (== x) []            [por copia.1]
     = True                       [por todos.1]

  Caso inductivo: Se supone la hipótesis de inducción (H.I.)
     todos (==x) (copia n x) = True
  Hay que demostrar que
     todos (==x) (copia (n+1) x) = True
  En efecto,
       todos (==x) (copia (n+1) x)
     = todos (==x) (x : copia n x )         [por copia.2]
     = x == x && todos (==x) (copia n x )   [por todos.2]
     = True && todos (==x) (copia n x )     [por def. de ==]
     = todos (==x) (copia n x )             [por def. de &&]
     = True                                 [por H.I.]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 3e. Definir por plegado la función
```

```
--    todos' :: (a -> Bool) -> [a] -> Bool
-- tal que (todos' p xs) se verifica si todos los elementos de xs cumplen
-- la propiedad p. Por ejemplo,
--    todos' even [2,6,4]  ==>  True
--    todos' even [2,5,4]  ==>  False
-- ---------------------------------------------------------------------

todos' :: (a -> Bool) -> [a] -> Bool
todos' p = foldr ((&&) . p) True

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    traspuesta :: [[a]] -> [[a]]
-- tal que (traspuesta m) es la traspuesta de la matriz m. Por ejemplo,
--    traspuesta [[1,2,3],[4,5,6]]    ==  [[1,4],[2,5],[3,6]]
--    traspuesta [[1,4],[2,5],[3,6]]  ==  [[1,2,3],[4,5,6]]
-- ---------------------------------------------------------------------

traspuesta :: [[a]] -> [[a]]
traspuesta []           = []
traspuesta ([]:xss)     = traspuesta xss
traspuesta ((x:xs):xss) =
    (x:[h | (h:_) <- xss]) : traspuesta (xs : [t | (_:t) <- xss])
```

Relación 21

Demostración de propiedades por inducción. Mayorías parlamentarias

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- Esta relación tiene dos partes. En la 1ª parte se plantean ejercicios
-- de demostración por inducción de propiedades de programas. En concreto,
--   * la equivalencia de las definiciones de factorial con y sin
--     acumulador,
--   * amplia xs y = xs ++ [y].
-- En la 2º parte se presenta un nuevo caso de estudio de los tipos
-- de datos algebraicos para estudiar las mayorías parlamentarias.
--
-- Estos ejercicios corresponden a los temas 8 y 9 cuyas transparencias
-- se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-8.pdf
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-9.pdf

-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares                                 --
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List (delete)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir por recursión la función
--   factR :: Integer -> Integer
-- tal que (factR n) es el factorial de n. Por ejemplo,
--   factR 4  ==  24
-- ---------------------------------------------------------------------

factR :: Integer -> Integer
factR 0 = 1
factR (n+1) = (n+1) * factR n

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir por comprensión la función
--   factC :: Integer -> Integer
-- tal que (factR n) es el factorial de n. Por ejemplo,
--   factC 4  ==  24
-- ---------------------------------------------------------------------

factC :: Integer -> Integer
factC n = product [1..n]

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comprobar con QuickCheck que las funciones factR y
-- factC son equivalentes sobre los números naturales.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_factR_factC :: Integer -> Bool
prop_factR_factC n =
    factR n' == factC n'
    where n' = abs n

-- La comprobación es
--    *Main> quickCheck prop_factR_factC
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Comprobar con QuickCheck si las funciones factR y
-- factC son equivalentes sobre los números enteros.
```

```haskell
-- ---------------------------------------------------------------------

-- La propiedad es
prop_factR_factC_Int :: Integer -> Bool
prop_factR_factC_Int n =
    factR n == factC n

-- La comprobación es
--    *Main> quickCheck prop_factR_factC_Int
--    *** Exception: Non-exhaustive patterns in function factR

-- No son iguales ya que factR no está definida para los números
-- negativos y factC de cualquier número negativo es 0.

-- ---------------------------------------------------------------------
-- Ejercicio 1.5. Se considera la siguiente definición iterativa de la
-- función factorial
--    factI :: Integer -> Integer
--    factI n = factI' n 1
--
--    factI' :: Integer -> Integer -> Integer
--    factI' 0     x = x                  -- factI'.1
--    factI' (n+1) x = factI' n (n+1)*x   -- factI'.2
-- Comprobar con QuickCheck que factI y factR son equivalentes sobre los
-- números naturales.
-- ---------------------------------------------------------------------

factI :: Integer -> Integer
factI n = factI' n 1

factI' :: Integer -> Integer -> Integer
factI' 0     x = x
factI' (n+1) x = factI' n (n+1)*x

-- La propiedad es
prop_factI_factR n =
    factI n' == factR n'
    where n' = abs n

-- La comprobación es
```

```
--    *Main> quickCheck prop_factI_factR
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 1.6. Comprobar con QuickCheck que para todo número natural
-- n, (factI' n x) es igual al producto de x y (factR n).
-- ---------------------------------------------------------------------

-- La propiedad es
prop_factI' :: Integer -> Integer -> Bool
prop_factI' n  x =
    factI' n' x == x * factR n'
    where n' = abs n

-- La comprobación es
--    *Main> quickCheck prop_factI'
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 1.7. Demostrar por inducción que para todo número natural
-- n, (factI' n x) es igual x*n!
-- ---------------------------------------------------------------------

{-
  Demostración (por inducción en n)

  Caso base: Hay que demostrar que factI' 0 x = x*0!
  En efecto,
     factI' 0 x
     = x           [por factI'.1]
     = x*0!        [por álgebra]

  Caso inductivo: Se supone la hipótesis de inducción: para todo x,
     factI' n x = x*n!
  hay que demostrar que para todo x
     factI' (n+1) x = x*(n+1)!
  En efecto,
     factI' (n+1) x
     = factI' n (n+1)*x    [por factI'.2]
     = (n+1)*x*n!          [por hipótesis de inducción]
```

```
     = x*(n+1)!            [por álgebra]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir, recursivamente y sin usar (++). la función
--    amplia :: [a] -> a -> [a]
-- tal que (amplia xs y) es la lista obtenida añadiendo el elemento y al
-- final de la lista xs. Por ejemplo,
--    amplia [2,5] 3  ==  [2,5,3]
-- ---------------------------------------------------------------------

amplia :: [a] -> a -> [a]
amplia []     y = [y]               -- amplia.1
amplia (x:xs) y = x : amplia xs y   -- amplia.2

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, mediante plegado. la función
--    ampliaF :: [a] -> a -> [a]
-- tal que (ampliaF xs y) es la lista obtenida añadiendo el elemento y al
-- final de la lista xs. Por ejemplo,
--    ampliaF [2,5] 3  ==  [2,5,3]
-- ---------------------------------------------------------------------

ampliaF :: [a] -> a -> [a]
ampliaF xs y = foldr (:) [y] xs

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Comprobar con QuickCheck que amplia y ampliaF son
-- equivalentes.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_amplia_ampliaF :: Eq a => [a] -> a -> Bool
prop_amplia_ampliaF xs y =
    amplia xs y == ampliaF xs y

-- La comprobación es
--    *Main> quickCheck prop_amplia_ampliaF
--    OK, passed 100 tests.
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2.4. Comprobar con QuickCheck que
--    amplia xs y = xs ++ [y]
-- ---------------------------------------------------------------------

-- La propiedad es
prop_amplia :: Eq a => [a] -> a -> Bool
prop_amplia xs y =
    amplia xs y == xs ++ [y]

-- La comprobación es
--    *Main> quickCheck prop_amplia
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2.5. Demostrar por inducción que
--    amplia xs y = xs ++ [y]
-- ---------------------------------------------------------------------

{-
  Demostración: Por inducción en xs.

  Caso base: Hay que demostrar que
     amplia [] y = [] ++ [y]
  En efecto,
     amplia [] y
     = [y]          [por amplia.1]
     = [] ++ [y]    [por (++).1]

  Caso inductivo: Se supone la hipótesis de inducción
     amplia xs y = xs ++ [y]
  Hay que demostrar que
     amplia (x:xs) y = (x:xs) ++ [y]
  En efecto,
     amplia (x:xs) y
     = x : amplia xs y     [por amplia.2]
     = x : (xs ++ [y])     [por hipótesis de inducción]
     = (x:xs) ++ [y]       [por (++).2]
-}
```

```haskell
-- ---------------------------------------------------------------------
-- Mayorías parlamentarias                                             --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir el tipo de datos Partido para representar los
-- partidos de un Parlamento. Los partidos son P1, P2,..., P8. La clase
-- Partido está contenida en Eq, Ord y Show.
-- ---------------------------------------------------------------------

data Partido
    = P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8
    deriving ( Eq, Ord, Show )

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir el tipo Parlamentarios para representar el
-- número de parlamentarios que posee un partido en el parlamento.
-- ---------------------------------------------------------------------

type Parlamentarios = Integer

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir el tipo (Tabla a b) para representar una lista
-- de pares de elementos el primero de tipo a y el segundo de tipo
-- b. Definir Asamblea para representar una tabla de partidos y
-- parlamentarios.
-- ---------------------------------------------------------------------

type Tabla a b = [(a,b)]
type Asamblea  = Tabla Partido Parlamentarios

-- ---------------------------------------------------------------------
-- Ejercicio 3.4. Definir la función
--    partidos :: Asamblea -> [Partido]
-- tal que (partidos a) es la lista de partidos en la asamblea a. Por
-- ejemplo,
--    partidos [(P1,3),(P3,5),(P4,3)]  ==>  [P1,P3,P4]
-- ---------------------------------------------------------------------

partidos :: Asamblea -> [Partido]
```

```haskell
partidos a = [p | (p,_) <- a]

-- ---------------------------------------------------------------------
-- Ejercicio 3.5. Definir la función
--    parlamentarios :: Asamblea -> Integer
-- tal que (parlamentarios a) es el número de parlamentarios en la
-- asamblea a. Por ejemplo,
--    parlamentarios [(P1,3),(P3,5),(P4,3)]  ==>  11
-- ---------------------------------------------------------------------

parlamentarios :: Asamblea -> Integer
parlamentarios a = sum [e | (_,e) <- a]

-- ---------------------------------------------------------------------
-- Ejercicio 3.6. Definir la función
--    busca :: Eq a => a -> Tabla a b -> b
-- tal que (busca x t) es el valor correspondiente a x en la tabla
-- t. Por ejemplo,
--    Main> busca P3 [(P1,2),(P3,19)]
--    19
--    Main> busca P8 [(P1,2),(P3,19)]
--    Program error: no tiene valor en la tabla
-- ---------------------------------------------------------------------

busca :: Eq a => a -> Tabla a b -> b
busca x t | null xs   = error "no tiene valor en la tabla"
          | otherwise = head xs
  where xs = [b | (a,b) <- t, a == x]


buscaR :: Eq a => a -> Tabla a b -> b
buscaR x []           = error "no tiene valor en la tabla"
buscaR x ((x',y):xys)
    | x == x'         = y
    | otherwise       = buscaR x xys

-- ---------------------------------------------------------------------
-- Ejercicio 3.7. Definir la función
--    busca' :: Eq a => a -> Table a b -> Maybe b
-- tal que (busca' x t) es justo el valor correspondiente a x en la
```

```haskell
-- tabla t, o Nothing si x no tiene valor. Por ejemplo,
--    busca' P3 [(P1,2),(P3,19)]   ==   Just 19
--    busca' P8 [(P1,2),(P3,19)]   ==   Nothing
-- ---------------------------------------------------------------------

busca' :: Eq a => a -> Tabla a b -> Maybe b
busca' x t | null xs   = Nothing
           | otherwise = Just (head xs)
   where xs = [b | (a,b) <- t, a == x]

-- Definición por recursión
busca'R :: Eq a => a -> Tabla a b -> Maybe b
busca'R x []           = Nothing
busca'R x ((x',y):xys)
    | x == x'          = Just y
    | otherwise        = busca'R x xys

-- ---------------------------------------------------------------------
-- Ejercicio 3.8. Comprobar con QuickCheck que si (busca' x t) es
-- Nothing, entonces x es distinto de todos los elementos de t.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_BuscaNothing :: Integer -> [(Integer,Integer)] -> Property
prop_BuscaNothing x t =
    busca' x t == Nothing ==>
    x `notElem` [ a | (a,_) <- t ]

-- La comprobación es
--    Main> quickCheck prop_BuscaNothing
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.9. Comprobar que la función busca' es equivalente a la
-- función lookup del Prelude.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_BuscaEquivLookup :: Integer -> [(Integer,Integer)] -> Bool
prop_BuscaEquivLookup x t =
```

```
    busca' x t == lookup x t

-- La comprobación es
--    Main> quickCheck prop_BuscaEquivLookup
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.10. Definir el tipo Coalicion como una lista de partidos.
-- ---------------------------------------------------------------------

type Coalicion = [Partido]

-- ---------------------------------------------------------------------
-- Ejercicio 3.11. Definir la función
--    mayoria :: Asamblea -> Integer
-- tal que (mayoria xs) es el número de parlamentarios que se necesitan
-- para tener la mayoría en la asamblea xs. Por ejemplo,
--    mayoria [(P1,3),(P3,5),(P4,3)]   ==   6
--    mayoria [(P1,3),(P3,6)]          ==   5
-- ---------------------------------------------------------------------

mayoria :: Asamblea -> Integer
mayoria xs = parlamentarios xs `div` 2 + 1

-- ---------------------------------------------------------------------
-- Ejercicio 3.12. Definir la función
--    coaliciones :: Asamblea -> Integer -> [Coalicion]
-- tal que (coaliciones xs n) es la lista de coaliciones necesarias para
-- alcanzar n parlamentarios. Por ejemplo,
--    coaliciones [(P1,3),(P3,5),(P4,3)] 6   ==  [[P3,P4],[P1,P4],[P1,P3]]
--    coaliciones [(P1,3),(P3,5),(P4,3)] 9   ==  [[P1,P3,P4]]
--    coaliciones [(P1,3),(P3,5),(P4,3)] 14  ==  []
--    coaliciones [(P1,3),(P3,5),(P4,3)] 2   ==  [[P4],[P3],[P1]]
-- ---------------------------------------------------------------------

coaliciones :: Asamblea -> Integer -> [Coalicion]
coaliciones _ n | n <= 0  = [[]]
coaliciones [] n          = []
coaliciones ((p,m):xs) n  =
    coaliciones xs n ++ [p:c | c <- coaliciones xs (n-m)]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 3.13. Definir la función
--    mayorias :: Asamblea -> [Coalicion]
-- tal que (mayorias a) es la lista de coaliciones mayoritarias en la
-- asamblea a. Por ejemplo,
--    mayorias [(P1,3),(P3,5),(P4,3)]   ==   [[P3,P4],[P1,P4],[P1,P3]]
--    mayorias [(P1,2),(P3,5),(P4,3)]   ==   [[P3],[P1,P4],[P1,P3]]
-- ---------------------------------------------------------------------

mayorias :: Asamblea -> [Coalicion]
mayorias asamblea =
    coaliciones asamblea (mayoria asamblea)

-- ---------------------------------------------------------------------
-- Ejercicio 3.14. Definir el tipo de datos Asamblea.
-- ---------------------------------------------------------------------

data Asamblea2 = A Asamblea
               deriving Show

-- ---------------------------------------------------------------------
-- Ejercicio 3.15. Definir la propiedad
--    esMayoritaria :: Coalicion -> Asamblea -> Bool
-- tal que (esMayoritaria c a) se verifica si la coalición c es
-- mayoritaria en la asamblea a. Por ejemplo,
--    esMayoritaria [P3,P4] [(P1,3),(P3,5),(P4,3)]   ==   True
--    esMayoritaria [P4] [(P1,3),(P3,5),(P4,3)]      ==   False
-- ---------------------------------------------------------------------

esMayoritaria :: Coalicion -> Asamblea -> Bool
esMayoritaria c a =
    sum [busca p a| p <- c] >= mayoria a

-- ---------------------------------------------------------------------
-- Ejercicio 3.16. Comprobar con QuickCheck que las coaliciones
-- obtenidas por (mayorias asamblea) son coaliciones mayoritarias en la
-- asamblea.
-- ---------------------------------------------------------------------
```

```haskell
-- La propiedad es
prop_MayoriasSonMayoritarias :: Asamblea2 -> Bool
prop_MayoriasSonMayoritarias (A asamblea) =
  and [esMayoritaria c asamblea | c <- mayorias asamblea]

-- La comprobación es
--    Main> quickCheck prop_MayoriasSonMayoritarias
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.17. Definir la función
--    esMayoritariaMinimal :: Coalicion -> Asamblea -> Bool
-- tal que (esMayoritariaMinimal c a) se verifica si la coalición c es
-- mayoritaria en la asamblea a, pero si se quita a c cualquiera de sus
-- partidos la coalición resultante no es mayoritaria. Por ejemplo,
--    esMayoritariaMinimal [P3,P4] [(P1,3),(P3,5),(P4,3)]     ==  True
--    esMayoritariaMinimal [P1,P3,P4] [(P1,3),(P3,5),(P4,3)]  ==  False
-- ---------------------------------------------------------------------

esMayoritariaMinimal :: Coalicion -> Asamblea -> Bool
esMayoritariaMinimal c a =
    esMayoritaria c a &&
    and [not(esMayoritaria (delete p c) a) | p <-c]

-- ---------------------------------------------------------------------
-- Ejercicio 3.18. Comprobar con QuickCheck si las coaliciones obtenidas
-- por (mayorias asamblea) son coaliciones mayoritarias minimales en la
-- asamblea.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_MayoriasSonMayoritariasMinimales :: Asamblea2 -> Bool
prop_MayoriasSonMayoritariasMinimales (A asamblea) =
  and [esMayoritariaMinimal c asamblea | c <- mayorias asamblea]

-- La comprobación es
--    Main> quickCheck prop_MayoriasSonMayoritariasMinimales
--    Falsifiable, after 0 tests:
--    A [(P1,1),(P2,0),(P3,1),(P4,1),(P5,0),(P6,1),(P7,0),(P8,1)]
```

```
-- Por tanto, no se cumple la propiedad. Para buscar una coalición no
-- minimal generada por mayorias, definimos la función
contraejemplo a =
    head [c | c <- mayorias a, not(esMayoritariaMinimal c a)]

-- el cálculo del contraejemplo es
-- Main> contraejemplo [(P1,1),(P2,0),(P3,1),(P4,1),(P5,0),(P6,1),(P7,0),(P8,1)]
-- [P4,P6,P7,P8]

-- La coalición [P4,P6,P7,P8] no es minimal ya que [P4,P6,P8] también es
-- mayoritaria. En efecto,
--    Main> esMayoritaria [P4,P6,P8]
--                        [(P1,1),(P2,0),(P3,1),(P4,1),
--                         (P5,0),(P6,1),(P7,0),(P8,1)]
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 3.19. Definir la función
--    coalicionesMinimales :: Asamblea -> Integer -> [Coalicion,Parlamentarios]
-- tal que (coalicionesMinimales xs n) es la lista de coaliciones
-- minimales necesarias para alcanzar n parlamentarios. Por ejemplo,
--    Main> coalicionesMinimales [(P1,3),(P3,5),(P4,3)] 6
--    [([P3,P4],8),([P1,P4],6),([P1,P3],8)]
--    Main> coalicionesMinimales [(P1,3),(P3,5),(P4,3)] 5
--    [([P3],5),([P1,P4],6)]
-- ---------------------------------------------------------------------

coalicionesMinimales :: Asamblea -> Integer -> [(Coalicion,Parlamentarios)]
coalicionesMinimales _ n | n <= 0  = [([],0)]
coalicionesMinimales [] n          = []
coalicionesMinimales ((p,m):xs) n  =
    coalicionesMinimales xs n ++
    [(p:ys, t+m) | (ys,t) <- coalicionesMinimales xs (n-m), t<n]

-- ---------------------------------------------------------------------
-- Ejercicio 3.20. Definir la función
--    mayoriasMinimales :: Asamblea -> [Coalicion]
-- tal que (mayoriasMinimales a) es la lista de coaliciones mayoritarias
-- minimales en la asamblea a. Por ejemplo,
--    mayoriasMinimales [(P1,3),(P3,5),(P4,3)] == [[P3,P4],[P1,P4],[P1,P3]]
```

```haskell
-- ---------------------------------------------------------------------

mayoriasMinimales :: Asamblea -> [Coalicion]
mayoriasMinimales asamblea =
    [c | (c,_) <- coalicionesMinimales asamblea (mayoria asamblea)]

-- ---------------------------------------------------------------------
-- Ejercicio 3.21. Comprobar con QuickCheck que las coaliciones
-- obtenidas por (mayoriasMinimales asamblea) son coaliciones
-- mayoritarias minimales en la asamblea.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_MayoriasMinimalesSonMayoritariasMinimales :: Asamblea2 -> Bool
prop_MayoriasMinimalesSonMayoritariasMinimales (A asamblea) =
  and [esMayoritariaMinimal c asamblea
       | c <- mayoriasMinimales asamblea]

-- La comprobación es
--    Main> quickCheck prop_MayoriasMinimalesSonMayoritariasMinimales
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Funciones auxiliares                                                --
-- ---------------------------------------------------------------------

-- (listaDe n g) es una lista de n elementos, donde cada elemento es
-- generado por g. Por ejemplo,
--    Main> muestra (listaDe 3 (arbitrary :: Gen Int))
--    [-1,1,-1]
--    [-2,-4,-1]
--    [1,-1,0]
--    [1,-1,1]
--    [1,-1,1]
--    Main> muestra (listaDe 3 (arbitrary :: Gen Bool))
--    [False,True,False]
--    [True,True,False]
--    [False,False,True]
--    [False,False,True]
--    [True,False,True]
```

```haskell
listaDe :: Int -> Gen a -> Gen [a]
listaDe n g = sequence [g | i <- [1..n]]

-- paresDeIgualLongitud genera pares de listas de igual longitud. Por
-- ejemplo,
--    Main> muestra (paresDeIgualLongitud (arbitrary :: Gen Int))
--    ([-4,5],[-4,2])
--    ([],[])
--    ([0,0],[-2,-3])
--    ([2,-2],[-2,1])
--    ([0],[-1])
--    Main> muestra (paresDeIgualLongitud (arbitrary :: Gen Bool))
--    ([False,True,False],[True,True,True])
--    ([True],[True])
--    ([],[])
--    ([False],[False])
--    ([],[])
paresDeIgualLongitud :: Gen a -> Gen ([a],[a])
paresDeIgualLongitud gen =
    do n <- arbitrary
       xs <- listaDe (abs n) gen
       ys <- listaDe (abs n) gen
       return (xs,ys)

-- generaAsamblea esun generador de datos de tipo Asamblea. Por ejemplo,
--    Main> muestra generaAsamblea
--    A [(P1,1),(P2,1),(P3,0),(P4,1),(P5,0),(P6,1),(P7,0),(P8,1)]
--    A [(P1,0),(P2,1),(P3,1),(P4,1),(P5,0),(P6,1),(P7,0),(P8,1)]
--    A [(P1,1),(P2,2),(P3,0),(P4,1),(P5,0),(P6,1),(P7,2),(P8,0)]
--    A [(P1,1),(P2,0),(P3,1),(P4,0),(P5,0),(P6,1),(P7,1),(P8,1)]
--    A [(P1,1),(P2,0),(P3,0),(P4,0),(P5,1),(P6,1),(P7,1),(P8,0)]
generaAsamblea :: Gen Asamblea2
generaAsamblea =
    do xs <- listaDe 8 (arbitrary :: Gen Integer)
       return (A (zip [P1,P2,P3,P4,P5,P6,P7,P8] (map abs xs)))

instance Arbitrary Asamblea2 where
    arbitrary   = generaAsamblea
    -- coarbitrary = undefined
```

Relación 22

Demostración de propiedades por inducción. Misceláneas

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- Esta relación contiene dos partes. En una parte se plantean
-- ejercicios de demostración de propiedades por inducción numérica
--   * numeroDeListasConSuma n = 2^(n-1),
--   * fibItAux n (fib k) (fib (k+1)) = fib (k+n),
--   * potencia x n == x^n,
-- por inducción sobre listas
--   * reverse (xs ++ ys) == reverse ys ++ reverse xs,
--   * reverse (reverse xs) = xs,
-- y por inducción sobre árboles binarios
--   * espejo (espejo x) = x,
--   * postorden (espejo x) = reverse (preorden x),
--   * reverse (preorden (espejo x)) = postorden x,
--   * nNodos (espejo x) == nNodos x,
--   * length (preorden x) == nNodos x,
--   * nNodos x <= 2^(profundidad x) - 1,
--   * nHojas x = nNodos x + 1,
--   * preordenItAux x ys = preorden x ++ ys
-- En la otra parte se plantea una miscelánea de ejercicios:
--   * las cadenas de longitud n formada con los caracteres de una cadena,
--   * polinomios con valores primos y
--   * factorización de un número.
```

```haskell
--
-- Estos ejercicios corresponden al temas 8 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-8.pdf

-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares                                 --
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Control.Monad

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    palabrasDeLongitud :: Int -> String -> [String]
-- tal que (palabrasDeLongitud n cs) es la lista de las cadenas de
-- longitud n formada con los caracteres de cs. Por ejemplo,
--    *Main> palabrasDeLongitud 3 "ab"
--    ["aaa","aab","aba","abb","baa","bab","bba","bbb"]
-- ---------------------------------------------------------------------

palabrasDeLongitud :: Int -> String -> [String]
palabrasDeLongitud 0 cs = [[]]
palabrasDeLongitud n cs =
    [x:xs | x <- cs,
            xs <- palabrasDeLongitud (n-1) cs]

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir la función
--    listaConSuma :: Int -> [[Int]]
-- que, dado un número natural n, devuelve todas las listas de enteros
-- positivos (esto es, enteros mayores o iguales que 1) cuya suma sea
-- n. Por ejemplo,
--    Main> listaConSuma 4
--    [[1,1,1,1],[1,1,2],[1,2,1],[1,3],[2,1,1],[2,2],[3,1],[4]]
-- ---------------------------------------------------------------------

listaConSuma :: Int -> [[Int]]
listaConSuma 0 = [[]]
listaConSuma n = [x:xs | x <- [1..n], xs <- listaConSuma (n-x)]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función
--    numeroDeListasConSuma :: Int -> Int
-- tal que (numeroDeListasConSuma n) es el número de elementos de
-- (listaConSuma n). Por ejemplo,
--    numeroDeListasConSuma 10  =  512
-- ---------------------------------------------------------------------

numeroDeListasConSuma :: Int -> Int
numeroDeListasConSuma = length . listaConSuma

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la constante
--    numerosDeListasConSuma :: [(Int,Int)]
-- tal que numerosDeListasConSuma es la lista de los pares formado por un
-- número natural n mayor que 0 y el número de elementos de
-- (listaConSuma n).
--
-- Calcular el valor de
--    take 10 numerosDeListasConSuma
-- ---------------------------------------------------------------------

-- La constante es
numerosDeListasConSuma :: [(Int,Int)]
numerosDeListasConSuma = [(n,numeroDeListasConSuma n) | n <- [1..]]

-- El cálculo es
--    *Main> take 10 numerosDeListasConSuma
--    [(1,1),(2,2),(3,4),(4,8),(5,16),(6,32),(7,64),(8,128),(9,256),(10,512)]

-- ---------------------------------------------------------------------
-- Ejercicio 2.4. A partir del ejercicio anterior, encontrar una fórmula
-- para calcular el valor de (numeroDeListasConSuma n) paras los
-- números n mayores que 0.
--
-- Demostrar dicha fórmula por inducción fuerte.
-- ---------------------------------------------------------------------

{-
```

```
 La fórmula es
    numeroDeListasConSuma n = 2^(n-1)
 La demostración, por inducción fuerte en n, es la siguiente:

 Caso base (n=1):
    numeroDeListasConSuma 1
    = length (listaConSuma 1)
         [por numeroDeListasConSuma]
    = length [[x:xs | x <- [1..1], xs <- listaConSuma [[]]]
         [por listaConSuma.2]
    = length [[1]]
         [por def. de listas de comprensión]
    = 1
         [por def. de length]
    = 2^(1-1)
         [por aritmética]

 Paso de inducción: Se supone que
    para todo x en [1..n-1],
       numeroDeListasConSuma x = 2^(x-1)
 Hay que demostrar que
    numeroDeListasConSuma n = 2^(n-1)
 En efecto,
    numeroDeListasConSuma n
    = length (listaConSuma n)
         [por numeroDeListasConSuma]
    = length [x:xs | x <- [1..n], xs <- listaConSuma (n-x)]
         [por listaConSuma.2]
    = sum [numeroDeListasConSuma (n-x) | x <- [1..n]]
         [por length y listas de comprensión]
    = sum [2^(n-x-1) | x <- [1..n-1]] + 1
         [por hip. de inducción y numeroDeListasConSuma]
    = 2^(n-2) + 2^(n-3) + ... + 2^1 + 2^0 + 1
    = 2^(n-1)
         [por el ejercicio 2c de la relación 15]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 2.4. A partir del ejercicio anterior, definir de manera más
-- eficiente la función numeroDeListasConSuma.
```

```haskell
-- ---------------------------------------------------------------------

numeroDeListasConSuma' :: Int -> Int
numeroDeListasConSuma' 0     = 1
numeroDeListasConSuma' (n+1) = 2^n

-- ---------------------------------------------------------------------
-- Ejercicio 2.5. Comparar la eficiencia de las dos definiciones
-- comparando el tiempo y el espacio usado para calcular
-- (numeroDeListasConSuma 20) y (numeroDeListasConSuma' 20).
-- ---------------------------------------------------------------------

-- La comparación es
--    *Main> :set +s
--    *Main> numeroDeListasConSuma 20
--    524288
--    (9.99 secs, 519419824 bytes)
--    *Main> numeroDeListasConSuma' 20
--    524288
--    (0.01 secs, 0 bytes)

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    conjetura :: Int -> Bool
-- tal que (conjetura n) se verifica si n^2+n+41 es primo. Por ejemplo,
--    conjetura 20  ==>  True
-- ---------------------------------------------------------------------

conjetura :: Int -> Bool
conjetura n = primo (n^2+n+41)

-- (primo x) se verifica si x es un número primo.
primo x = divisores x == [1,x]

-- (divisores x) es la lista de los divisores de x.
divisores x = [y | y <- [1..x], mod x y == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    conjeturaHasta :: Int -> Bool
```

```
-- tal que (conjeturaHasta n) se verifica si x^2+x+41 es primo para todo
-- los números x de 1 a n. Por ejemplo,
--    conjeturaHasta 20  ==>  True
-- ---------------------------------------------------------------------

conjeturaHasta :: Int -> Bool
conjeturaHasta n = all conjetura [1..n]

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir la constante
--    menorContraejemplo :: Int
-- tal que menorContraejemplo es el menor número x tal que x^2+x+41 no
-- es primo.
--
-- Calcular el valor de menorContraejemplo.
-- ---------------------------------------------------------------------

menorContraejemplo :: Int
menorContraejemplo = head [x| x <- [1..], not (conjetura x)]

-- La evaluación es
--    *Main> menorContraejemplo
--    40

-- ---------------------------------------------------------------------
-- Ejercicio 3.4. Definir la constante
--    menorC :: Int
-- tal que menorC es el c tal que x^2-79*x+c es primo para los m
-- primeros números con m mayor que 40.
--
-- Calcular menorC y el mayor m tal que para todos los x entre 1 y m
-- x^2-79*x+menorC es primo.
-- ---------------------------------------------------------------------

-- La definición es
menorC = head [c | c <- [1..],
                   head [x | x <- [1..],
                             not (primo (x^2-79*x+c))] > 40]

-- El cálculo de menorC es
```

```
--    *Main> menorC
--    1601

-- Para calcular el menor m, se define
menorM = head [x | x <- [1..],
                   not (primo (x^2-79*x+1601))]

-- El cálculo del menor m es
--    *Main> menorM
--    80

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir la función
--    menorFactor :: Integer -> Integer
-- tal que (menorFactor n) es el menor factor primo de n. Por ejemplo,
--    menorFactor 15  ==  3
-- ---------------------------------------------------------------------

menorFactor :: Integer -> Integer
menorFactor n = head [x | x <- [2..], rem n x == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir la función
--    factorizacion :: Integer -> [Integer]
-- tal que (factorizacion n) es la lista de todos los factores primos
-- de n; es decir, es una lista de números primos cuyo producto es
-- n. Por ejemplo,
--    factorizacion 300  ==  [2,2,3,5,5]
-- ---------------------------------------------------------------------

factorizacion :: Integer -> [Integer]
factorizacion n | primo n   = [n]
                | otherwise = x : factorizacion (div n x)
                where x = menorFactor n

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Comprobar con QuickCheck que para todo número natural
-- n >= 2, todos los elementos de (factorizacion n) son números primos.
-- ---------------------------------------------------------------------
```

```haskell
-- La propiedad es
prop_factorizacion_primos :: Integer -> Property
prop_factorizacion_primos n =
    n >=2 ==> all primo (factorizacion n)

-- La comprobación es
--    *Main> quickCheck prop_factorizacion_primos
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4.4. Comprobar con QuickCheck que para todo número natural
-- n >= 2, el producto de los elementos de (factorizacion n) es n.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_factorizacion_producto :: Integer -> Property
prop_factorizacion_producto n =
    n >=2 ==> product (factorizacion n) == n

-- La comprobación es
--    *Main> quickCheck prop_factorizacion_producto
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.0. La sucesión de Fibonacci
--    0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
-- puede definirse por recursión como
--    fib :: Int -> Int
--    fib 0     = 0                     -- fib.1
--    fib 1     = 1                     -- fib.2
--    fib (n+2) = (fib (n+1)) + fib n   -- fib.3
-- También puede definirse por recursición iterativa como
--    fibIt :: Int -> Int
--    fibIt n = fibItAux n 0 1
-- donde la función auxiliar se define por
--    fibItAux :: Int -> Int -> Int -> Int
--    fibItAux 0     a b = a                    -- fibItAux.1
--    fibItAux (n+1) a b = fibItAux n b (a+b)   -- fibItAux.2
-- ---------------------------------------------------------------------
```

```haskell
fib :: Int -> Int
fib 0     = 0
fib 1     = 1
fib (n+2) = fib (n+1) + fib n

fibIt :: Int -> Int
fibIt n = fibItAux n 0 1

fibItAux :: Int -> Int -> Int -> Int
fibItAux 0     a b = a
fibItAux (n+1) a b = fibItAux n b (a+b)

-- ---------------------------------------------------------------------
-- Ejercicio 5.1. Comprobar con QuickCheck que para todo número natural
-- n tal que n <= 20, se tiene que
--    fib n = fibIt n
-- ---------------------------------------------------------------------

-- La propiedad es
prop_fib :: Int -> Property
prop_fib n =
    n >= 0 && n <= 20 ==> fib n == fibIt n

-- La comprobación es
--    *Main> quickCheck prop_fib
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.2. Sea f la función definida por
--    f :: Int -> Int -> Int
--    f n k = fibItAux n (fib k) (fib (k+1))
-- Definir la función
--    grafoDeF :: Int -> [(Int,Int)]
-- tal que (grafoDeF n)
--    *Main> take 7 (grafoDeF 3)
--    [(1,3),(2,5),(3,8),(4,13),(5,21),(6,34),(7,55)]
--    *Main> take 7 (grafoDeF 5)
--    [(1,8),(2,13),(3,21),(4,34),(5,55),(6,89),(7,144)]
-- ---------------------------------------------------------------------
```

```haskell
f :: Int -> Int -> Int
f n k = fibItAux n (fib k) (fib (k+1))

grafoDeF :: Int -> [(Int,Int)]
grafoDeF n = [(k, f n k) | k <- [1..]]

-- ---------------------------------------------------------------------
-- Ejercicio 5.3. Comprobar con QuickCheck que para todo par de números
-- naturales n, k tales que n+k <= 20, se tiene que
--    fibItAux n (fib k) (fib (k+1)) = fib (k+n)
-- ---------------------------------------------------------------------

-- La propiedad es
prop_fibItAux :: Int -> Int -> Property
prop_fibItAux n k =
    n >= 0 && k >= 0 && n+k <= 20 ==>
    fibItAux n (fib k) (fib (k+1)) == fib (k+n)

-- La comprobación es
--    *Main> quickCheck prop_fibItAux
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 5.4. Demostrar por inducción que para todo n y todo k,
--    fibItAux n (fib k) (fib (k+1)) = fib (k+n)
-- ---------------------------------------------------------------------

{-
 Demostración: Por inducción en n se prueba que
    para todo k, fibItAux n (fib k) (fib (k+1)) = fib (k+n)

 Caso base (n=0): Hay que demostrar que
    para todo k, fibItAux 0 (fib k) (fib (k+1)) = fib k
 En efecto, sea k un número natural. Se tiene
    fibItAux 0 (fib k) (fib (k+1))
    = fib k                          [por fibItAux.1]

 Paso de inducción: Se supone la hipótesis de inducción
    para todo k, fibItAux n (fib k) (fib (k+1)) = fib (k+n)
 Hay que demostrar que
```

```
    para todo k, fibItAux (n+1) (fib k) (fib (k+1)) = fib (k+n+1)
 En efecto. Sea k un número natural,
    fibItAux (n+1) (fib k) (fib (k+1))
    = fibItAux n (fib (k+1)) ((fib k) + (fib (k+1)))
         [por fibItAux.2]
    = fibItAux n (fib (k+1)) (fib (k+2))
         [por fib.3]
    = fib (n+k+1)
         [por hipótesis de inducción]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 5.5. Demostrar que para todo n,
--    fibIt n = fib n
-- ---------------------------------------------------------------------

{-
 Demostración
    fibIt n
    = fibItAux n 0 1               [por fibIt]
    = fibItAux n (fib 0) (fib 1)   [por fib.1 y fib.2]
    = fib n                        [por ejercicio 5.4]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 6.1. La función potencia puede definirse por
--    potencia :: Int -> Int -> Int
--    potencia x 0 = 1
--    potencia x n | even n    = potencia (x*x) (div n 2)
--                 | otherwise = x * potencia (x*x) (div n 2)
-- Comprobar con QuickCheck que para todo número natural n y todo
-- número entero x, (potencia x n) es x^n.
-- ---------------------------------------------------------------------

potencia :: Integer -> Integer -> Integer
potencia x 0 = 1
potencia x n | even n    = potencia (x*x) (div n 2)
             | otherwise = x * potencia (x*x) (div n 2)

-- La propiedad es
```

```
prop_potencia :: Integer -> Integer -> Property
prop_potencia x n =
    n >= 0 ==> potencia x n == x^n

-- La comprobación es
--    *Main> quickCheck prop_potencia
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 6.2. Demostrar por inducción que que para todo número
-- natural n y todo número entero x, (potencia x n) es x^n
-- ---------------------------------------------------------------------

{-
 Demostración: Por inducción en n.

 Caso base: Hay que demostrar que
    para todo x, potencia x 0 = 2^0
 Sea x un número entero, entonces
    potencia x 0
    = 1              [por potencia.1]
    = 2^0            [por aritmética]

 Paso de inducción: Se supone que n>0 y la hipótesis de inducción:
    para todo m<n y para todo x, potencia x (n-1) = x^(n-1)
 Tenemos que demostrar que
    para todo x, potencia x n = x^n
 Lo haremos distinguiendo casos según la paridad de n.

 Caso 1: Supongamos que n es par. Entonces, existe un k tal que
    n = 2*k.                    (1)
 Por tanto,
    potencia  n
    = potencia (x*x) (div n 2)  [por potencia.2]
    = potencia (x*x) k          [por (1)]
    = (x*x)^k                   [por hip. de inducción]
    = x^(2*k)                   [por aritmética]
    = x^n                       [por (1)]

 Caso 2: Supongamos que n es impar. Entonces, existe un k tal que
```

```
    n = 2*k+1.                          (2)
 Por tanto,
    potencia  n
    = x * potencia (x*x) (div n 2)   [por potencia.3]
    = x * potencia (x*x) k           [por (1)]
    = x * (x*x)^k                    [por hip. de inducción]
    = x^(2*k+1)                      [por aritmética]
    = x^n                            [por (1)]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 7.1. Comprobar con QuickCheck que para todo par de listas
-- xs, ys se tiene que
--    reverse (xs ++ ys) == reverse ys ++ reverse xs
-- ---------------------------------------------------------------------

-- La propiedad es
prop_reverse_conc :: [Int] -> [Int] -> Bool
prop_reverse_conc xs ys =
    reverse (xs ++ ys) == reverse ys ++ reverse xs

-- La comprobación es
--    *Main> quickCheck prop_reverse_conc
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 7.2. Demostrar por inducción que para todo par de listas
-- xs, ys se tiene que
--    reverse (xs ++ ys) == reverse ys ++ reverse xs
--
-- Las definiciones de reverse y (++) son
--    reverse [] = []                      -- reverse.1
--    reverse (x:xs) = reverse xs ++ [x]   -- reverse.2
--
--    [] ++ ys     = ys                    -- ++.1
--    (x:xs) ++ ys = x : (xs ++ ys)        -- ++.2
-- ---------------------------------------------------------------------

{-
 Demostración por inducción en xs.
```

```
 Caso base: Hay que demostrar que para toda ys,
    reverse ([] ++ ys) == reverse ys ++ reverse []
 En efecto,
    reverse ([] ++ ys)
    = reverse ys                  [por ++.1]
    = reverse ys ++ []            [por propiedad de ++]
    = reverse ys ++ reverse []    [por reverse.1]

 Paso de inducción: Se supone que para todo ys,
    reverse (xs ++ ys) == reverse ys ++ reverse xs
 Hay que demostrar que para todo ys,
    reverse ((x:xs) ++ ys) == reverse ys ++ reverse (x:xs)
 En efecto,
    reverse ((x:xs) ++ ys)
    = reverse (x:(xs ++ ys))               [por ++.2]
    = reverse (xs ++ ys) ++ [x]            [por reverse.2]
    = (reverse ys ++ reverse xs) ++ [x]    [por hip. de inducción]
    = reverse ys ++ (reverse xs ++ [x])    [por asociativa de ++]
    = reverse ys ++ reverse (x:xs)         [por reverse.2]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 7.3. Demostrar por inducción que para toda lista xs,
--    reverse (reverse xs) = xs
-- ---------------------------------------------------------------------

{-
 Demostración por inducción en xs.

 Caso Base: Hay que demostrar que
    reverse (reverse []) = []
 En efecto,
    reverse (reverse [])
    = reverse []           [por reverse.1]
    = []                   [por reverse.1]

 Paso de inducción: Se supone que
    reverse (reverse xs) = xs
 Hay que demostrar que
```

```
    reverse (reverse (x:xs)) = x:xs
 En efecto,
    reverse (reverse (x:xs))
    = reverse (reverse xs ++ [x])           [por reverse.2]
    = reverse [x] ++ reverse (reverse xs)   [por ejercicio 7.2]
    = [x] ++ reverse (reverse xs)           [por reverse]
    = [x] ++ xs                             [por hip. de inducción]
    = x:xs                                  [por ++.2]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 8.0. En los siguientes ejercicios se demostrarán
-- propiedades de los árboles binarios definidos como sigue
--    data Arbol a = Hoja
--                 | Nodo a (Arbol a) (Arbol a)
--                 deriving (Show, Eq)
-- En los ejemplos se usará el siguiente árbol
--    arbol = Nodo 9
--                   (Nodo 3
--                         (Nodo 2 Hoja Hoja)
--                         (Nodo 4 Hoja Hoja))
--                   (Nodo 7 Hoja Hoja)
-- ---------------------------------------------------------------------

data Arbol a = Hoja
             | Nodo a (Arbol a) (Arbol a)
             deriving (Show, Eq)

arbol = Nodo 9
               (Nodo 3
                     (Nodo 2 Hoja Hoja)
                     (Nodo 4 Hoja Hoja))
               (Nodo 7 Hoja Hoja)

-- ---------------------------------------------------------------------
-- Nota. Para comprobar propiedades de árboles con QuickCheck se
-- utilizará el siguiente generador.
-- ---------------------------------------------------------------------

instance Arbitrary a => Arbitrary (Arbol a) where
```

```
  arbitrary = sized arbol
    where
      arbol 0       = return Hoja
      arbol n | n>0 = oneof [return Hoja,
                             liftM3 Nodo arbitrary subarbol subarbol]
                      where subarbol = arbol (div n 2)

-- ---------------------------------------------------------------------
-- Ejercicio 8.1. Definir la función
--    espejo :: Arbol a -> Arbol a
-- tal que (espejo x) es la imagen especular del árbol x. Por ejemplo,
--    *Main> espejo arbol
--    Nodo 9
--         (Nodo 7 Hoja Hoja)
--         (Nodo 3
--               (Nodo 4 Hoja Hoja)
--               (Nodo 2 Hoja Hoja))
-- ---------------------------------------------------------------------

espejo :: Arbol a -> Arbol a
espejo Hoja         = Hoja
espejo (Nodo x i d) = Nodo x (espejo d) (espejo i)

-- ---------------------------------------------------------------------
-- Ejercicio 8.2. Comprobar con QuickCheck que para todo árbol x,
--    espejo (espejo x) = x
-- ---------------------------------------------------------------------

prop_espejo :: Arbol Int -> Bool
prop_espejo x =
    espejo (espejo x) == x

-- ---------------------------------------------------------------------
-- Ejercicio 8.3. Demostrar por inducción que para todo árbol x,
--    espejo (espejo x) = x
-- ---------------------------------------------------------------------

{-
 Demostración por inducción en x
```

```
 Caso base: Hay que demostrar que
    espejo (espejo Hoja) = Hoja
 En efecto,
    espejo (espejo Hoja)
    = espejo Hoja          [por espejo.1]
    = Hoja                 [por espejo.1]

 Paso de inducción: Se supone la hipótesis de inducción
    espejo (espejo i) = i
    espejo (espejo d) = d
 Hay que demostrar que
    espejo (espejo (Nodo x i d)) = Nodo x i d
 En efecto,
    espejo (espejo (Nodo x i d))
    = espejo (Nodo x (espejo d) (espejo i))             [por espejo.2]
    = Nodo x (espejo (espejo i)) (espejo (espejo d))    [por espejo.2]
    = Nodo x i d                                        [por hip. inducción]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 8.4. Definir la función
--    preorden :: Arbol a -> [a]
-- tal que (preorden x) es la lista correspondiente al recorrido
-- preorden del árbol x; es decir, primero visita la raíz del árbol, a
-- continuación recorre el subárbol izquierdo y, finalmente, recorre el
-- subárbol derecho. Por ejemplo,
--    *Main> arbol
--    Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--    *Main> preorden arbol
--    [9,3,2,4,7]
-- ---------------------------------------------------------------------

preorden :: Arbol a -> [a]
preorden Hoja         = []
preorden (Nodo x i d) = x : (preorden i ++ preorden d)

-- ---------------------------------------------------------------------
-- Ejercicio 8.5. Definir la función
--    postorden :: Arbol a -> [a]
-- tal que (postorden x) es la lista correspondiente al recorrido
```

```haskell
-- postorden del árbol x; es decir, primero recorre el subárbol
-- izquierdo, a continuación el subárbol derecho y, finalmente, la raíz
-- del árbol. Por ejemplo,
--    *Main> arbol
--    Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--    *Main> postorden arbol
--    [2,4,3,7,9]
-- ---------------------------------------------------------------------

postorden :: Arbol a -> [a]
postorden Hoja         = []
postorden (Nodo x i d) = postorden i ++ postorden d ++ [x]

-- ---------------------------------------------------------------------
-- Ejercicio 8.6. Comprobar con QuickCheck que para todo árbol x,
--    postorden (espejo x) = reverse (preorden x)
-- ---------------------------------------------------------------------

-- La propiedad es
prop_recorrido :: Arbol Int -> Bool
prop_recorrido x =
   postorden (espejo x) == reverse (preorden x)

-- La comprobación es
--    *Main> quickCheck prop_recorrido
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 8.7. Demostrar por inducción que para todo árbol x,
--    postorden (espejo x) = reverse (preorden x)
-- ---------------------------------------------------------------------

{-
 Demostración por inducción en x.

 Caso base: Hay que demostrar que
    postorden (espejo Hoja) = reverse (preorden Hoja)
 En efecto,
    postorden (espejo Hoja)
    = postorden Hoja           [por espejo.1]
```

```
    = []                          [por postorden.1]
    = reverse []                  [por reverse.1]
    = reverse (preorden Hoja)     [por preorden.1]

 Paso de inducción: Se supone la hipótesis de inducción
    postorden (espejo i) = reverse (preorden i)
    postorden (espejo d) = reverse (preorden d)
 Hay que demostrar que
    postorden (espejo (Nodo x i d)) = reverse (preorden (Nodo x i d))
 En efecto,
    postorden (espejo (Nodo x i d))
    = postorden (Nodo x (espejo d) (espejo i))   [por espejo.2]
    = postorden (espejo d) ++ postorden (espejo i) ++ [x]
                                                 [por postorden.2]
    = reverse (preorden d) ++ reverse (preorden i) ++ x
                                                 [por hip. inducción]
    = reverse ([x] ++ preorden (espejo i) ++ preorden (espejo d))
                                                 [por ejercicio 7.1]
    = reverse (preorden (Nodo x i d))            [por preorden.1]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 8.8. Comprobar con QuickCheck que para todo árbol binario
-- x, se tiene que
--    reverse (preorden (espejo x)) = postorden x
-- ---------------------------------------------------------------------

-- La propiedad es
prop_reverse_preorden_espejo :: Arbol Int -> Bool
prop_reverse_preorden_espejo x =
   reverse (preorden (espejo x)) == postorden x

-- La comprobación es
--    *Main> quickCheck prop_reverse_preorden_espejo
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 8.9. Demostrar que para todo árbol binario x, se tiene que
--    reverse (preorden (espejo x)) = preorden x
-- ---------------------------------------------------------------------
```

```
{-
 Demostración:
    reverse (preorden (espejo x))
    = postorden (espejo (espejo x))    [por ejercicio 8.7]
    = postorden x                      [por ejercicio 8.3]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 8.10. Definir la función
--    nNodos :: Arbol a -> Int
-- tal que (nNodos x) es el número de nodos del árbol x. Por ejemplo,
--    *Main> arbol
--    Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--    *Main> nNodos arbol
--    5
-- ---------------------------------------------------------------------

nNodos :: Arbol a -> Int
nNodos Hoja         = 0
nNodos (Nodo x i d) = 1 + nNodos i + nNodos d

-- ---------------------------------------------------------------------
-- Ejercicio 8.11. Comprobar con QuickCheck que el número de nodos de la
-- imagen especular de un árbol es el mismo que el número de nodos del
-- árbol.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_nNodos_espejo :: Arbol Int -> Bool
prop_nNodos_espejo x =
   nNodos (espejo x) == nNodos x

-- La comprobación es
--    *Main> quickCheck prop_nNodos_espejo
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 8.12. Demostrar por inducción que el número de nodos de la
-- imagen especular de un árbol es el mismo que el número de nodos del
```

```
-- árbol.
-- ---------------------------------------------------------------------

{-
 Demostración: Hay que demostrar, por inducción en x, que
    nNodos (espejo x) == nNodos x

 Caso base: Hay que demostrar que
    nNodos (espejo Hoja) == nNodos Hoja
 En efecto,
    nNodos (espejo Hoja)
    = nNodos Hoja           [por espejo.1]

 Paso de inducción: Se supone la hipótesis de inducción
    nNodos (espejo i) == nNodos i
    nNodos (espejo d) == nNodos d
 Hay que demostrar que
    nNodos (espejo (Nodo x i d)) == nNodos (Nodo x i d)
 En esfecto,
    nNodos (espejo (Nodo x i d))
    = nNodos (Nodo x (espejo d) (espejo i))       [por espejo.2]
    = 1 + nNodos (espejo d) + nNodos (espejo i)   [por nNodos.2]
    = 1 + nNodos d + nNodos i                     [por hip.de inducción]
    = 1 + nNodos i + nNodos d                     [por aritmética]
    = nNodos (Nodo x i d)                         [por nNodos.2]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 8.13. Comprobar con QuickCheck que la longitud de la lista
-- obtenida recorriendo un árbol en sentido preorden es igual al número
-- de nodos del árbol.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_length_preorden :: Arbol Int -> Bool
prop_length_preorden x =
   length (preorden x) == nNodos x

-- La comprobación es
--    *Main> quickCheck prop_length_preorden
```

```
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 8.14. Demostrar por inducción que la longitud de la lista
-- obtenida recorriendo un árbol en sentido preorden es igual al número
-- de nodos del árbol.
-- ---------------------------------------------------------------------

{-
 Demostración: Por inducción en x, hay que demostrar que
    length (preorden x) == nNodos x

 Caso base: Hay que demostrar que
    length (preorden Hoja) = nNodos Hoja
 En efecto,
    length (preorden Hoja)
    = length []              [por preorden.1]
    = 0                      [por length.1]
    = nNodos Hoja            [por nNodos.1]

 Paso de inducción: Se supone la hipótesis de inducción
    length (preorden i) == nNodos i
    length (preorden d) == nNodos d
 Hay que demostrar que
    length (preorden (Nodo x i d)) == nNodos (Nodo x i d)
 En efecto,
    length (preorden (Nodo x i d))
    = length ([x] ++ (peorden i) ++ (preorden d))
         [por preorden.2]
    = length [x] + length (preorden i) + length (preorden d)
         [propiedad de length: length (xs++ys) = length xs + length ys]
    = 1 + length (preorden i) + length (preorden d)
         [por def. de length]
    = 1 + nNodos i + nNodos d
         [por hip. de inducción]
    = nNodos (x i d)
         [por nNodos.2]
-}

-- ---------------------------------------------------------------------
```

```haskell
-- Ejercicio 8.15. Definir la función
--    profundidad :: Arbol a -> Int
-- tal que (profundidad x) es la profundidad del árbol x. Por ejemplo,
--    *Main> arbol
--    Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--    *Main> profundidad arbol
--    3
-- ---------------------------------------------------------------------

profundidad :: Arbol a -> Int
profundidad Hoja = 0
profundidad (Nodo x i d) = 1 + max (profundidad i) (profundidad d)

-- ---------------------------------------------------------------------
-- Ejercicio 8.16. Comprobar con QuickCheck que para todo árbol biario
-- x, se tiene que
--    nNodos x <= 2^(profundidad x) - 1
-- ---------------------------------------------------------------------

-- La propiedad es
prop_nNodosProfundidad :: Arbol Int -> Bool
prop_nNodosProfundidad x =
   nNodos x <= 2^(profundidad x) - 1

-- La comprobación es
--    *Main> quickCheck prop_nNodosProfundidad
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 8.17. Demostrar por inducción que para todo árbol binario
-- x, se tiene que
--    nNodos x <= 2^(profundidad x) - 1
-- ---------------------------------------------------------------------

{-
 Demostración por inducción en x

 Caso base: Hay que demostrar que
    nNodos Hoja <= 2^(profundidad Hoja) - 1
 En efecto,
```

```
    nNodos Hoja
    = 0                             [por nNodos.1]
    = 2^0 - 1                       [por aritmética]
    = 2^(profundidad Hoja) - 1      [por profundidad.1]

 Paso de inducción: Se supone la hipótesis de inducción
    nNodos i <= 2^(profundidad i) - 1
    nNodos d <= 2^(profundidad d) - 1
 Hay que demostrar que
    nNodos (Nodo x i d) <= 2^(profundidad (Nodo x i d)) - 1
 En efecto,
    nNodos (Nodo x i d)
    =  1 + nNodos i + nNodos d
          [por nNodos.1]
    <= 1 + (2^(profundidad i) - 1) + (2^(profundidad d) - 1)
          [por hip. de inducción]
    =  2^(profundidad i) + 2^(profundidad d) - 1
          [por aritmética]
    <= 2^máx(profundidad i,profundidad d)+2^máx(profundidad i,profundidad d)-1
          [por aritmética]
    =  2*2^máx(profundidad i,profundidad d) - 1
          [por aritmética]
    =  2^(1+máx(profundidad i,profundidad d)) - 1
          [por aritmética]
    =  2^profundidad(Nodo x i d) - 1
          [por profundidad.2]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 8.18. Definir la función
--    nHojas :: Arbol a -> Int
-- tal que (nHojas x) es el número de hojas del árbol x. Por ejemplo,
--    *Main> arbol
--    Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--    *Main> nHojas arbol
--    6
-- ---------------------------------------------------------------------

nHojas :: Arbol a -> Int
nHojas Hoja         = 1
```

```
nHojas (Nodo x i d) = nHojas i + nHojas d

-- ---------------------------------------------------------------------
-- Ejercicio 8.19. Comprobar con QuickCheck que en todo árbol binario el
-- número de sus hojas es igual al número de sus nodos más uno.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_nHojas :: Arbol Int -> Bool
prop_nHojas x =
    nHojas x == nNodos x + 1

-- La comprobación es
--    *Main> quickCheck prop_nHojas
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 8.20. Demostrar por inducción que en todo árbol binario el
-- número de sus hojas es igual al número de sus nodos más uno.
-- ---------------------------------------------------------------------

{-
 Demostración: Hay que demostrar, por inducción en x, que
    nHojas x = nNodos x + 1

 Caso base: Hay que demotrar que
    nHojas Hoja = nNodos Hoja + 1
 En efecto,
    nHojas Hoja
    = 1                 [por nHojas.1]
    = 0 + 1             [por aritmética]
    = nNodos Hoja + 1   [por nNodos.1]

 Paso de inducción: Se supone la hipótesis de inducción
    nHojas i = nNodos i + 1
    nHojas d = nNodos d + 1
 Hay que demostrar que
    nHojas (Nodo x i d) = nNodos (Nodo x i d) + 1
 En efecto,
    nHojas (Nodo x i d)
```

```
    = nHojas i + nHojas d                [por nHojas.2]
    = (nNodos i + 1) + (nNodos d +1)     [por hip. de inducción]
    = (1 + nNodos i + nNodos d) + 1      [por aritmética]
    = nNodos (Nodo x i d) + 1            [por nNodos.2]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 8.21. Definir, usando un acumulador, la función
--    preordenIt :: Arbol a -> [a]
-- tal que (preordenIt x) es la lista correspondiente al recorrido
-- preorden del árbol x; es decir, primero visita la raíz del árbol, a
-- continuación recorre el subárbol izquierdo y, finalmente, recorre el
-- subárbol derecho. Por ejemplo,
--    *Main> arbol
--    Nodo 9 (Nodo 3 (Nodo 2 Hoja Hoja) (Nodo 4 Hoja Hoja)) (Nodo 7 Hoja Hoja)
--    *Main> preordenIt arbol
--    [9,3,2,4,7]
-- Nota: No usar (++) en la definición
-- ---------------------------------------------------------------------

preordenIt :: Arbol a -> [a]
preordenIt x = preordenItAux x []

preordenItAux :: Arbol a -> [a] -> [a]
preordenItAux Hoja xs         = xs
preordenItAux (Nodo x i d) xs = x : preordenItAux i (preordenItAux d xs)

-- ---------------------------------------------------------------------
-- Ejercicio 8.22. Comprobar con QuickCheck que preordenIt es
-- equivalente a preorden.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_preordenIt :: Arbol Int -> Bool
prop_preordenIt x =
    preordenIt x == preorden x

-- La comprobación es
--    *Main> quickCheck prop_preordenIt
--    OK, passed 100 tests.
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 8.22. Demostrar que preordenIt es equivalente a preorden.
-- ---------------------------------------------------------------------

prop_preordenItAux :: Arbol Int -> [Int] -> Bool
prop_preordenItAux x ys =
   preordenItAux x ys == preorden x ++ ys

{-
 Demostración: La propiedad es consecuencia del siguiente lema:

 Lema: Para todo árbol binario x, se tiene que
    para toda ys, preordenItAux x ys = preorden x ++ ys

 Demostración de la propiedad usando el lema:
    preordenIt x
    = preordenItAux x []    [por preordnIt]
    = preorden x ++ []      [por el lema]
    = preorden x            [propiedad de ++]

 Demostración del lema: Por inducción en x.

 Caso base: Hay que demotrar que
    para toda ys, preordenItAux Hoja ys = preorden Hoja ++ ys
 En efecto,
    preordenItAux Hoja ys
    = ys                    [por preordenItAux.1]
    = [] ++ ys              [por propiedad de ++]
    = preorden Hoja ++ ys   [por preorden.1]

 Paso de inducción: Se supone la hipótesis de inducción
    para toda ys, preordenItAux i ys = preorden i ++ ys
    para toda ys, preordenItAux d ys = preorden d ++ ys
 Hay que demostrar que
    para toda ys, preordenItAux (Nodo x i d) ys = preorden (Nodo x i d) ++ ys
 En efecto,
    preordenItAux (Nodo x i d) ys
    = x : (preordenItAux i (preordenItAux d ys))   [por preordenItAux.2]
    = x : (preordenItAux i (preorden d ++ ys))     [por hip. de inducción]
```

```
    = x : (preorden i ++ (preorden d ++ ys))       [por hip. de inducción]
    = ([x] ++ preorden i ++ preorden d) ++ ys      [por prop. de listas]
    = preorden (Nodo x i d) ++ ys                  [por preorden.2]
-}
```

Relación 23

Ecuación con factoriales

```haskell
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación de ejercicios es resolver la ecuación
--    a! * b! = a! + b! + c!
-- donde a, b y c son números naturales.

-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares                                 --
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    factorial :: Integer -> Integer
-- tal que (factorial n) es el factorial de n. Por ejemplo,
--    factorial 5  ==  120
-- ---------------------------------------------------------------------

factorial :: Integer -> Integer
factorial n = product [1..n]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la constante
--    factoriales :: [Integer]
```

```haskell
-- tal que factoriales es la lista de los factoriales de los números
-- naturales. Por ejemplo,
--    take 7 factoriales  ==  [1,1,2,6,24,120,720]
-- ---------------------------------------------------------------------

factoriales :: [Integer]
factoriales = [factorial n | n <- [0..]]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir, usando factoriales, la función
--    esFactorial :: Integer -> Bool
-- tal que (esFactorial n) se verifica si existe un  número natural m
-- tal que n es m!. Por ejemplo,
--    esFactorial 120  ==  True
--    esFactorial  20  ==  False
-- ---------------------------------------------------------------------

esFactorial :: Integer -> Bool
esFactorial n = n == head (dropWhile (<n) factoriales)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la constante
--    posicionesFactoriales :: [(Integer,Integer)]
-- tal que posicionesFactoriales es la lista de los factoriales con su
-- posición. Por ejemplo,
--    *Main> take 7 posicionesFactoriales
--    [(0,1),(1,1),(2,2),(3,6),(4,24),(5,120),(6,720)]
-- ---------------------------------------------------------------------

posicionesFactoriales :: [(Integer,Integer)]
posicionesFactoriales = zip [0..] factoriales

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    invFactorial :: Integer -> Maybe Integer
-- tal que (invFactorial x) es (Just n) si el factorial de n es x y es
-- Nothing, en caso contrario. Por ejemplo,
--    invFactorial 120  == Just 5
--    invFactorial 20   == Nothing
-- ---------------------------------------------------------------------
```

```haskell
invFactorial :: Integer -> Maybe Integer
invFactorial x
    | esFactorial x = Just (head [n | (n,y) <- posicionesFactoriales, y==x])
    | otherwise     = Nothing

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la constante
--    pares :: [(Integer,Integer)]
-- tal que pares es la lista de todos los pares de números naturales. Por
-- ejemplo,
--    *Main> take 11 pares
--    [(0,0),(0,1),(1,1),(0,2),(1,2),(2,2),(0,3),(1,3),(2,3),(3,3),(0,4)]
-- ---------------------------------------------------------------------

pares :: [(Integer,Integer)]
pares = [(x,y) | y <- [0..], x <- [0..y]]

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la constante
--    solucionFactoriales :: (Integer,Integer,Integer)
-- tal que solucionFactoriales es una terna (a,b,c) que es una solución
-- de la ecuación
--    a! * b! = a! + b! + c!
-- Calcular el valor de solucionFactoriales.
-- ---------------------------------------------------------------------

solucionFactoriales :: (Integer,Integer,Integer)
solucionFactoriales = (a,b,c)
    where (a,b)  = head [(x,y) | (x,y) <- pares,
                                 esFactorial (f x * f y - f x - f y)]
          f      = factorial
          Just c = invFactorial (f a * f b - f a - f b)

-- El cálculo es
--    *Main> solucionFactoriales
--    (3,3,4)

-- ---------------------------------------------------------------------
-- Ejercicio 8. Comprobar con QuickCheck que solucionFactoriales es la
```

```haskell
-- única solución de la ecuación
--    a! * b! = a! + b! + c!
-- con a, b y c números naturales
-- ---------------------------------------------------------------------

prop_solucionFactoriales :: Integer -> Integer -> Integer -> Property
prop_solucionFactoriales x y z =
    x >= 0 && y >= 0 && (x,y,z) /= solucionFactoriales
    ==> not (esFactorial (f x * f y - f x - f y))
    where f = factorial

-- ---------------------------------------------------------------------
-- Nota: El ejercicio se basa en el artículo "Ecuación con factoriales"
-- del blog Gaussianos publicado en
--    http://gaussianos.com/ecuacion-con-factoriales
-- ---------------------------------------------------------------------
```

Relación 24

El TAD de las pilas

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación de ejercicios es definir funciones sobre
-- el TAD de las pilas, utilizando las implementaciones estudiadas en el
-- tema 14 que se pueden descargar desde
--    http://www.cs.us.es/~jalonso/cursos/i1m/codigos.zip
--
-- Las transparencias del tema 14 se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-14.pdf
--
-- Para realizar los ejercicios hay que descargar el código de los TAD
-- que se encuentra en
--    http://www.cs.us.es/~jalonso/cursos/i1m/codigos.zip
--
-- Hay que hacer los ejercicios con la primera implementación
-- (PilaConTipoDeDatoAlgebraico) y comprobar que las definiciones
-- también son válidas con la segunda implementación (PilaConListas).

-- ---------------------------------------------------------------------
-- Importación de librerías                                            --
-- ---------------------------------------------------------------------

import Data.List

-- Hay que elegir una implementación del TAD pilas.
import PilaConTipoDeDatoAlgebraico
```

```haskell
-- import PilaConListas

import Data.List
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejemplos
-- ---------------------------------------------------------------------

-- A lo largo de esta relación de ejercicios usaremos los siguientes
-- ejemplos de pila
p1, p2, p3, p4, p5 :: Pila Int
p1 = foldr apila vacia [1..20]
p2 = foldr apila vacia [2,5..18]
p3 = foldr apila vacia [3..10]
p4 = foldr apila vacia [4,-1,7,3,8,10,0,3,3,4]
p5 = foldr apila vacia [1..5]

-- ---------------------------------------------------------------------
-- Ejercicio 1: Definir la función
--    filtraPila :: (a -> Bool) -> Pila a -> Pila a
-- tal que (filtraPila p q) es la pila obtenida con los elementos de
-- pila q que verifican el predicado p, en el mismo orden. Por ejemplo,
--    ghci> p1
--    1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20|-
--    ghci> filtraPila even p1
--    2|4|6|8|10|12|14|16|18|20|-
-- ---------------------------------------------------------------------

filtraPila :: (a -> Bool) -> Pila a -> Pila a
filtraPila p q
    | esVacia q = vacia
    | p cq      = apila cq (filtraPila p dq)
    | otherwise = filtraPila p dq
    where cq = cima q
          dq = desapila q

-- ---------------------------------------------------------------------
-- Ejercicio 2: Definir la función
--    mapPila :: (a -> a) -> Pila a -> Pila a
```

```
-- tal que (mapPila f p) es la pila formada con las imágenes por f de
-- los elementos de pila p, en el mismo orden. Por ejemplo,
--    ghci> mapPila (+7) p1
--    8|9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|-
-- ---------------------------------------------------------------------

mapPila :: (a -> a) -> Pila a -> Pila a
mapPila f p
    | esVacia p = p
    | otherwise = apila (f cp) (mapPila f dp)
    where cp = cima p
          dp = desapila p

-- ---------------------------------------------------------------------
-- Ejercicio 3: Definir la función
--    pertenecePila :: Eq a => a -> Pila a -> Bool
-- tal que (pertenecePila y p) se verifica si y es un elemento de la
-- pila p. Por ejemplo,
--    pertenecePila 7 p1  == True
--    pertenecePila 70 p1 == False
-- ---------------------------------------------------------------------

pertenecePila :: Eq a => a -> Pila a -> Bool
pertenecePila x p
    | esVacia p  = False
    | otherwise  = x == cp || pertenecePila x dp
    where cp = cima p
          dp = desapila p

-- ---------------------------------------------------------------------
-- Ejercicio 4: Definir la función
--    contenidaPila :: Eq a => Pila a -> Pila a -> Bool
-- tal que (contenidaPila p1 p2) se verifica si todos los elementos de
-- de la pila p1 son elementos de la pila p2. Por ejemplo,
--    contenidaPila p2 p1  == True
--    contenidaPila p1 p2  == False
-- ---------------------------------------------------------------------

contenidaPila :: Eq a => Pila a -> Pila a -> Bool
contenidaPila p1 p2
```

```
    | esVacia p1 = True
    | otherwise  = pertenecePila cp1 p2 && contenidaPila dp1 p2
    where cp1 = cima p1
          dp1 = desapila p1

-- ---------------------------------------------------------------------
-- Ejercicio 4: Definir la función
--    prefijoPila :: Eq a => Pila a -> Pila a -> Bool
-- tal que (prefijoPila p1 p2) se verifica si la pila p1 es justamente
-- un prefijo de la pila p2. Por ejemplo,
--    prefijoPila p3 p2 == False
--    prefijoPila p5 p1 == True
-- ---------------------------------------------------------------------

prefijoPila :: Eq a => Pila a -> Pila a -> Bool
prefijoPila p1 p2
    | esVacia p1 = True
    | esVacia p2 = False
    | otherwise  = cp1 == cp2 && prefijoPila dp1 dp2
    where cp1 = cima p1
          dp1 = desapila p1
          cp2 = cima p2
          dp2 = desapila p2

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    subPila :: Eq a => Pila a -> Pila a -> Bool
-- tal que (subPila p1 p2) se verifica si p1 es una subpila de p2. Por
-- ejemplo,
--    subPila p2 p1 == False
--    subPila p3 p1 == True
-- ---------------------------------------------------------------------

subPila :: Eq a => Pila a -> Pila a -> Bool
subPila p1 p2
    | esVacia p1 = True
    | esVacia p2 = False
    | cp1 == cp2 = prefijoPila dp1 dp2
    | otherwise  = subPila dp1 dp2
    where cp1 = cima p1
```

```
          dp1 = desapila p1
          cp2 = cima p2
          dp2 = desapila p2

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    ordenadaPila :: Ord a => Pila a -> Bool
-- tal que (ordenadaPila p) se verifica si los elementos de la pila p
-- están ordenados en orden creciente. Por ejemplo,
--    ordenadaPila p1 == True
--    ordenadaPila p4 == False
-- ---------------------------------------------------------------------

ordenadaPila :: Ord a => Pila a -> Bool
ordenadaPila p
    | esVacia p  = True
    | esVacia dp = True
    | otherwise = cp <= cdp && ordenadaPila dp
    where cp  = cima p
          dp  = desapila p
          cdp = cima dp

-- ---------------------------------------------------------------------
-- Ejercicio 7.1. Definir la función
--    lista2Pila :: [a] -> Pila a
-- tal que (lista2Pila xs) es la pila formada por los elementos de
-- xs. Por ejemplo,
--    lista2Pila [1..6] == 1|2|3|4|5|6|-
-- ---------------------------------------------------------------------

lista2Pila :: [a] -> Pila a
lista2Pila = foldr apila vacia

-- ---------------------------------------------------------------------
-- Ejercicio 7.2. Definir la función
--    pila2Lista :: Pila a -> [a]
-- tal que (pila2Lista p) es la lista formada por los elementos de la
-- lista p. Por ejemplo,
--    pila2Lista p2 == [2,5,8,11,14,17]
-- ---------------------------------------------------------------------
```

```haskell
pila2Lista :: Pila a -> [a]
pila2Lista p
    | esVacia p = []
    | otherwise = cp : pila2Lista dp
    where cp = cima p
          dp = desapila p

-- ---------------------------------------------------------------------
-- Ejercicio 7.3. Comprobar con QuickCheck que la función pila2Lista es
-- la inversa de lista2Pila, y recíprocamente.
-- ---------------------------------------------------------------------

prop_pila2Lista p =
    lista2Pila (pila2Lista p) == p

-- ghci> quickCheck prop_pila2Lista
-- +++ OK, passed 100 tests.

prop_lista2Pila xs =
    pila2Lista (lista2Pila xs) == xs

-- ghci> quickCheck prop_lista2Pila
-- +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 8.1. Definir la función
--    ordenaInserPila :: Ord a => Pila a -> Pila a
-- tal que (ordenaInserPila p) es la pila obtnida ordenando por
-- inserción los los elementos de la pila p. Por ejemplo,
--    ghci> ordenaInserPila p4
--    -1|0|3|3|3|4|4|7|8|10|-
-- ---------------------------------------------------------------------

ordenaInserPila :: (Ord a) => Pila a -> Pila a
ordenaInserPila p
    | esVacia p = p
    | otherwise = insertaPila cp (ordenaInserPila dp)
    where cp = cima p
          dp = desapila p
```

```haskell
insertaPila :: Ord a => a -> Pila a -> Pila a
insertaPila x p
    | esVacia p = apila x p
    | x < cp    = apila x p
    | otherwise = apila cp (insertaPila x dp)
    where cp = cima p
          dp = desapila p

-- ---------------------------------------------------------------------
-- Ejercicio 8.2. Comprobar con QuickCheck que la pila
--    (ordenaInserPila p)
-- está ordenada correctamente.
-- ---------------------------------------------------------------------

prop_ordenaInserPila p =
    pila2Lista (ordenaInserPila p) == sort (pila2Lista p)

-- ghci> quickCheck prop_ordenaInserPila
-- +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 9.1. Definir la función
--    nubPila :: Eq a => Pila a -> Pila a
-- tal que (nubPila p) es la pila con los elementos de p sin
-- repeticiones. Por ejemplo,
--    ghci> p4
--    4|-1|7|3|8|10|0|3|3|4|-
--    ghci> nubPila p4
--    -1|7|8|10|0|3|4|-
-- ---------------------------------------------------------------------

nubPila :: (Eq a) => Pila a -> Pila a
nubPila p
    | esVacia p           = vacia
    | pertenecePila cp dp = nubPila dp
    | otherwise           = apila cp (nubPila dp)
    where cp = cima p
          dp = desapila p
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 9.2. Definir la propiedad siguiente: "la composición de
-- las funciones nub y pila2Lista coincide con la composición de las
-- funciones pila2Lista y nubPila", y comprobarla con quickCheck.
-- En caso de ser falsa, redefinir la función nubPila para que se
-- verifique la propiedad.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_nubPila p =
    nub (pila2Lista p) == pila2Lista (nubPila p)

-- La comprobación es
--    ghci> quickCheck prop_nubPila
--    *** Failed! Falsifiable (after 8 tests):
--    -7|-2|0|-5|-7|-
--    ghci> let p = foldr apila vacia [-7,-2,0,-5,-7]
--    ghci> p
--    -7|-2|0|-5|-7|-
--    ghci> pila2Lista p
--    [-7,-2,0,-5,-7]
--    ghci> nub (pila2Lista p)
--    [-7,-2,0,-5]
--    ghci> nubPila p
--    -2|0|-5|-7|-
--    ghci> pila2Lista (nubPila p)
--    [-2,0,-5,-7]

-- Falla porque nub quita el último de los elementos repetidos de la
-- lista, mientras que nubPila quita el primero de ellos.

-- La redefinimos
nubPila' :: Eq a => Pila a -> Pila a
nubPila' p
    | esVacia p           = p
    | pertenecePila cp dp = apila cp (nubPila' (eliminaPila cp dp))
    | otherwise           = apila cp (nubPila' dp)
    where cp = cima p
          dp = desapila p
```

```haskell
eliminaPila :: Eq a => a -> Pila a -> Pila a
eliminaPila x p
    | esVacia p = p
    | x == cp   = eliminaPila x dp
    | otherwise = apila cp (eliminaPila x dp)
    where cp = cima p
          dp = desapila p

-- La propiedad es
prop_nubPila' p =
    nub (pila2Lista p) == pila2Lista (nubPila' p)

-- La comprobación es
--    ghci> quickCheck prop_nubPila'
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 10: Definir la función
--    maxPila :: Ord a => Pila a -> a
-- tal que (maxPila p) sea el mayor de los elementos de la pila p. Por
-- ejemplo,
--    ghci> p4
--    4|-1|7|3|8|10|0|3|3|4|-
--    ghci> maxPila p4
--    10
-- ---------------------------------------------------------------------

maxPila :: Ord a => Pila a -> a
maxPila p
    | esVacia p = error "pila vacia"
    | esVacia dp = cp
    | otherwise = max cp (maxPila dp)
    where cp = cima p
          dp = desapila p

-- ---------------------------------------------------------------------
-- Generador de pilas                                               --
-- ---------------------------------------------------------------------

-- genPila es un generador de pilas. Por ejemplo,
```

```
--    ghci> sample genPila
--    -
--    0|0|-
--    -
--    -6|4|-3|3|0|-
--    -
--    9|5|-1|-3|0|-8|-5|-7|2|-
--    -3|-10|-3|-12|11|6|1|-2|0|-12|-6|-
--    2|-14|-5|2|-
--    5|9|-
--    -1|-14|5|-
--    6|13|0|17|-12|-7|-8|-19|-14|-5|10|14|3|-18|2|-14|-11|-6|-
genPila :: (Num a, Arbitrary a) => Gen (Pila a)
genPila = do xs <- listOf arbitrary
             return (foldr apila vacia xs)

-- El tipo pila es una instancia del arbitrario.
instance (Arbitrary a, Num a) => Arbitrary (Pila a) where
    arbitrary = genPila
```

Relación 25

El TAD de las colas

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación de ejercicios es definir funciones sobre
-- el TAD de las colas, utilizando las implementaciones estudiadas en el
-- tema 15 que se pueden descargar desde
--    http://www.cs.us.es/~jalonso/cursos/i1m/codigos.zip
--
-- Las transparencias del tema 15 se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-15.pdf
--
-- Hay que hacer los ejercicios con la primera implementación
-- (ColasConListas) y comprobar que las definiciones también son válidas
-- con la segunda implementación (ColasConDosListas).

-- ---------------------------------------------------------------------
-- Importación de librerías                                            --
-- ---------------------------------------------------------------------

-- Hay que elegir una implementación del TAD colas:
import ColaConListas
-- import ColaConDosListas

import Data.List
import Test.QuickCheck
```

```haskell
-- ---------------------------------------------------------------------
-- Nota. A lo largo de la relación de ejercicios usaremos los siguientes
-- ejemplos de colas:
c1, c2, c3, c4, c5, c6 :: Cola Int
c1 = foldr inserta vacia [1..20]
c2 = foldr inserta vacia [2,5..18]
c3 = foldr inserta vacia [3..10]
c4 = foldr inserta vacia [4,-1,7,3,8,10,0,3,3,4]
c5 = foldr inserta vacia [15..20]
c6 = foldr inserta vacia (reverse [1..20])
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1: Definir la función
--    ultimoCola :: Cola a -> a
-- tal que (ultimoCola c) es el último elemento de la cola c. Por
-- ejemplo:
--    ultimoCola c4 == 4
--    ultimoCola c5 == 15
-- ---------------------------------------------------------------------

ultimoCola :: Cola a -> a
ultimoCola c
    | esVacia c  = error "cola vacia"
    | esVacia rc = pc
    | otherwise  = ultimoCola rc
    where pc = primero c
          rc = resto c

-- ---------------------------------------------------------------------
-- Ejercicio 2: Definir la función
--    longitudCola :: Cola a -> Int
-- tal que (longitudCola c) es el número de elementos de la cola c. Por
-- ejemplo,
--     longitudCola c2 == 6
-- ---------------------------------------------------------------------

longitudCola :: Cola a -> Int
longitudCola c
    | esVacia c = 0
```

```
    | otherwise = 1 + longitudCola rc
    where rc = resto c

-- ---------------------------------------------------------------------
-- Ejercicio 3: Definir la función
--    todosVerifican :: (a -> Bool) -> Cola a -> Bool
-- tal que (todosVerifican p c) se verifica si todos los elementos de la
-- cola c cumplen la propiedad p. Por ejemplo,
--    todosVerifican (>0) c1 == True
--    todosVerifican (>0) c4 == False
-- ---------------------------------------------------------------------

todosVerifican :: (a -> Bool) -> Cola a -> Bool
todosVerifican p c
    | esVacia c = True
    | otherwise = p pc && todosVerifican p rc
    where pc = primero c
          rc = resto c

-- ---------------------------------------------------------------------
-- Ejercicio 4: Definir la función
--    algunoVerifica :: (a -> Bool) -> Cola a -> Bool
-- tal que (algunoVerifica p c) se verifica si algún elemento de la cola
-- c cumple la propiedad p. Por ejemplo,
--   algunoVerifica (<0) c1 == False
--   algunoVerifica (<0) c4 == True
-- ---------------------------------------------------------------------

algunoVerifica :: (a -> Bool) -> Cola a -> Bool
algunoVerifica p c
    | esVacia c = False
    | otherwise = p pc || algunoVerifica p rc
    where pc = primero c
          rc = resto c

-- ---------------------------------------------------------------------
-- Ejercicio 5: Definir la función
--    ponAlaCola :: Cola a -> Cola a -> Cola a
-- tal que (ponAlaCola c1 c2) es la cola que resulta de poner los
-- elementos de c2 a la cola de c1. Por ejemplo,
```

```
--    ponAlaCola c2 c3 == C [17,14,11,8,5,2,10,9,8,7,6,5,4,3]
-- ---------------------------------------------------------------------

ponAlaCola :: Cola a -> Cola a -> Cola a
ponAlaCola c1 c2
    | esVacia c2 = c1
    | otherwise  = ponAlaCola (inserta pc2 c1) rq2
    where pc2 = primero c2
          rq2 = resto c2

-- ---------------------------------------------------------------------
-- Ejercicio 6: Definir la función
--    mezclaColas :: Cola a -> Cola a -> Cola a
-- tal que (mezclaColas c1 c2) es la cola formada por los elementos de
-- c1 y c2 colocados en una cola, de forma alternativa, empezando por
-- los elementos de c1. Por ejemplo,
--    mezclaColas c2 c4 == C [17,4,14,3,11,3,8,0,5,10,2,8,3,7,-1,4]
-- ---------------------------------------------------------------------

mezclaColas :: Cola a -> Cola a -> Cola a
mezclaColas c1 c2 = aux c1 c2 vacia
    where aux c1 c2 c
              |esVacia c1 = ponAlaCola c c2
              |esVacia c2 = ponAlaCola c c1
              |otherwise = aux rc1 rc2 (inserta pc2 (inserta pc1 c))
              where pc1 = primero c1
                    rc1 = resto c1
                    pc2 = primero c2
                    rc2 = resto c2

-- ---------------------------------------------------------------------
-- Ejercicio 7: Definir la función
--    agrupaColas :: [Cola a] -> Cola a
-- tal que (agrupaColas [c1,c2,c3,...,cn]) es la cola formada mezclando
-- las colas de la lista como sigue: mezcla c1 con c2, el resultado con
-- c3, el resultado con c4, y así sucesivamente. Por ejemplo,
--    ghci> agrupaColas [c3,c3,c4]
--    C [10,4,10,3,9,3,9,0,8,10,8,8,7,3,7,7,6,-1,6,4,5,5,4,4,3,3]
-- ---------------------------------------------------------------------
```

```haskell
agrupaColas :: [Cola a] -> Cola a
agrupaColas []           = vacia
agrupaColas [c]          = c
agrupaColas (c1:c2:colas) = agrupaColas (mezclaColas c1 c2 : colas)

-- ---------------------------------------------------------------------
-- Ejercicio 8: Definir la función
--    perteneceCola :: Eq a => a -> Cola a -> Bool
-- tal que (perteneceCola x c) se verifica si x es un elemento de la
-- cola c. Por ejemplo,
--    perteneceCola 7 c1  == True
--    perteneceCola 70 c1 == False
-- ---------------------------------------------------------------------

perteneceCola :: Eq a => a -> Cola a -> Bool
perteneceCola x c
    | esVacia c  = False
    | otherwise  = pc == x || perteneceCola x rc
    where pc = primero c
          rc = resto c

-- ---------------------------------------------------------------------
-- Ejercicio 9: Definir la función
--    contenidaCola :: Eq a => Cola a -> Cola a -> Bool
-- tal que (contenidaCola c1 c2) se verifica si todos los elementos de
-- c1 son elementos de c2. Por ejemplo,
--    contenidaCola c2 c1 == True
--    contenidaCola c1 c2 == False
-- ---------------------------------------------------------------------

contenidaCola :: Eq a => Cola a -> Cola a -> Bool
contenidaCola c1 c2
    | esVacia c1 = True
    | esVacia c2 = esVacia c1
    | otherwise  = perteneceCola pc1 c2 && contenidaCola rc1 c2
    where pc1 = primero c1
          rc1 = resto c1

-- ---------------------------------------------------------------------
-- Ejercicio 10: Definir la función
```

```haskell
--    prefijoCola :: Eq a => Cola a -> Cola a -> Bool
-- tal que (prefijoCola c1 c2) se verifica si la cola c1 es un prefijo
-- de la cola c2. Por ejemplo,
--    prefijoCola c3 c2 == False
--    prefijoCola c5 c1 == True
-- ---------------------------------------------------------------------

prefijoCola :: Eq a => Cola a -> Cola a -> Bool
prefijoCola c1 c2
    | esVacia c1 = True
    | esVacia c2 = False
    | otherwise  = pc1 == pc2 && prefijoCola rc1 rc2
    where pc1 = primero c1
          rc1 = resto c1
          pc2 = primero c2
          rc2 = resto c2

-- ---------------------------------------------------------------------
-- Ejercicio 11: Definir la función
--    subCola :: Eq a => Cola a -> Cola a -> Bool
-- tal que (subCola c1 c2) se verifica si c1 es una subcola de c2. Por
-- ejemplo,
--    subCola c2 c1 == False
--    subCola c3 c1 == True
-- ---------------------------------------------------------------------

subCola :: Eq a => Cola a -> Cola a -> Bool
subCola c1 c2
    | esVacia c1 = True
    | esVacia c2 = False
    | pc1 == pc2   = prefijoCola rc1 rc2
    | otherwise  = subCola c1 rc2
    where pc1 = primero c1
          rc1 = resto c1
          pc2 = primero c2
          rc2 = resto c2

-- ---------------------------------------------------------------------
-- Ejercicio 12: Definir la función
--    ordenadaCola :: Ord a => Cola a -> Bool
```

```haskell
-- tal que (ordenadaCola c) se verifica si los elementos de la cola c
-- están ordenados en orden creciente. Por ejemplo,
--    ordenadaCola c6 == True
--    ordenadaCola c4 == False
-- ----------------------------------------------------------------------

ordenadaCola :: Ord a => Cola a -> Bool
ordenadaCola c
    | esVacia c = True
    | esVacia rc = True
    | otherwise = pc <= prc && ordenadaCola rc
    where pc  = primero c
          rc  = resto c
          prc = primero q

-- ----------------------------------------------------------------------
-- Ejercicio 13.1: Definir una función
--    lista2Cola :: [a] -> Cola a
-- tal que (lista2Cola xs) es una cola formada por los elementos de xs.
-- Por ejemplo,
--    lista2Cola [1..6] == C [1,2,3,4,5,6]
-- ----------------------------------------------------------------------

lista2Cola :: [a] -> Cola a
lista2Cola xs = foldr inserta vacia (reverse xs)

-- ----------------------------------------------------------------------
-- Ejercicio 13.2: Definir una función
--    cola2Lista :: Cola a -> [a]
-- tal que (cola2Lista c) es la lista formada por los elementos de p.
-- Por ejemplo,
--    cola2Lista c2 == [17,14,11,8,5,2]
-- ----------------------------------------------------------------------

cola2Lista :: Cola a -> [a]
cola2Lista c
    | esVacia c = []
    | otherwise = pc : cola2Lista rc
    where pc = primero c
          rc = resto c
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 13.3. Comprobar con QuickCheck que la función cola2Lista es
-- la inversa de  lista2Cola, y recíprocamente.
-- ---------------------------------------------------------------------

prop_cola2Lista :: Cola Int -> Bool
prop_cola2Lista c =
    lista2Cola (cola2Lista c) == c

-- ghci> quickCheck prop_cola2Lista
-- +++ OK, passed 100 tests.

prop_lista2Cola :: [Int] -> Bool
prop_lista2Cola xs =
    cola2Lista (lista2Cola xs) == xs

-- ghci> quickCheck prop_lista2Cola
-- +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 14: Definir la función
--    maxCola :: Ord a => Cola a -> a
-- tal que (maxCola c) es el mayor de los elementos de la cola c. Por
-- ejemplo,
--    maxCola c4 == 10
-- ---------------------------------------------------------------------

maxCola :: Ord a => Cola a -> a
maxCola c
    | esVacia c = error "cola vacia"
    | esVacia rc = pc
    | otherwise = max pc (maxCola rc)
    where pc  = primero c
          rc  = resto c
          prc = primero q

prop_maxCola c =
    not (esVacia c) ==>
        maxCola c == maximum (cola2Lista c)
```

```haskell
-- ghci> quickCheck prop_maxCola
-- +++ OK, passed 100 tests.

-- ------------------------------------------------------------------
-- Generador de colas                                               --
-- ------------------------------------------------------------------

-- genCola es un generador de colas de enteros. Por ejemplo,
--    ghci> sample genCola
--    C ([],[])
--    C ([],[])
--    C ([],[])
--    C ([],[])
--    C ([7,8,4,3,7],[5,3,3])
--    C ([],[])
--    C ([1],[13])
--    C ([18,28],[12,21,28,28,3,18,14])
--    C ([47],[64,45,7])
--    C ([8],[])
--    C ([42,112,178,175,107],[])
genCola :: (Num a, Arbitrary a) => Gen (Cola a)
genCola = frequency [(1, return vacia),
                     (30, do n <- choose (10,100)
                             xs <- vectorOf n arbitrary
                             return (creaCola xs))]
          where creaCola = foldr inserta vacia

-- El tipo cola es una instancia del arbitrario.
instance (Arbitrary a, Num a) => Arbitrary (Cola a) where
    arbitrary = genCola
```

Relación 26

Aplicaciones de la programación funcional con listas infinitas

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- En esta relación se estudia distintas aplicaciones de la programación
-- funcional que usan listas infinitas
-- * definición alternativa de la sucesión de Hamming estudiada en el
--   tema 11,
-- * propiedades de la sucesión de Hamming,
-- * problemas 10 y 12 del proyecto Euler y
-- * numero de pares de naturales en un círculo.

-- ---------------------------------------------------------------------
-- Importación de librerías                                            --
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    divisoresEn :: Integer -> [Integer] -> Bool
-- tal que (divisoresEn x ys) se verifica si x puede expresarse como un
-- producto de potencias de elementos de ys. Por ejemplo,
--    divisoresEn 12 [2,3,5]  ==  True
```

```haskell
--    divisoresEn 14 [2,3,5]  ==  False
-- ---------------------------------------------------------------------

divisoresEn :: Integer -> [Integer] -> Bool
divisoresEn 1 _                   = True
divisoresEn x []                  = False
divisoresEn x (y:ys) | mod x y == 0 = divisoresEn (div x y) (y:ys)
                     | otherwise    = divisoresEn x ys

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Los números de Hamming forman una sucesión
-- estrictamente creciente de números que cumplen las siguientes
-- condiciones:
--    1. El número 1 está en la sucesión.
--    2. Si x está en la sucesión, entonces 2x, 3x y 5x también están.
--    3. Ningún otro número está en la sucesión.
-- Definir, usando divisoresEn, la constante
--    hamming :: [Integer]
-- tal que hamming es la sucesión de Hamming. Por ejemplo,
--    take 12 hamming  ==  [1,2,3,4,5,6,8,9,10,12,15,16]
-- ---------------------------------------------------------------------

hamming :: [Integer]
hamming = [x | x <- [1..], divisoresEn x [2,3,5]]

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    cantidadHammingMenores :: Integer -> Int
-- tal que (cantidadHammingMenores x) es la cantidad de números de
-- Hamming menores que x. Por ejemplo,
--    cantidadHammingMenores 6  ==  5
--    cantidadHammingMenores 7  ==  6
--    cantidadHammingMenores 8  ==  6
-- ---------------------------------------------------------------------

cantidadHammingMenores :: Integer -> Int
cantidadHammingMenores x = length (takeWhile (<x) hamming)

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Definir la función
```

```haskell
--    siguienteHamming :: Integer -> Integer
-- tal que (siguienteHamming x) es el menor número de la sucesión de
-- Hamming mayor que x. Por ejemplo,
--    siguienteHamming 6  ==  8
--    siguienteHamming 21  ==  24
-- ---------------------------------------------------------------------

siguienteHamming :: Integer -> Integer
siguienteHamming x = head (dropWhile (<=x) hamming)

-- ---------------------------------------------------------------------
-- Ejercicio 1.5. Definir la función
--    huecoHamming :: Integer -> [(Integer,Integer)]
-- tal que (huecoHamming n) es la lista de pares de números consecutivos
-- en la sucesión de Hamming cuya distancia es mayor o igual que n. Por
-- ejemplo,
--    take 4 (huecoHamming 2)   ==  [(12,15),(20,24),(27,30),(32,36)]
--    take 3 (huecoHamming 2)   ==  [(12,15),(20,24),(27,30)]
--    take 2 (huecoHamming 3)   ==  [(20,24),(32,36)]
--    head (huecoHamming 10)    ==  (108,120)
--    head (huecoHamming 1000)  ==  (34992,36000)
-- ---------------------------------------------------------------------

huecoHamming :: Integer -> [(Integer,Integer)]
huecoHamming n = [(x,y) | x <- hamming,
                          let y = siguienteHamming x,
                          y-x > n]

-- ---------------------------------------------------------------------
-- Ejercicio 1.6. Comprobar con QuickCheck que para todo n, existen
-- pares de números consecutivos en la sucesión de Hamming cuya
-- distancia es mayor o igual que n.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_Hamming :: Integer -> Bool
prop_Hamming n = huecoHamming n' /= []
                 where n' = abs n

-- La comprobación es
```

```
--    *Main> quickCheck prop_Hamming
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2. (Problema 10 del Proyecto Euler)
-- Definir la función
--    sumaPrimoMenores :: Integer -> Integer
-- tal que (sumaPrimoMenores n) es la suma de los primos menores que
-- n. Por ejemplo,
--    sumaPrimoMenores 10  ==  17
-- ---------------------------------------------------------------------

-- La definición es
sumaPrimoMenores :: Integer -> Integer
sumaPrimoMenores n = sumaMenores n primos 0
   where sumaMenores n (x:xs) a | n <= x    = a
                                | otherwise = sumaMenores n xs (a+x)

-- primos es la lista de los número primos obtenida mediante la criba de
-- Erastótenes. Por ejemplo,
--    primos  =>  [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,...
primos :: [Integer]
primos = criba [2..]
         where criba (p:ps) = p : criba [n | n<-ps, mod n p /= 0]

-- ---------------------------------------------------------------------
-- Ejercicio 3. (Problema 12 del Proyecto Euler)
-- La sucesión de los números triangulares se obtiene sumando los
-- números naturales. Así, el 7º número triangular es
--    1 + 2 + 3 + 4 + 5 + 6 + 7 = 28.
-- Los primeros 10 números triangulares son
--    1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...
-- Los divisores de los primeros 7 números triangulares son:
--     1: 1
--     3: 1,3
--     6: 1,2,3,6
--    10: 1,2,5,10
--    15: 1,3,5,15
--    21: 1,3,7,21
--    28: 1,2,4,7,14,28
```

```haskell
-- Como se puede observar, 28 es el menor número triangular con más de 5
-- divisores.
--
-- Definir la función
--    euler12 :: Int -> Integer
-- tal que (euler12 n) es el menor número triangular con más de n
-- divisores. Por ejemplo,
--    euler12 5  ==  28
-- ---------------------------------------------------------------------

euler12 :: Int -> Integer
euler12 n = head [x | x <- triangulares, nDivisores x > n]

-- triangulares es la lista de los números triangulares
--    take 10 triangulares  =>  [1,3,6,10,15,21,28,36,45,55]
triangulares :: [Integer]
triangulares = 1:[x+y | (x,y) <- zip [2..] triangulares]

-- Otra definición de triangulares es
triangulares' :: [Integer]
triangulares' = scanl (+) 1 [2..]

-- (divisores n) es la lista de los divisores de n. Por ejemplo,
--    divisores 28  ==  [1,2,4,7,14,28]
divisores :: Integer -> [Integer]
divisores x = [y | y <- [1..x], mod x y == 0]

-- (nDivisores n) es el número de los divisores de n. Por ejemplo,
--    nDivisores 28  ==  6
nDivisores :: Integer -> Int
nDivisores x = length (divisores x)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    circulo :: Int -> Int
-- tal que (circulo n) es el la cantidad de pares de números naturales
-- (x,y) que se encuentran dentro del círculo de radio n. Por ejemplo,
--    circulo 3  ==  9
--    circulo 4  ==  15
--    circulo 5  ==  22
```

```haskell
-- ---------------------------------------------------------------------

circulo :: Int -> Int
circulo n = length [(x,y) | x <- [0..n], y <- [0..n], x^2+y^2 < n^2]

-- La eficiencia puede mejorarse con
circulo' :: Int -> Int
circulo' n = length [(x,y) | x <- [0..m], y <- [0..m], x^2+y^2 < n^2]
    where m = raizCuadradaEntera n

-- (raizCuadradaEntera n) es la parte entera de la raíz cuadrada de
-- n. Por ejemplo,
--    raizCuadradaEntera 17  ==  4
raizCuadradaEntera :: Int -> Int
raizCuadradaEntera n = truncate (sqrt (fromIntegral n))
```

Relación 27

El TAD de los montículos

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación de ejercicios es definir funciones sobre
-- el TAD de las montículos, utilizando las implementaciones estudiadas
-- en el tema 20 que se pueden descargar desde
--    http://www.cs.us.es/~jalonso/cursos/i1m/codigos.zip
--
-- Las transparencias del tema 20 se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-20.pdf

-- ---------------------------------------------------------------------
-- Importación de librerías                                            --
-- ---------------------------------------------------------------------

{-# LANGUAGE FlexibleInstances #-}

import Monticulo
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejemplos                                                            --
-- ---------------------------------------------------------------------

-- Para los ejemplos se usarán los siguientes montículos.
m1, m2, m3 :: Monticulo Int
```

```haskell
m1 = foldr inserta vacio [6,1,4,8]
m2 = foldr inserta vacio [7,5]
m3 = foldr inserta vacio [6,1,4,8,7,5]

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    numeroDeNodos :: Ord a => Monticulo a -> Int
-- tal que (numeroDeNodos m) es el número de nodos del montículo m. Por
-- ejemplo,
--    numeroDeNodos m1  ==  4
-- ---------------------------------------------------------------------

numeroDeNodos :: Ord a => Monticulo a -> Int
numeroDeNodos m
    | esVacio m = 0
    | otherwise = 1 + numeroDeNodos (resto m)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    filtra :: Ord a => (a -> Bool) -> Monticulo a -> Monticulo a
-- tal que (filtra p m) es el montículo con los nodos del montículo m
-- que cumplen la propiedad p. Por ejemplo,
--    ghci> m1
--    M 1 2 (M 4 1 (M 8 1 Vacio Vacio) Vacio) (M 6 1 Vacio Vacio)
--    ghci> filtra even m1
--    M 4 1 (M 6 1 (M 8 1 Vacio Vacio) Vacio) Vacio
--    ghci> filtra odd m1
--    M 1 1 Vacio Vacio
-- ---------------------------------------------------------------------

filtra :: Ord a => (a -> Bool) -> Monticulo a -> Monticulo a
filtra p m
    | esVacio m = vacio
    | p mm      = inserta mm (filtra p rm)
    | otherwise = filtra p rm
    where mm = menor m
          rm = resto m

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
```

```
--    menores :: Ord a => Int -> Monticulo a -> [a]
-- tal que (menores n m) es la lista de los n menores elementos del
-- montículo m. Por ejemplo,
--    ghci> m1
--    M 1 2 (M 4 1 (M 8 1 Vacio Vacio) Vacio) (M 6 1 Vacio Vacio)
--    ghci> menores 3 m1
--    [1,4,6]
-- ------------------------------------------------------------------

menores :: Ord a => Int -> Monticulo a -> [a]
menores 0 m  = []
menores (n+1) m | esVacio m = []
                | otherwise = menor m : menores n (resto m)

-- ------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    restantes :: Ord a => Int -> Monticulo a -> Monticulo a
-- tal que (restantes n m) es el montículo obtenido rliminando los n
-- menores elementos del montículo m. Por ejemplo,
--    ghci> m1
--    M 1 2 (M 4 1 (M 8 1 Vacio Vacio) Vacio) (M 6 1 Vacio Vacio)
--    ghci> restantes 3 m1
--    M 8 1 Vacio Vacio
--    ghci> restantes 2 m1
--    M 6 1 (M 8 1 Vacio Vacio) Vacio
-- ------------------------------------------------------------------

restantes :: Ord a => Int -> Monticulo a -> Monticulo a
restantes 0 m  = m
restantes (n+1) m | esVacio m = vacio
                  | otherwise = restantes n (resto m)

-- ------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    lista2Monticulo :: Ord a => [a] -> Monticulo a
-- tal que (lista2Monticulo xs) es el montículo cuyos nodos son los
-- elementos de la lista xs. Por ejemplo,
--    ghci> lista2Monticulo [2,5,3,7]
--    M 2 1 (M 3 2 (M 7 1 Vacio Vacio) (M 5 1 Vacio Vacio)) Vacio
-- ------------------------------------------------------------------
```

```haskell
lista2Monticulo :: Ord a => [a] -> Monticulo a
lista2Monticulo = foldr inserta vacio

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    monticulo2Lista :: Ord a => Monticulo a -> [a]
-- tal que (monticulo2Lista m) es la lista ordenada de los nodos del
-- montículo m. Por ejemplo,
--    ghci> m1
--    M 1 2 (M 4 1 (M 8 1 Vacio Vacio) Vacio) (M 6 1 Vacio Vacio)
--    ghci> monticulo2Lista m1
--    [1,4,6,8]
-- ---------------------------------------------------------------------

monticulo2Lista :: Ord a => Monticulo a -> [a]
monticulo2Lista m
    | esVacio m = []
    | otherwise = menor m : monticulo2Lista (resto m)

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    ordenada :: Ord a => [a] -> Bool
-- tal que (ordenada xs) se verifica si xs es una lista ordenada de
-- forma creciente. Por ejemplo,
--    ordenada [3,5,9]  ==  True
--    ordenada [3,5,4]  ==  False
--    ordenada [7,5,4]  ==  False
-- ---------------------------------------------------------------------

ordenada :: Ord a => [a] -> Bool
ordenada (x:y:zs) = x <= y && ordenada (y:zs)
ordenada _        = True

-- ---------------------------------------------------------------------
-- Ejercicio 8. Comprobar con QuickCheck que para todo montículo m,
-- (monticulo2Lista m) es una lista ordenada creciente.
-- ---------------------------------------------------------------------

-- La propiedad es
```

```haskell
prop_monticulo2Lista_ordenada :: Monticulo Int -> Bool
prop_monticulo2Lista_ordenada m =
    ordenada (monticulo2Lista m)

-- La comprobación es
--    ghci> quickCheck prop_monticulo2Lista_ordenada
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 10. Usando monticulo2Lista y lista2Monticulo, definir la
-- función
--    ordena :: Ord a => [a] -> [a]
-- tal que (ordena xs) es la lista obtenida ordenando de forma creciente
-- los elementos de xs. Por ejemplo,
--    ordena [7,5,3,6,5]  ==  [3,5,5,6,7]
-- ---------------------------------------------------------------------

ordena :: Ord a => [a] -> [a]
ordena  = monticulo2Lista . lista2Monticulo

-- ---------------------------------------------------------------------
-- Ejercicio 11. Comprobar con QuickCheck que para toda lista xs,
-- (ordena xs) es una lista ordenada creciente.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_ordena_ordenada :: [Int] -> Bool
prop_ordena_ordenada xs =
    ordenada (ordena xs)

-- La comprobación es
--    ghci> quickCheck prop_ordena_ordenada
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    borra :: Eq a => a -> [a] -> [a]
-- tal que (borra x xs) es la lista obtenida borrando una ocurrencia de
-- x en la lista xs. Por ejemplo,
--    borra 1 [1,2,1]  ==  [2,1]
```

```
--    borra 3 [1,2,1]  ==  [1,2,1]
-- ---------------------------------------------------------------------

borra :: Eq a => a -> [a] -> [a]
borra x []                  = []
borra x (y:ys) | x == y     = ys
               | otherwise = y : borra x ys

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función esPermutación tal que
-- (esPermutación xs ys) se verifique si xs es una permutación de
-- ys. Por ejemplo,
--    esPermutación [1,2,1] [2,1,1]  ==  True
--    esPermutación [1,2,1] [1,2,2]  ==  False
-- ---------------------------------------------------------------------

esPermutacion :: Eq a => [a] -> [a] -> Bool
esPermutacion []     []     = True
esPermutacion []     (y:ys) = False
esPermutacion (x:xs) ys     = elem x ys && esPermutacion xs (borra x ys)

-- ---------------------------------------------------------------------
-- Ejercicio 15. Comprobar con QuickCheck que para toda lista xs,
-- (ordena xs) es una permutación de xs.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_ordena_permutacion :: [Int] -> Bool
prop_ordena_permutacion xs =
    esPermutacion (ordena xs) xs

-- La comprobación es
--    ghci> quickCheck prop_ordena_permutacion
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Generador de montículos                                           --
-- ---------------------------------------------------------------------

-- genMonticulo es un generador de montículos. Por ejemplo,
```

```
--    ghci> sample genMonticulo
--    VacioM
--    M (-1) 1 (M 1 1 VacioM VacioM) VacioM
--    ...
genMonticulo :: Gen (Monticulo Int)
genMonticulo = do xs <- listOf arbitrary
                  return (foldr inserta vacio xs)

-- Montículo es una instancia de la clase arbitraria.
instance Arbitrary (Monticulo Int) where
    arbitrary = genMonticulo
```

Relación 28

Vectores y matrices

```haskell
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación es hacer ejercicios sobre vectores y
-- matrices con el tipo de tablas de las tablas, definido en el módulo
-- Data.Array y explicado en el tema 18 cuyas transparencias se
-- encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-18t.pdf
-- Además, en algunos ejemplos de usan matrices con números racionales.
-- En Haskell, el número racional x/y se representa por x%y. El TAD de
-- los números racionales está definido en el módulo Data.Ratio.

-- ---------------------------------------------------------------------
-- Importación de librerías                                            --
-- ---------------------------------------------------------------------

import Data.Array
import Data.Ratio

-- ---------------------------------------------------------------------
-- Tipos de los vectores y de las matrices                             --
-- ---------------------------------------------------------------------

-- Los vectores son tablas cuyos índices son números naturales.
type Vector a = Array Int a
```

```haskell
-- Las matrices son tablas cuyos índices son pares de números
-- naturales.
type Matriz a = Array (Int,Int) a

-- ---------------------------------------------------------------------
-- Operaciones básicas con matrices                                    --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    listaVector :: Num a => [a] -> Vector a
-- tal que (listaVector xs) es el vector correspondiente a la lista
-- xs. Por ejemplo,
--    ghci> listaVector [3,2,5]
--    array (1,3) [(1,3),(2,2),(3,5)]
-- ---------------------------------------------------------------------

listaVector :: Num a => [a] -> Vector a
listaVector xs = listArray (1,n) xs
    where n = length xs

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    listaMatriz :: Num a => [[a]] -> Matriz a
-- tal que (listaMatriz xss) es la matriz cuyas filas son los elementos
-- de xss. Por ejemplo,
--    ghci> listaMatriz [[1,3,5],[2,4,7]]
--    array ((1,1),(2,3)) [((1,1),1),((1,2),3),((1,3),5),
--                         ((2,1),2),((2,2),4),((2,3),7)]
-- ---------------------------------------------------------------------

listaMatriz :: Num a => [[a]] -> Matriz a
listaMatriz xss = listArray ((1,1),(m,n)) (concat xss)
    where m = length xss
          n = length (head xss)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    numFilas :: Num a => Matriz a -> Int
-- tal que (numFilas m) es el número de filas de la matriz m. Por
```

```haskell
-- ejemplo,
--    numFilas (listaMatriz [[1,3,5],[2,4,7]])  ==  2
-- ---------------------------------------------------------------------

numFilas :: Num a => Matriz a -> Int
numFilas = fst . snd . bounds

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    numColumnas :: Num a => Matriz a -> Int
-- tal que (numColumnas m) es el número de columnas de la matriz
-- m. Por ejemplo,
--    numColumnas (listaMatriz [[1,3,5],[2,4,7]])  ==  3
-- ---------------------------------------------------------------------

numColumnas:: Num a => Matriz a -> Int
numColumnas = snd . snd . bounds

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    dimension :: Num a => Matriz a -> (Int,Int)
-- tal que (dimension m) es el número de columnas de la matriz m. Por
-- ejemplo,
--    dimension (listaMatriz [[1,3,5],[2,4,7]])  ==  (2,3)
-- ---------------------------------------------------------------------

dimension :: Num a => Matriz a -> (Int,Int)
dimension p = (numFilas p, numColumnas p)

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    separa :: Int -> [a] -> [[a]]
-- tal que (separa n xs) es la lista obtenida separando los elementos de
-- xs en grupos de n elementos (salvo el último que puede tener menos de
-- n elementos). Por ejemplo,
--    separa 3 [1..11]  ==  [[1,2,3],[4,5,6],[7,8,9],[10,11]]
-- ---------------------------------------------------------------------

separa :: Int -> [a] -> [[a]]
separa _ [] = []
```

```
separa n xs = take n xs : separa n (drop n xs)

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    matrizLista :: Num a => Matriz a -> [[a]]
-- tal que (matrizLista x) es la lista de las filas de la matriz x. Por
-- ejemplo,
--    ghci> let m = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> m
--    array ((1,1),(2,3)) [((1,1),5),((1,2),1),((1,3),0),
--                         ((2,1),3),((2,2),2),((2,3),6)]
--    ghci> matrizLista m
--    [[5,1,0],[3,2,6]]
-- ---------------------------------------------------------------------

matrizLista :: Num a => Matriz a -> [[a]]
matrizLista p = separa (numColumnas p) (elems p)

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    vectorLista :: Num a => Vector a -> [a]
-- tal que (vectorLista x) es la lista de los elementos del vector
-- v. Por ejemplo,
--    ghci> let v = listaVector [3,2,5]
--    ghci> v
--    array (1,3) [(1,3),(2,2),(3,5)]
--    ghci> vectorLista v
--    [3,2,5]
-- ---------------------------------------------------------------------

vectorLista :: Num a => Vector a -> [a]
vectorLista = elems

-- ---------------------------------------------------------------------
-- Suma de matrices                                                    --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    sumaMatrices:: Num a => Matriz a -> Matriz a -> Matriz a
```

```haskell
-- tal que (sumaMatrices x y) es la suma de las matrices x e y. Por
-- ejemplo,
--    ghci> let m1 = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> let m2 = listaMatriz [[4,6,3],[1,5,2]]
--    ghci> matrizLista (sumaMatrices m1 m2)
--    [[9,7,3],[4,7,8]]
-- ---------------------------------------------------------------------

sumaMatrices:: Num a => Matriz a -> Matriz a -> Matriz a
sumaMatrices p q =
    array ((1,1),(m,n)) [((i,j),p!(i,j)+q!(i,j)) |
                         i <- [1..m], j <- [1..n]]
    where (m,n) = dimension p

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    filaMat :: Num a => Int -> Matriz a -> Vector a
-- tal que (filaMat i p) es el vector correspondiente a la fila i-ésima
-- de la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,5,7]]
--    ghci> filaMat 2 p
--    array (1,3) [(1,3),(2,2),(3,6)]
--    ghci> vectorLista (filaMat 2 p)
--    [3,2,6]
-- ---------------------------------------------------------------------

filaMat :: Num a => Int -> Matriz a -> Vector a
filaMat i p = array (1,n) [(j,p!(i,j)) | j <- [1..n]]
    where n = numColumnas p

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--    columnaMat :: Num a => Int -> Matriz a -> Vector a
-- tal que (columnaMat j p) es el vector correspondiente a la columna
-- j-ésima de la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,5,7]]
--    ghci> columnaMat 2 p
--    array (1,3) [(1,1),(2,2),(3,5)]
--    ghci> vectorLista (columnaMat 2 p)
--    [1,2,5]
```

```haskell
-- ---------------------------------------------------------------------

columnaMat :: Num a => Int -> Matriz a -> Vector a
columnaMat j p = array (1,m) [(i,p!(i,j)) | i <- [1..m]]
    where m = numFilas p

-- ---------------------------------------------------------------------
-- Producto de matrices                                                --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    prodEscalar :: Num a => Vector a -> Vector a -> a
-- tal que (prodEscalar v1 v2) es el producto escalar de los vectores v1
-- y v2. Por ejemplo,
--    ghci> let v = listaVector [3,1,10]
--    ghci> prodEscalar v v
--    110
-- ---------------------------------------------------------------------

prodEscalar :: Num a => Vector a -> Vector a -> a
prodEscalar v1 v2 =
    sum [i*j | (i,j) <- zip (elems v1) (elems v2)]

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la función
--    prodMatrices:: Num a => Matriz a -> Matriz a -> Matriz a
-- tal que (prodMatrices p q) es el producto de las matrices p y q. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[3,1],[2,4]]
--    ghci> prodMatrices p p
--    array ((1,1),(2,2)) [((1,1),11),((1,2),7),((2,1),14),((2,2),18)]
--    ghci> matrizLista (prodMatrices p p)
--    [[11,7],[14,18]]
--    ghci> let q = listaMatriz [[7],[5]]
--    ghci> prodMatrices p q
--    array ((1,1),(2,1)) [((1,1),26),((2,1),34)]
--    ghci> matrizLista (prodMatrices p q)
--    [[26],[34]]
-- ---------------------------------------------------------------------
```

```haskell
prodMatrices:: Num a => Matriz a -> Matriz a -> Matriz a
prodMatrices p q =
    array ((1,1),(m,n))
          [((i,j), prodEscalar (filaMat i p) (columnaMat j q)) |
           i <- [1..m], j <- [1..n]]
    where m = numFilas p
          n = numColumnas q

-- ---------------------------------------------------------------------
-- Traspuestas y simétricas                                           --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función
--    traspuesta :: Num a => Matriz a -> Matriz a
-- tal que (traspuesta p) es la traspuesta de la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> traspuesta p
--    array ((1,1),(3,2)) [((1,1),5),((1,2),3),
--                         ((2,1),1),((2,2),2),
--                         ((3,1),0),((3,2),6)]
--    ghci> matrizLista (traspuesta p)
--    [[5,3],[1,2],[0,6]]
-- ---------------------------------------------------------------------

traspuesta :: Num a => Matriz a -> Matriz a
traspuesta p =
    array ((1,1),(n,m))
          [((i,j), p!(j,i)) | i <- [1..n], j <- [1..m]]
    where (m,n) = dimension p

-- ---------------------------------------------------------------------
-- Ejercicio 15. Definir la función
--    esCuadrada :: Num a => Matriz a -> Bool
-- tal que (esCuadrada p) se verifica si la matriz p es cuadrada. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> esCuadrada p
--    False
```

```
--    ghci> let q = listaMatriz [[5,1],[3,2]]
--    ghci> esCuadrada q
--    True
-- ---------------------------------------------------------------------

esCuadrada :: Num a => Matriz a -> Bool
esCuadrada x = numFilas x == numColumnas x

-- ---------------------------------------------------------------------
-- Ejercicio 16. Definir la función
--    esSimetrica :: Num a => Matriz a -> Bool
-- tal que (esSimetrica p) se verifica si la matriz p es simétrica. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[5,1,3],[1,4,7],[3,7,2]]
--    ghci> esSimetrica p
--    True
--    ghci> let q = listaMatriz [[5,1,3],[1,4,7],[3,4,2]]
--    ghci> esSimetrica q
--    False
-- ---------------------------------------------------------------------

esSimetrica :: Num a => Matriz a -> Bool
esSimetrica x = x == traspuesta x

-- ---------------------------------------------------------------------
-- Diagonales de una matriz                                            --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 17. Definir la función
--    diagonalPral :: Num a => Matriz a -> Vector a
-- tal que (diagonalPral p) es la diagonal principal de la matriz p. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> diagonalPral p
--    array (1,2) [(1,5),(2,2)]
--    ghci> vectorLista (diagonalPral p)
--    [5,2]
-- ---------------------------------------------------------------------
```

```haskell
diagonalPral :: Num a => Matriz a -> Vector a
diagonalPral p = array (1,n) [(i,p!(i,i)) | i <- [1..n]]
    where n = min (numFilas p) (numColumnas p)

-- ---------------------------------------------------------------------
-- Ejercicio 18. Definir la función
--    diagonalSec :: Num a => Matriz a -> Vector a
-- tal que (diagonalSec p) es la diagonal secundaria de la matriz p. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6]]
--    ghci> diagonalSec p
--    array (1,2) [(1,1),(2,3)]
--    ghci> vectorLista (diagonalPral p)
--    [5,2]
-- ---------------------------------------------------------------------

diagonalSec :: Num a => Matriz a -> Vector a
diagonalSec p = array (1,n) [(i,p!(i,m+1-i)) | i <- [1..n]]
    where n = min (numFilas p) (numColumnas p)
          m = numFilas p

-- ---------------------------------------------------------------------
-- Submatrices                                                         --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 19. Definir la función
--    submatriz :: Num a => Int -> Int -> Matriz a -> Matriz a
-- tal que (submatriz i j p) es la matriz obtenida a partir de la p
-- eliminando la fila i y la columna j. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> submatriz 2 3 p
--    array ((1,1),(2,2)) [((1,1),5),((1,2),1),((2,1),4),((2,2),6)]
--    ghci> matrizLista (submatriz 2 3 p)
--    [[5,1],[4,6]]
-- ---------------------------------------------------------------------

submatriz :: Num a => Int -> Int -> Matriz a -> Matriz a
submatriz i j p =
    array ((1,1), (m-1,n -1))
```

```
          [((k,l), p ! f k l) | k <- [1..m-1], l <- [1.. n-1]]
    where (m,n) = dimension p
          f k l | k < i  && l < j  = (k,l)
                | k >= i && l < j  = (k+1,l)
                | k < i  && l >= j = (k,l+1)
                | otherwise        = (k+1,l+1)

-- ---------------------------------------------------------------------
-- Transformaciones elementales                                        --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 20. Definir la función
--    intercambiaFilas :: Num a => Int -> Int -> Matriz a -> Matriz a
-- tal que (intercambiaFilas k l p) es la matriz obtenida intercambiando
-- las filas k y l de la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> intercambiaFilas 1 3 p
--    array ((1,1),(3,3)) [((1,1),4),((1,2),6),((1,3),9),
--                         ((2,1),3),((2,2),2),((2,3),6),
--                         ((3,1),5),((3,2),1),((3,3),0)]
--    ghci> matrizLista (intercambiaFilas 1 3 p)
--    [[4,6,9],[3,2,6],[5,1,0]]
-- ---------------------------------------------------------------------

intercambiaFilas :: Num a => Int -> Int -> Matriz a -> Matriz a
intercambiaFilas k l p =
    array ((1,1), (m,n))
          [((i,j), p! f i j) | i <- [1..m], j <- [1..n]]
    where (m,n) = dimension p
          f i j | i == k    = (l,j)
                | i == l    = (k,j)
                | otherwise = (i,j)

-- ---------------------------------------------------------------------
-- Ejercicio 21. Definir la función
--    intercambiaColumnas :: Num a => Int -> Int -> Matriz a -> Matriz a
-- tal que (intercambiaColumnas k l p) es la matriz obtenida
-- intercambiando las columnas k y l de la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
```

```
--    ghci> matrizLista (intercambiaColumnas 1 3 p)
--    [[0,1,5],[6,2,3],[9,6,4]]
-- ---------------------------------------------------------------------

intercambiaColumnas :: Num a => Int -> Int -> Matriz a -> Matriz a
intercambiaColumnas k l p =
    array ((1,1), (m,n))
          [((i,j), p ! f i j) | i <- [1..m], j <- [1..n]]
    where (m,n) = dimension p
          f i j | j == k    = (i,l)
                | j == l    = (i,k)
                | otherwise = (i,j)

-- ---------------------------------------------------------------------
-- Ejercicio 22. Definir la función
--    multFilaPor :: Num a => Int -> a -> Matriz a -> Matriz a
-- tal que (multFilaPor k x p) es a matriz obtenida multiplicando la
-- fila k de la matriz p por el número x. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> matrizLista (multFilaPor 2 3 p)
--    [[5,1,0],[9,6,18],[4,6,9]]
-- ---------------------------------------------------------------------

multFilaPor :: Num a => Int -> a -> Matriz a -> Matriz a
multFilaPor k x p =
    array ((1,1), (m,n))
          [((i,j), f i j)  | i <- [1..m], j <- [1..n]]
    where (m,n) = dimension p
          f i j | i == k    = x*(p!(i,j))
                | otherwise = p!(i,j)

-- ---------------------------------------------------------------------
-- Ejercicio 23. Definir la función
--    sumaFilaFila :: Num a => Int -> Int -> Matriz a -> Matriz a
-- tal que (sumaFilaFila k l p) es la matriz obtenida sumando la fila l
-- a la fila k d la matriz p. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> matrizLista (sumaFilaFila 2 3 p)
--    [[5,1,0],[7,8,15],[4,6,9]]
-- ---------------------------------------------------------------------
```

```haskell
sumaFilaFila :: Num a => Int -> Int -> Matriz a -> Matriz a
sumaFilaFila k l p =
    array ((1,1), (m,n))
          [((i,j), f i j) | i <- [1..m], j <- [1..n]]
    where (m,n) = dimension p
          f i j | i == k    = p!(i,j) + p!(l,j)
                | otherwise = p!(i,j)

-- ---------------------------------------------------------------------
-- Ejercicio 24. Definir la función
--    sumaFilaPor :: Num a => Int -> Int -> a -> Matriz a -> Matriz a
-- tal que (sumaFilaPor k l x p) es la matriz obtenida sumando a la fila
-- k de la matriz p la fila l multiplicada por x. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> matrizLista (sumaFilaPor 2 3 10 p)
--    [[5,1,0],[43,62,96],[4,6,9]]
-- ---------------------------------------------------------------------

sumaFilaPor :: Num a => Int -> Int -> a -> Matriz a -> Matriz a
sumaFilaPor k l x p =
    array ((1,1), (m,n))
          [((i,j), f i j) | i <- [1..m], j <- [1..n]]
    where (m,n) = dimension p
          f i j | i == k    = p!(i,j) + x*p!(l,j)
                | otherwise = p!(i,j)

-- ---------------------------------------------------------------------
-- Triangularización de matrices                                       --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 25. Definir la función
--    buscaIndiceDesde :: Num a => Matriz a -> Int -> Int -> Maybe Int
-- tal que (buscaIndiceDesde p j i) es el menor índice k, mayor o igual
-- que i, tal que el elemento de la matriz p en la posición (k,j) es no
-- nulo. Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> buscaIndiceDesde p 3 2
--    Just 2
```

```
--    ghci> let q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
--    ghci> buscaIndiceDesde q 3 2
--    Nothing
-- ---------------------------------------------------------------------

buscaIndiceDesde :: Num a => Matriz a -> Int -> Int -> Maybe Int
buscaIndiceDesde p j i
    | null xs   = Nothing
    | otherwise = Just (head xs)
    where xs = [k | ((k,j'),y) <- assocs p, j == j', y /= 0, k>=i]

-- ---------------------------------------------------------------------
-- Ejercicio 26. Definir la función
--    buscaPivoteDesde :: Num a => Matriz a -> Int -> Int -> Maybe a
-- tal que (buscaPivoteDesde p j i) es el elemento de la matriz p en la
-- posición (k,j) donde k es (buscaIndiceDesde p j i). Por ejemplo,
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
--    ghci> buscaPivoteDesde p 3 2
--    Just 6
--    ghci> let q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
--    ghci> buscaPivoteDesde q 3 2
--    Nothing
-- ---------------------------------------------------------------------

buscaPivoteDesde :: Num a => Matriz a -> Int -> Int -> Maybe a
buscaPivoteDesde p j i
    | null xs   = Nothing
    | otherwise = Just (head xs)
    where xs = [y | ((k,j'),y) <- assocs p, j == j', y /= 0, k>=i]

-- ---------------------------------------------------------------------
-- Ejercicio 27. Definir la función
--    anuladaColumnaDesde :: Num a => Int -> Int -> Matriz a -> Bool
-- tal que (anuladaColumnaDesde j i p) se verifica si todos los
-- elementos de la columna j de la matriz p desde i+1 en adelante son
-- nulos. Por ejemplo,
--    ghci> let q = listaMatriz [[5,1,1],[3,2,0],[4,6,0]]
--    ghci> anuladaColumnaDesde q 3 2
--    True
--    ghci> let p = listaMatriz [[5,1,0],[3,2,6],[4,6,9]]
```

```
--    ghci> anuladaColumnaDesde p 3 2
--    False
-- ---------------------------------------------------------------------

anuladaColumnaDesde :: Num a => Matriz a -> Int -> Int -> Bool
anuladaColumnaDesde p j i =
    buscaIndiceDesde p j (i+1) == Nothing

-- ---------------------------------------------------------------------
-- Ejercicio 28. Definir la función
--    anulaEltoColumnaDesde :: Num a => Matriz a -> Int -> Int -> Matriz a
-- tal que (anulaEltoColumnaDesde p j i) es la matriz obtenida a partir
-- de p anulando el primer elemento de la columna j por debajo de la
-- fila i usando el elemento de la posición (i,j). Por ejemplo,
--    ghci> let p = listaMatriz [[2,3,1],[5,0,5],[8,6,9]] :: Matriz Double
--    ghci> matrizLista (anulaEltoColumnaDesde p 2 1)
--    [[2.0,3.0,1.0],[5.0,0.0,5.0],[4.0,0.0,7.0]]
-- ---------------------------------------------------------------------

anulaEltoColumnaDesde :: Fractional a => Matriz a -> Int -> Int -> Matriz a
anulaEltoColumnaDesde p j i =
    sumaFilaPor l i (-(p!(l,j)/a)) p
    where Just l = buscaIndiceDesde p j (i+1)
          a      = p!(i,j)

-- ---------------------------------------------------------------------
-- Ejercicio 29. Definir la función
--    anulaColumnaDesde :: Fractional a => Matriz a -> Int -> Int -> Matriz a
-- tal que (anulaColumnaDesde p j i) es la matriz obtenida anulando
-- todos los elementos de la columna j de la matriz p por debajo del la
-- posición (i,j) (se supone que el elemnto p_(i,j) es no nulo). Por
-- ejemplo,
--    ghci> let p = listaMatriz [[2,2,1],[5,4,5],[10,8,9]] :: Matriz Double
--    ghci> matrizLista (anulaColumnaDesde p 2 1)
--    [[2.0,2.0,1.0],[1.0,0.0,3.0],[2.0,0.0,5.0]]
--    ghci> let p = listaMatriz [[4,5],[2,7%2],[6,10]]
--    ghci> matrizLista (anulaColumnaDesde p 1 1)
--    [[4 % 1,5 % 1],[0 % 1,1 % 1],[0 % 1,5 % 2]]
-- ---------------------------------------------------------------------
```

```
anulaColumnaDesde :: Fractional a => Matriz a -> Int -> Int -> Matriz a
anulaColumnaDesde p j i
    | anuladaColumnaDesde p j i = p
    | otherwise = anulaColumnaDesde (anulaEltoColumnaDesde p j i) j i

-- ---------------------------------------------------------------------
-- Algoritmo de Gauss para triangularizar matrices                     --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 30. Definir la función
--    elementosNoNulosColDesde :: Num a => Matriz a -> Int -> Int -> [a]
-- tal que (elementosNoNulosColDesde p j i) es la lista de los elementos
-- no nulos de la columna j a partir de la fila i. Por ejemplo,
--    ghci> let p = listaMatriz [[3,2],[5,1],[0,4]]
--    ghci> elementosNoNulosColDesde p 1 2
--    [5]
-- ---------------------------------------------------------------------

elementosNoNulosColDesde :: Num a => Matriz a -> Int -> Int -> [a]
elementosNoNulosColDesde p j i =
    [x | ((k,j'),x) <- assocs p, x /= 0, j' == j, k >= i]

-- ---------------------------------------------------------------------
-- Ejercicio 31. Definir la función
--    existeColNoNulaDesde :: Num a => Matriz a -> Int -> Int -> Bool
-- tal que (existeColNoNulaDesde p j i) se verifica si la matriz p tiene
-- una columna a partir de la j tal que tiene algún elemento no nulo por
-- debajo de la j; es decir, si la submatriz de p obtenida eliminando
-- las i-1 primeras filas y las j-1 primeras columnas es no nula. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[3,2,5],[5,0,0],[6,0,0]]
--    ghci> existeColNoNulaDesde p 2 2
--    False
--    ghci> let q = listaMatriz [[3,2,5],[5,7,0],[6,0,0]]
--    ghci> existeColNoNulaDesde q 2 2
-- ---------------------------------------------------------------------

existeColNoNulaDesde :: Num a => Matriz a -> Int -> Int -> Bool
existeColNoNulaDesde p j i =
```

```
    or [not (null (elementosNoNulosColDesde p l i)) | l <- [j..n]]
    where n = numColumnas p

-- ---------------------------------------------------------------------
-- Ejercicio 32. Definir la función
--    menorIndiceColNoNulaDesde
--      :: Num a => Matriz a -> Int -> Int -> Maybe Int
-- tal que (menorIndiceColNoNulaDesde p j i) es el índice de la primera
-- columna, a partir de la j, en el que la matriz p tiene un elemento no
-- nulo a partir de la fila i. Por ejemplo,
--    ghci> let p = listaMatriz [[3,2,5],[5,7,0],[6,0,0]]
--    ghci> menorIndiceColNoNulaDesde p 2 2
--    Just 2
--    ghci> let q = listaMatriz [[3,2,5],[5,0,0],[6,0,2]]
--    ghci> menorIndiceColNoNulaDesde q 2 2
--    Just 3
--    ghci> let r = listaMatriz [[3,2,5],[5,0,0],[6,0,0]]
--    ghci> menorIndiceColNoNulaDesde r 2 2
--    Nothing
-- ---------------------------------------------------------------------

menorIndiceColNoNulaDesde :: (Num a) => Matriz a -> Int -> Int -> Maybe Int
menorIndiceColNoNulaDesde p j i
    | null js   = Nothing
    | otherwise = Just (head js)
    where n  = numColumnas p
          js = [j' | j' <- [j..n],
                     not (null (elementosNoNulosColDesde p j' i))]

-- ---------------------------------------------------------------------
-- Ejercicio 33. Definir la función
--    gaussAux :: Fractional a => Matriz a -> Int -> Int -> Matriz a
-- tal que (gauss p) es la matriz que en el que las i-1 primeras filas y
-- las j-1 primeras columnas son las de p y las restantes están
-- triangularizadas por el método de Gauss; es decir,
--    1. Si la dimensión de p es (i,j), entonces p.
--    2. Si la submatriz de p sin las i-1 primeras filas y las j-1
--       primeras columnas es nulas, entonces p.
--    3. En caso contrario, (gaussAux p' (i+1) (j+1)) siendo
--    3.1. j' la primera columna a partir de la j donde p tiene
```

```
--         algún elemento no nulo a partir de la fila i,
--    3.2. p1 la matriz obtenida intercambiando las columnas j y j'
--         de p,
--    3.3. i' la primera fila a partir de la i donde la columna j de
--         p1 tiene un elemento no nulo,
--    3.4. p2 la matriz obtenida intercambiando las filas i e i' de
--         la matriz p1 y
--    3.5. p' la matriz obtenida anulando todos los elementos de la
--         columna j de p2 por debajo de la fila i.
-- Por ejemplo,
--    ghci> let p = listaMatriz [[1.0,2,3],[1,2,4],[3,2,5]]
--    ghci> matrizLista (gaussAux p 2 2)
--    [[1.0,2.0,3.0],[1.0,2.0,4.0],[2.0,0.0,1.0]]
-- ----------------------------------------------------------------------

gaussAux :: Fractional a => Matriz a -> Int -> Int -> Matriz a
gaussAux p i j
    | dimension p == (i,j)             = p                          -- 1
    | not (existeColNoNulaDesde p j i) = p                          -- 2
    | otherwise                        = gaussAux p' (i+1) (j+1)    -- 3
    where Just j' = menorIndiceColNoNulaDesde p j i                 -- 3.1
          p1      = intercambiaColumnas j j' p                      -- 3.2
          Just i' = buscaIndiceDesde p1 j i                         -- 3.3
          p2      = intercambiaFilas i i' p1                        -- 3.4
          p'      = anulaColumnaDesde p2 j i                        -- 3.5

-- ----------------------------------------------------------------------
-- Ejercicio 34. Definir la función
--    gauss :: Fractional a => Matriz a -> Matriz a
-- tal que (gauss p) es la triangularización de la matriz p por el método
-- de Gauss. Por ejemplo,
--    ghci> let p = listaMatriz [[1.0,2,3],[1,2,4],[1,2,5]]
--    ghci> gauss p
--    array ((1,1),(3,3)) [((1,1),1.0),((1,2),3.0),((1,3),2.0),
--                         ((2,1),0.0),((2,2),1.0),((2,3),0.0),
--                         ((3,1),0.0),((3,2),0.0),((3,3),0.0)]
--    ghci> matrizLista (gauss p)
--    [[1.0,3.0,2.0],[0.0,1.0,0.0],[0.0,0.0,0.0]]
--    ghci> let p = listaMatriz [[3.0,2,3],[1,2,4],[1,2,5]]
--    ghci> matrizLista (gauss p)
```

```
--    [[3.0,2.0,3.0],[0.0,1.3333333333333335,3.0],[0.0,0.0,1.0]]
--    ghci> let p = listaMatriz [[3%1,2,3],[1,2,4],[1,2,5]]
--    ghci> matrizLista (gauss p)
--    [[3 % 1,2 % 1,3 % 1],[0 % 1,4 % 3,3 % 1],[0 % 1,0 % 1,1 % 1]]
--    ghci> let p = listaMatriz [[1.0,0,3],[1,0,4],[3,0,5]]
--    ghci> matrizLista (gauss p)
--    [[1.0,3.0,0.0],[0.0,1.0,0.0],[0.0,0.0,0.0]]
-- ---------------------------------------------------------------------

gauss :: Fractional a => Matriz a -> Matriz a
gauss p = gaussAux p 1 1

-- ---------------------------------------------------------------------
-- Determinante                                                        --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 35. Definir la función
--    determinante :: Fractional a => Matriz a -> a
-- tal que (determinante p) es el determinante de la matriz p. Por
-- ejemplo,
--    ghci> let p = listaMatriz [[1.0,2,3],[1,2,4],[1,2,5]]
--    ghci> determinante p
--    0.0
--    ghci> let p = listaMatriz [[1.0,2,3],[1,3,4],[1,2,5]]
--    ghci> determinante p
--    2.0
-- ---------------------------------------------------------------------

determinante :: Fractional a => Matriz a -> a
determinante p = product (elems (diagonalPral (gauss p)))
```

Relación 29

Implementación del TAD de los grafos mediante listas

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación es implementar el TAD de los grafos
-- mediante listas, de manera análoga a las implementaciones estudiadas
-- en el tema 22 que se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-22.pdf
-- y usando la mismas signatura.

-- ---------------------------------------------------------------------
-- Signatura                                                           --
-- ---------------------------------------------------------------------

module GrafoConListasDePares
    (Grafo,
     creaGrafo,  -- (Ix v,Num p) => Bool -> (v,v) -> [(v,v,p)] -> Grafo v p
     adyacentes, -- (Ix v,Num p) => Grafo v p -> v -> [v]
     nodos,      -- (Ix v,Num p) => Grafo v p -> [v]
     aristasND,  -- (Ix v,Num p) => Grafo v p -> [(v,v,p)]
     aristasD,   -- (Ix v,Num p) => Grafo v p -> [(v,v,p)]
     aristaEn,   -- (Ix v,Num p) => Grafo v p -> (v,v) -> Bool
     peso        -- (Ix v,Num p) => v -> v -> Grafo v p -> p
    ) where
```

```
-- ---------------------------------------------------------------------
-- Librerías auxiliares                                                --
-- ---------------------------------------------------------------------

import Data.Array
import Data.List

-- ---------------------------------------------------------------------
-- Representación de los grafos mediante listas                        --
-- ---------------------------------------------------------------------

-- (Grafo v p) es un grafo con vértices de tipo v y pesos de tipo p.
newtype Grafo v p = G [((v,v),p)]
    deriving (Eq, Show)

-- grafoL es el grafo
--             12
--        1 -------- 2
--        | \78     /|
--        |  \   32/ |
--        |   \   /  |
--      34|     5    |55
--        |   /   \  |
--        |  /44   \ |
--        | /     93\|
--        3 -------- 4
--             61
-- representado mediante un vector de adyacencia.
grafoL :: Grafo Int Int
grafoL = G [((1,2),12),((1,3),34),((1,5),78),
            ((2,1),12),((2,4),55),((2,5),32),
            ((3,1),34),((3,4),61),((3,5),44),
            ((4,2),55),((4,3),61),((4,5),93),
            ((5,1),78),((5,2),32),((5,3),44),((5,4),93)]

-- ---------------------------------------------------------------------
-- Ejercicios                                                          --
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
```

```
-- Ejercicio 1. Definir la función
--    creaGrafo :: (Ix v, Num p) => Bool -> (v,v) -> [(v,v,p)] -> Grafo v p
-- tal que (creaGrafo d cs as) es un grafo (dirigido si d es True y no
-- dirigido en caso contrario), con el par de cotas cs y listas de
-- aristas as (cada arista es un trío formado por los dos vértices y su
-- peso). Por ejemplo,
--    ghci> creaGrafo True (1,3) [(1,2,12),(1,3,34)]
--    G [((1,2),12),((1,3),34)]
--    ghci> creaGrafo False (1,3) [(1,2,12),(1,3,34)]
--    G [((1,2),12),((1,3),34),((2,1),12),((3,1),34)]
-- ---------------------------------------------------------------------

creaGrafo :: (Ix v, Num p) => Bool -> (v,v) -> [(v,v,p)] -> Grafo v p
creaGrafo d cs as =
    G ([((x1,x2),w) |
        (x1,x2,w) <- as] ++
        if d then []
        else [((x2,x1),w) | (x1,x2,w) <- as, x1 /= x2])

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    adyacentes :: (Ix v, Num p) => Grafo v p -> v -> [v]
-- tal que (adyacentes g v) es la lista de los vértices adyacentes al
-- nodo v en el grafo g. Por ejemplo,
--    adyacentes grafoL 4  ==  [2,3,5]
-- ---------------------------------------------------------------------

adyacentes :: (Ix v, Num p) => Grafo v p -> v -> [v]
adyacentes (G g) v = nub [u | ((w,u),_) <- g, w == v]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    nodos :: (Ix v,Num p) => Grafo v p -> [v]
-- tal que (nodos g) es la lista de todos los nodos del grafo g. Por
-- ejemplo,
--    nodos grafoL  ==  [1,2,3,4,5]
-- ---------------------------------------------------------------------

nodos :: (Ix v,Num p) => Grafo v p -> [v]
nodos (G g) = nub [x | ((x,_),_) <- g] ++ [y | ((_,y),_) <- g]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    aristaEn :: (Ix v,Num p) => Grafo v p -> (v,v) -> Bool
-- (aristaEn g a) se verifica si a es una arista del grafo g. Por
-- ejemplo,
--    aristaEn grafoL (5,1)  ==  True
--    aristaEn grafoL (4,1)  ==  False
-- ---------------------------------------------------------------------

aristaEn :: (Ix v,Num p) => Grafo v p -> (v,v) -> Bool
aristaEn g (x,y) = y `elem` adyacentes g x

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    peso :: (Ix v,Num p) => v -> v -> Grafo v p -> p
-- tal que (peso v1 v2 g) es el peso de la arista que une los vértices
-- v1 y v2 en el grafo g. Por ejemplo,
--    peso 1 5 grafoL  ==  78
-- ---------------------------------------------------------------------

peso :: (Ix v,Num p) => v -> v -> Grafo v p -> p
peso x y (G gs) = head [c | ((x',y'),c) <- gs, x==x', y==y']

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    aristasD :: (Ix v,Num p) => Grafo v p -> [(v,v,p)]
-- (aristasD g) es la lista de las aristas del grafo dirigido g. Por
-- ejemplo,
--    ghci> aristasD grafoL
--    [(1,2,12),(1,3,34),(1,5,78),
--     (2,1,12),(2,4,55),(2,5,32),
--     (3,1,34),(3,4,61),(3,5,44),
--     (4,2,55),(4,3,61),(4,5,93),
--     (5,1,78),(5,2,32),(5,3,44),(5,4,93)]
-- ---------------------------------------------------------------------

aristasD :: (Ix v,Num p) => Grafo v p -> [(v,v,p)]
aristasD (G g) = [(v1,v2,p) | ((v1,v2),p) <- g]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    aristasND :: (Ix v,Num p) => Grafo v p -> [(v,v,p)]
-- tal que (aristasND g) es la lista de las aristas del grafo no
-- dirigido g. Por ejemplo,
--    ghci> aristasND grafoL
--    [(1,2,12),(1,3,34),(1,5,78),
--     (2,4,55),(2,5,32),
--     (3,4,61),(3,5,44),
--     (4,5,93)]
-- ---------------------------------------------------------------------

aristasND :: (Ix v,Num p) => Grafo v p -> [(v,v,p)]
aristasND (G g) = [(v1,v2,p) | ((v1,v2),p) <- g, v1 < v2]
```

Relación 30

Problemas básicos con el TAD de los grafos

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación de ejercicios es definir funciones sobre
-- el TAD de los grafos, utilizando las implementaciones estudiadas
-- en el tema 22 que se pueden descargar desde
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/codigos.zip
--
-- Las transparencias del tema 22 se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-22.pdf

-- ---------------------------------------------------------------------
-- Importación de librerías                                            --
-- ---------------------------------------------------------------------

{-# LANGUAGE FlexibleInstances, TypeSynonymInstances #-}

import Data.Array
import Data.List (nub)
import Test.QuickCheck

-- Hay que seleccionar una implementación del TAD de los grafos
import GrafoConVectorDeAdyacencia
-- import GrafoConMatrizDeAdyacencia
```

```haskell
-- import GrafoConListas

-- ---------------------------------------------------------------------
-- Ejemplos                                                            --
-- ---------------------------------------------------------------------

-- Para los ejemplos se usarán los siguientes grafos.
g1, g2, g3, g4, g5, g6, g7, g8 :: Grafo Int Int
g1 = creaGrafo False (1,5) [(1,2,12),(1,3,34),(1,5,78),
                            (2,4,55),(2,5,32),
                            (3,4,61),(3,5,44),
                            (4,5,93)]
g2 = creaGrafo True (1,5) [(1,2,12),(1,3,34),(1,5,78),
                           (2,4,55),(2,5,32),
                           (4,3,61),(4,5,93)]
g3 = creaGrafo True (1,3) [(1,2,0),(2,2,0),(3,1,0),(3,2,0)]
g4 = creaGrafo True (1,4) [(1,2,3),(2,1,5)]
g5 = creaGrafo True (1,1) [(1,1,0)]
g6 = creaGrafo True (1,4) [(1,3,0),(3,1,0),(3,3,0),(4,2,0)]
g7 = creaGrafo False (1,4) [(1,3,0)]
g8 = creaGrafo True (1,5) [(1,1,0),(1,2,0),(1,3,0),(2,4,0),(3,1,0),
                           (4,1,0),(4,2,0),(4,4,0),(4,5,0)]

-- ---------------------------------------------------------------------
-- Ejercicio 1. El grafo completo de orden n, K(n), es un grafo no
-- dirigido cuyos conjunto de vértices es {1,..n} y tiene una arista
-- entre par de vértices distintos. Definir la función,
--    completo :: Int -> Grafo Int Int
-- tal que (completo n) es el grafo completo de orden n. Por ejemplo,
--    ghci> completo 4
--    array (1,4) [(1,[(2,0),(3,0),(4,0)]),
--                 (2,[(1,0),(3,0),(4,0)]),
--                 (3,[(1,0),(2,0),(4,0)]),
--                 (4,[(1,0),(2,0),(3,0)])]
-- ---------------------------------------------------------------------

completo :: Int -> Grafo Int Int
completo n = creaGrafo False (1,n) xs
    where xs = [(x,y,0) | x <- [1..n], y <- [1..n], x < y]
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 2. El ciclo de orden n, C(n), es un grafo no dirigido
-- cuyo conjunto de vértices es {1,...,n} y las aristas son
--    (1,2), (2,3), ..., (n-1,n), (n,1)
-- Definir la función
--    grafoCiclo :: Int -> Grafo Int Int
-- tal que (grafoCiclo n) es el grafo ciclo de orden n. Por ejemplo,
--    ghci> grafoCiclo 3
--    array (1,3) [(1,[(3,0),(2,0)]),(2,[(1,0),(3,0)]),(3,[(2,0),(1,0)])]
-- ---------------------------------------------------------------------

grafoCiclo :: Int -> Grafo Int Int
grafoCiclo n = creaGrafo False (1,n) xs
    where xs = [(x,x+1,0) | x <- [1..n-1]] ++ [(n,1,0)]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    nVertices :: (Ix v,Num p) => Grafo v p ->  Int
-- tal que (nVertices g) es el número de vértices del grafo g. Por
-- ejemplo,
--    nVertices (completo 4)  ==  4
--    nVertices (completo 5)  ==  5
-- ---------------------------------------------------------------------

nVertices :: (Ix v,Num p) => Grafo v p ->  Int
nVertices = length . nodos

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la función
--    dirigido :: (Ix v,Num p) => Grafo v p ->  Bool
-- tal que (dirigido g) se verifica si el grafo g es dirigido; es decir,
-- existe una arista (x,y) en g tal que que (y,x) no es una arista de
-- g. Por ejemplo,
--    dirigido g1            ==  False
--    dirigido g2            ==  True
--    dirigido (completo 4)  ==  False
-- ---------------------------------------------------------------------

dirigido :: (Ix v,Num p) => Grafo v p ->  Bool
dirigido g = or [not (aristaEn g (y,x)) |
```

```
                x <- vs, y <- vs, aristaEn g (x,y)]
    where vs = nodos g

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la función
--    noDirigido :: (Ix v,Num p) => Grafo v p ->  Bool
-- tal que (noDirigido g) se verifica si el grafo g es no dirigido. Por
-- ejemplo,
--    noDirigido g1            ==  True
--    noDirigido g2            ==  False
--    noDirigido (completo 4)  ==  True
-- ---------------------------------------------------------------------

noDirigido :: (Ix v,Num p) => Grafo v p ->  Bool
noDirigido = not . dirigido

-- ---------------------------------------------------------------------
-- Ejercicio 15. La función aristasND definida en el TAD de los grafos
-- se aplica a grafos no dirigidos sin lazos. Definir la función
--    aristasND' :: (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
-- tal que (aristasND' g) es la lista de las aristas del grafo no
-- dirigido g que puede tener lazos. Por ejemplo,
--    g5             ==  array (1,1) [(1,[(1,0)])]
--    aristasND g5   ==  []
--    aristasND' g5  ==  [(1,1,0)]
-- ---------------------------------------------------------------------

aristasND' :: (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
aristasND' g =
    [(v1,v2,peso v1 v2 g) | v1 <- nodos g, v2 <- contiguos g v1, v1 <= v2]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    aristas :: (Ix v,Num p) => Grafo v p ->  [(v,v)]
-- tal que (aristas g) es el conjunto de las aristas del grafo g. Por
-- ejemplo,
--    ghci> aristas g1
--    [(1,2),(1,3),(1,5),(2,4),(2,5),(3,4),(3,5),(4,5)]
--    ghci> aristas g2
--    [(1,2),(1,3),(1,5),(2,4),(2,5),(3,4),(4,5)]
```

```
--    ghci> aristas (completo 4)
--    [(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
--    ghci> aristas (creaGrafo True (1,3) [(2,1,0),(3,3,0)])
--    [(1,2),(3,3)]
--    ghci> aristas (creaGrafo True (1,3) [(2,1,0),(3,3,0),(1,2,0)])
--    [(1,2),(3,3)]
-- ---------------------------------------------------------------------

aristas :: (Ix v,Num p) => Grafo v p -> [(v,v,p)]
aristas g | dirigido g = aristasD g
          | otherwise  = aristasND' g

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    nAristas :: (Ix v,Num p) => Grafo v p ->  Int
-- tal que (nAristas g) es el número de aristas del grafo g. Por
-- ejemplo,
--    nAristas g1            ==  8
--    nAristas g2            ==  7
--    nAristas (completo 4)  ==  6
--    nAristas (completo 5)  ==  10
-- ---------------------------------------------------------------------

nAristas :: (Ix v,Num p) => Grafo v p ->  Int
nAristas = length . aristas

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    prop_nAristasCompleto :: Int -> Bool
-- tal que (prop_nAristasCompleto n) se verifica si el número de aristas
-- del grafo completo de orden n es n*(n-1)/2 y, usando la función,
-- comprobar que la propiedad se cumple para n de 1 a 20.
-- ---------------------------------------------------------------------

prop_nAristasCompleto :: Int -> Bool
prop_nAristasCompleto n =
    nAristas (completo n) == n*(n-1) `div` 2

-- La comprobación es
--    ghci> and [prop_nAristasCompleto n | n <- [1..20]]
```

```
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    lazos :: (Ix v,Num p) => Grafo v p -> [(v,v)]
-- tal que (lazos g) es el conjunto de los lazos (es decir, aristas
-- cuyos extremos son iguales) del grafo g. Por ejemplo,
--    ghci> lazos g3
--    [(2,2)]
--    ghci> lazos g2
--    []
-- ---------------------------------------------------------------------

lazos :: (Ix v,Num p) => Grafo v p -> [(v,v)]
lazos g = [(x,x) | x <- nodos g, aristaEn g (x,x)]

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    nLazos :: (Ix v,Num p) => Grafo v p ->  Int
-- tal que (nLazos g) es el número de lazos del grafo g. Por
-- ejemplo,
--    nLazos g3  ==  1
--    nLazos g2  ==  0
-- ---------------------------------------------------------------------

nLazos :: (Ix v,Num p) => Grafo v p ->  Int
nLazos = length . lazos

-- ---------------------------------------------------------------------
-- Ejercicio 9. En un un grafo g, los incidentes de un vértice v es el
-- conjuntos de vértices x de g para los que hay un arco (o una arista)
-- de x a v; es decir, que v es adyacente a x. Definir la función
--    incidentes :: (Ix v,Num p) => (Grafo v p) -> v -> [v]
-- tal que (incidentes g v) es la lista de los vértices incidentes en el
-- vértice v. Por ejemplo,
--    incidentes g2 5  ==  [1,2,4]
--    adyacentes g2 5  ==  []
--    incidentes g1 5  ==  [1,2,3,4]
--    adyacentes g1 5  ==  [1,2,3,4]
-- ---------------------------------------------------------------------
```

```haskell
incidentes :: (Ix v,Num p) => Grafo v p -> v -> [v]
incidentes g v = [x | x <- nodos g, v `elem` adyacentes g x]

-- ---------------------------------------------------------------------
-- Ejercicio 10. En un un grafo g, los contiguos de un vértice v es el
-- conjuntos de vértices x de g tales que x es adyacente o incidente con
-- v. Definir la función
--    contiguos :: (Ix v,Num p) => Grafo v p -> v -> [v]
-- tal que (contiguos g v) es el conjunto de los vértices de g contiguos
-- con el vértice v. Por ejemplo,
--    contiguos g2 5  ==  [1,2,4]
--    contiguos g1 5  ==  [1,2,3,4]
-- ---------------------------------------------------------------------

contiguos :: (Ix v,Num p) => Grafo v p -> v -> [v]
contiguos g v = nub (adyacentes g v ++ incidentes g v)

-- ---------------------------------------------------------------------
-- Ejercicio 11. El grado positivo de un vértice v de un grafo dirigido
-- g, es el número de vértices de g adyacentes con v. Definir la función
--    gradoPos :: (Ix v,Num p) => Grafo v p -> v -> Int
-- tal que (gradoPos g v) es el grado positivo del vértice v en el grafo
-- g. Por ejemplo,
--    gradoPos g1 5  ==  4
--    gradoPos g2 5  ==  0
--    gradoPos g2 1  ==  3
-- ---------------------------------------------------------------------

gradoPos :: (Ix v,Num p) => Grafo v p -> v -> Int
gradoPos g v = length (adyacentes g v)

-- ---------------------------------------------------------------------
-- Ejercicio 12. El grado negativo de un vértice v de un grafo dirigido
-- g, es el número de vértices de g incidentes con v. Definir la función
--    gradoNeg :: (Ix v,Num p) => Grafo v p -> v -> Int
-- tal que (gradoNeg g v) es el grado negativo del vértice v en el grafo
-- g. Por ejemplo,
--    gradoNeg g1 5  ==  4
--    gradoNeg g2 5  ==  3
```

```
--    gradoNeg g2 1  ==  0
-- ---------------------------------------------------------------------

gradoNeg :: (Ix v,Num p) => Grafo v p -> v -> Int
gradoNeg g v = length (incidentes g v)

-- ---------------------------------------------------------------------
-- Ejercicio 16. El grado de un vértice v de un grafo g, es el número de
-- aristas de g que contiene a v (los lazos se cuentan 2 veces). Definir
-- la función
--    grado :: (Ix v,Num p) => Grafo v p -> v -> Int
-- tal que (grado g v) es el grado del vértice v en el grafo g. Por
-- ejemplo,
--    grado g1 5  ==  4
--    grado g2 5  ==  3
--    grado g2 1  ==  3
--    grado g3 2  ==  4
--    grado g5 1  ==  2
-- ---------------------------------------------------------------------

grado :: (Ix v,Num p) => Grafo v p -> v -> Int
grado g v = sum [1 | (x,y,p) <- as, x == v] +
            sum [1 | (x,y,p) <- as, y == v]
    where as = aristas g

-- ---------------------------------------------------------------------
-- Ejercicio 17. Comprobar con QuickCheck que si g es un grafo dirigido,
-- entonces, para todo vértice v de g, el grado de v en g es la suma del
-- grado positivo y del grado negativo de v en g.
-- ---------------------------------------------------------------------

prop_grados :: Grafo Int Int -> Property
prop_grados g =
    dirigido g ==>
    and [grado g v == gradoPos g v + gradoNeg g v | v <- nodos g]

-- La comprobación es
--    ghci> quickCheck prop_grados
--    +++ OK, passed 100 tests.
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 18. Comprobar con QuickCheck que para cualquier grafo g, la
-- suma de los grados positivos de los vértices de g es igual que la
-- suma de los grados negativos de los vértices de g.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_sumaGrados:: Grafo Int Int -> Bool
prop_sumaGrados g =
    sum [gradoPos g v | v <- vs] == sum [gradoNeg g v | v <- vs]
    where vs = nodos g

-- La comprobación es
--    ghci> quickCheck prop_sumaGrados
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 19. Comprobar con QuickCheck que para cualquier grafo g, la
-- suma de los grados de los vértices de g es el doble del número de
-- aristas de g.
-- ---------------------------------------------------------------------

prop_sumaGradosDoble:: Grafo Int Int -> Bool
prop_sumaGradosDoble g =
    sum [grado g v | v <- nodos g] == 2 * nAristas g

-- La comprobación es
--    ghci> quickCheck prop_sumaGradosDoble
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 20. Comprobar con QuickCheck que en todo grafo, el número
-- de nodos de grado impar es par.
-- ---------------------------------------------------------------------

prop_numNodosGradoImpar :: Grafo Int Int -> Bool
prop_numNodosGradoImpar g = even m
    where vs = nodos g
          m = length [v | v <- vs, odd(grado g v)]
```

```haskell
-- La comprobación es
--    ghci> quickCheck prop_numNodosGradoImpar
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 21. Un grafo es regular si todos sus vértices tienen el
-- mismo grado. Definir la función
--    regular :: (Ix v,Num p) => Grafo v p -> Bool
-- tal que (regular g) se verifica si todos los nodos de g tienen el
-- mismo grado.
--    regular g1            ==  False
--    regular g2            ==  False
--    regular (completo 4)  ==  True
-- ---------------------------------------------------------------------

regular :: (Ix v,Num p) => Grafo v p -> Bool
regular g = and [grado g v == k | v <- vs]
    where vs = nodos g
          k  = grado g (head vs)

-- ---------------------------------------------------------------------
-- Ejercicio 22. Definir la propiedad
--    prop_CompletoRegular :: Int -> Int -> Bool
-- tal que (prop_CompletoRegular m n) se verifica si todos los grafos
-- completos desde el de orden m hasta el de orden m son regulares y
-- usarla para comprobar que todos los grafos completo desde el de orden
-- 1 hasta el de orden 30 son regulares.
-- ---------------------------------------------------------------------

prop_CompletoRegular :: Int -> Int -> Bool
prop_CompletoRegular m n = and [regular (completo x) | x <- [m..n]]

-- La comprobación es
--    ghci> prop_CompletoRegular 1 30
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 23. Un grafo es k-regular si todos sus vértices son de
-- grado k. Definir la función
--    regularidad :: (Ix v,Num p) => Grafo v p -> Maybe Int
```

```haskell
-- tal que (regularidad g) es la regularidad de g. Por ejemplo,
--    regularidad g1              ==  Nothing
--    regularidad (completo 4)    ==  Just 3
--    regularidad (completo 5)    ==  Just 4
--    regularidad (grafoCiclo 4)  ==  Just 2
--    regularidad (grafoCiclo 5)  ==  Just 2
-- ---------------------------------------------------------------------

regularidad :: (Ix v,Num p) => Grafo v p -> Maybe Int
regularidad g | regular g = Just (grado g (head (nodos g)))
              | otherwise = Nothing

-- ---------------------------------------------------------------------
-- Ejercicio 24. Definir la propiedad
--    prop_completoRegular :: Int -> Bool
-- tal que (prop_completoRegular n) se verifica si el grafo completo de
-- orden n es (n-1)-regular. Por ejemplo,
--    prop_completoRegular 5  ==  True
-- y usarla para comprobar que la cumplen todos los grafos completos
-- desde orden 1 hasta 20.
-- ---------------------------------------------------------------------

prop_completoRegular :: Int -> Bool
prop_completoRegular n =
   regularidad (completo n) == Just (n-1)

-- La comprobación es
--    ghci> and [prop_completoRegular n | n <- [1..20]]
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 25. Definir la propiedad
--    prop_cicloRegular :: Int -> Bool
-- tal que (prop_cicloRegular n) se verifica si el grafo ciclo de orden
-- n es 2-regular. Por ejemplo,
--    prop_cicloRegular 2  ==  True
-- y usarla para comprobar que la cumplen todos los grafos ciclos
-- desde orden 3 hasta 20.
-- ---------------------------------------------------------------------
```

```haskell
prop_cicloRegular :: Int -> Bool
prop_cicloRegular n =
   regularidad (grafoCiclo n) == Just 2

-- La comprobación es
--    ghci> and [prop_cicloRegular n | n <- [3..20]]
--    True

-- ---------------------------------------------------------------------
-- Generador de grafos                                                 --
-- ---------------------------------------------------------------------

instance Arbitrary (Grafo Int Int) where
    arbitrary = genG

genG :: Gen (Grafo Int Int)
genG = do d <- choose (True,False)
          n <- choose (1,10)
          xs <- vectorOf (n*n) arbitrary
          if d then return (generaGD n xs)
               else return (generaGND n xs)

generaGND :: Int -> [Int] -> Grafo Int Int
generaGND n ls  = creaGrafo False (1,n) l3
    where l1 = [(x,y) | x <-[1..n], y <- [1..n], x < y]
          l2 = zip l1 ls
          l3 = [(x,y,z) | ((x,y),z) <- l2, z > 0]

generaGD :: Int -> [Int] -> Grafo Int Int
generaGD n ls = creaGrafo True (1,n) l3
    where l1 = [(x,y) | x <-[1..n], y <- [1..n]]
          l2 = zip l1 ls
          l3 = [(x,y,z) | ((x,y),z) <- l2, z > 0]
```

Relación 31

Enumeraciones del conjunto de los números racionales

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación es construir dos enumeraciones de los
-- números racionales. Concretamente,
-- * una enumeración basada en las representaciones hiperbinarias y
-- * una enumeración basada en los los árboles de Calkin-Wilf.
-- También se incluye la comprobación de la igualdad de las dos
-- sucesiones y una forma alternativa de calcular el número de
-- representaciones hiperbinarias mediante la función fucs.
--
-- Esta relación se basa en los siguientes artículos:
-- * Gaussianos "Sorpresa sumando potencias de 2" http://goo.gl/AHdAG
-- * N. Calkin y H.S. Wilf "Recounting the rationals" http://goo.gl/gVZtW
-- * Wikipedia "Calkin-Wilf tree" http://goo.gl/cB3vn

-- ---------------------------------------------------------------------
-- Importación de librerías                                            --
-- ---------------------------------------------------------------------

import Data.List
import Test.QuickCheck

-- ---------------------------------------------------------------------
```

```haskell
-- Numeración de los racionales mediante representaciones hiperbinarias
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la constante
--    potenciasDeDos :: [Integer]
-- tal que potenciasDeDos es la lista de las potencias de 2. Por
-- ejemplo,
--    take 10 potenciasDeDos  ==  [1,2,4,8,16,32,64,128,256,512]
-- ---------------------------------------------------------------------

potenciasDeDos :: [Integer]
potenciasDeDos = [2^n | n <- [0..]]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    empiezaConDos :: Eq a => a -> [a] -> Bool
-- tal que (empiezaConDos x ys) se verifica si los dos primeros
-- elementos de ys son iguales a x. Por ejemplo,
--    empiezaConDos 5 [5,5,3,7]  ==  True
--    empiezaConDos 5 [5,3,5,7]  ==  False
--    empiezaConDos 5 [5,5,5,7]  ==  True
-- ---------------------------------------------------------------------

empiezaConDos x (y1:y2:ys) = y1 == x && y2 == x
empiezaConDos x (y1:y2:ys) = y1 == x && y2 == x
empiezaConDos x _          = False

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    representacionesHB :: Integer -> [[Integer]]
-- tal que (representacionesHB n) es la lista de las representaciones
-- hiperbinarias del número n como suma de potencias de 2 donde cada
-- sumando aparece como máximo 2 veces. Por ejemplo
--    representacionesHB 5  ==  [[1,2,2],[1,4]]
--    representacionesHB 6  ==  [[1,1,2,2],[1,1,4],[2,4]]
-- ---------------------------------------------------------------------

representacionesHB :: Integer -> [[Integer]]
representacionesHB n = representacionesHB' n potenciasDeDos
```

```
representacionesHB' n (x:xs)
    | n == 0    = [[]]
    | x == n    = [[x]]
    | x <  n    = [x:ys | ys <- representacionesHB' (n-x) (x:xs),
                          not (empiezaConDos x ys)] ++
                  representacionesHB' n xs
    | otherwise = []

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    nRepresentacionesHB :: Integer -> Integer
-- tal que (nRepresentacionesHB n) es el número de las representaciones
-- hiperbinarias del número n como suma de potencias de 2 donde cada
-- sumando aparece como máximo 2 veces. Por ejemplo,
--    ghci> [nRepresentacionesHB n | n <- [0..20]]
--    [1,1,2,1,3,2,3,1,4,3,5,2,5,3,4,1,5,4,7,3,8]
-- ---------------------------------------------------------------------

nRepresentacionesHB :: Integer -> Integer
nRepresentacionesHB = genericLength . representacionesHB

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    termino :: Integer -> (Integer,Integer)
-- tal que (termino n) es el par formado por el número de
-- representaciones hiperbinarias de n y de n+1 (que se interpreta como
-- su cociente). Por ejemplo,
--    termino 4  ==  (3,2)
-- ---------------------------------------------------------------------

termino :: Integer -> (Integer,Integer)
termino n = (nRepresentacionesHB n, nRepresentacionesHB (n+1))

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    sucesionHB :: [(Integer,Integer)]
-- sucesionHB es la la sucesión cuyo témino n-ésimo es (termino n); es
-- decir, el par formado por el número de representaciones hiperbinarias
-- de n y de n+1. Por ejemplo,
--    ghci> take 10 sucesionHB
```

```haskell
--    [(1,1),(1,2),(2,1),(1,3),(3,2),(2,3),(3,1),(1,4),(4,3),(3,5)]
-- ---------------------------------------------------------------------

sucesionHB :: [(Integer,Integer)]
sucesionHB = [termino n | n <- [0..]]

-- ---------------------------------------------------------------------
-- Ejercicio 7. Comprobar con QuickCheck que, para todo n,
-- (nRepresentacionesHB n) y  (nRepresentacionesHB (n+1)) son primos
-- entre sí.
-- ---------------------------------------------------------------------

prop_irreducibles :: Integer -> Property
prop_irreducibles n =
    n >= 0 ==>
    gcd (nRepresentacionesHB n) (nRepresentacionesHB (n+1)) == 1

-- La comprobación es
--    ghci> quickCheck prop_irreducibles
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 8. Comprobar con QuickCheck que todos los elementos de la
-- sucesionHB son distintos.
-- ---------------------------------------------------------------------

prop_distintos :: Integer -> Integer -> Bool
prop_distintos n m =
    termino n' /= termino m'
    where n' = abs n
          m' = n' + abs m + 1

-- La comprobación es
--    ghci> quickCheck prop_distintos
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    contenido :: Integer -> Integer -> Bool
-- tal que (contenido n) se verifica si la expresiones reducidas de
```

```
-- todas las fracciones x/y, con x e y entre 1 y n, pertenecen a la
-- sucesionHB. Por ejemplo,
--    contenidos 5  ==  True
-- ---------------------------------------------------------------------

contenido :: Integer -> Bool
contenido n =
    and [pertenece (reducida (x,y)) sucesionHB |
         x <- [1..n], y <- [1..n]]
    where pertenece x (y:ys) = x == y || pertenece x ys
          reducida (x,y) = (x `div` z, y `div` z)
              where z = gcd x y

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    indice :: (Integer,Integer) -> Integer
-- tal que (indice (a,b)) es el índice del par (a,b) en la sucesión de
-- los racionales. Por ejemplo,
--    indice (3,2)  ==  4
-- ---------------------------------------------------------------------

indice :: (Integer,Integer) -> Integer
indice (a,b) = head [n | (n,(x,y)) <- zip [0..] sucesionHB,
                         (x,y) == (a,b)]

-- ---------------------------------------------------------------------
-- Numeraciones mediante árboles de Calkin-Wilf                        --
-- ---------------------------------------------------------------------

-- El árbol de Calkin-Wilf es el árbol definido por las siguientes
-- reglas:
--    * El nodo raíz es el (1,1)
--    * Los hijos del nodo (x,y) son (x,x+y) y (x+y,y)
-- Por ejemplo, los 4 primeros niveles del árbol de Calkin-Wilf son
--                          (1,1)
--                            |
--               +------------+------------+
--               |                         |
--             (1,2)                     (2,1)
--               |                         |
```

```
--          +-----+-----+           +-----+-----+
--          |           |           |           |
--        (1,3)       (3,2)       (2,3)       (3,1)
--          |           |           |           |
--       +--+--+     +--+--+     +--+--+     +--+--+
--       |     |     |     |     |     |     |     |
--     (1,4) (4,3) (3,5) (5,2) (2,5) (5,3) (3,4) (4,1)

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--    sucesores :: (Integer,Integer) -> [(Integer,Integer)]
-- tal que (sucesores (x,y)) es la lista de los hijos del par (x,y) en
-- el árbol de Calkin-Wilf. Por ejemplo,
--    sucesores (3,2)  ==  [(3,5),(5,2)]
-- ---------------------------------------------------------------------

sucesores :: (Integer,Integer) -> [(Integer,Integer)]
sucesores (x,y) = [(x,x+y),(x+y,y)]

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    siguiente :: [(Integer,Integer)] -> [(Integer,Integer)]
-- tal que (siguiente xs) es la lista formada por los hijos de los
-- elementos de xs en el árbol de Calkin-Wilf. Por ejemplo,
--    ghci> siguiente [(1,3),(3,2),(2,3),(3,1)]
--    [(1,4),(4,3),(3,5),(5,2),(2,5),(5,3),(3,4),(4,1)]
-- ---------------------------------------------------------------------

siguiente :: [(Integer,Integer)] -> [(Integer,Integer)]
siguiente xs = [p | x <- xs, p <- sucesores x]

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la constante
--    nivelesCalkinWilf:: [[(Integer,Integer)]]
-- tal que nivelesCalkinWilf es la lista de los niveles del árbol de
-- Calkin-Wilf. Por ejemplo,
--    ghci> take 4 nivelesCalkinWilf
--    [[(1,1)],
--     [(1,2),(2,1)],
--     [(1,3),(3,2),(2,3),(3,1)],
```

```
--     [(1,4),(4,3),(3,5),(5,2),(2,5),(5,3),(3,4),(4,1)]]
-- ---------------------------------------------------------------------

nivelesCalkinWilf:: [[(Integer,Integer)]]
nivelesCalkinWilf = iterate siguiente [(1,1)]

-- ---------------------------------------------------------------------
-- Ejercicio 14. Definir la constante
--    sucesionCalkinWilf :: [(Integer,Integer)]
-- tal que sucesionCalkinWilf es la lista correspondiente al recorrido
-- en anchura del árbol de Calkin-Wilf. Por ejemplo,
--    ghci> take 10 sucesionCalkinWilf
--    [(1,1),(1,2),(2,1),(1,3),(3,2),(2,3),(3,1),(1,4),(4,3),(3,5)]
-- ---------------------------------------------------------------------

sucesionCalkinWilf :: [(Integer,Integer)]
sucesionCalkinWilf = concat nivelesCalkinWilf

-- ---------------------------------------------------------------------
-- Ejercicio 15. Definir la función
--    igual_sucesion_HB_CalkinWilf :: Int -> Bool
-- tal que (igual_sucesion_HB_CalkinWilf n) se verifica si los n
-- primeros términos de la sucesión HB son iguales que los de la
-- sucesión de Calkin-Wilf. Por ejemplo,
--    igual_sucesion_HB_CalkinWilf 20  ==  True
-- ---------------------------------------------------------------------

igual_sucesion_HB_CalkinWilf :: Int -> Bool
igual_sucesion_HB_CalkinWilf n =
    take n sucesionCalkinWilf == take n sucesionHB

-- ---------------------------------------------------------------------
-- Número de representaciones hiperbinarias mediante la función fusc
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 16. Definir la función
--    fusc :: Integer -> Integer
-- tal que
--    fusc(0)    = 1
```

```haskell
--    fusc(2n+1) = fusc(n)
--    fusc(2n+2) = fusc(n+1)+fusc(n)
-- Por ejemplo,
--    fusc 4  ==  3
-- ---------------------------------------------------------------------

fusc :: Integer -> Integer
fusc 0 = 1
fusc  n | odd n     = fusc ((n-1) `div` 2)
        | otherwise = fusc(m+1) + fusc m
                      where m = (n-2) `div` 2

-- ---------------------------------------------------------------------
-- Ejercicio 17. Comprobar con QuickCheck que, para todo n, (fusc n) es
-- el número de las representaciones hiperbinarias del número n como
-- suma de potencias de 2 donde cada sumando aparece como máximo 2
-- veces; es decir, que las funciones fusc y nRepresentacionesHB son
-- equivalentes.
-- ---------------------------------------------------------------------

prop_fusc :: Integer -> Bool
prop_fusc n = nRepresentacionesHB n' == fusc n'
              where n' = abs n

-- La comprobación es
--    ghci> quickCheck prop_fusc
--    +++ OK, passed 100 tests.
```

Relación 32

El problema del granjero mediante búsqueda en espacio de estado

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- Un granjero está parado en un lado del río y con él tiene un lobo,
-- una cabra y una repollo. En el río hay un barco pequeño. El granjero
-- desea cruzar el río con sus tres posesiones. No hay puentes y en el
-- barco hay solamente sitio para el granjero y un artículo. Si deja
-- la cabra con la repollo sola en un lado del río la cabra comerá la
-- repollo. Si deja el lobo y la cabra en un lado, el lobo se comerá a
-- la cabra. ¿Cómo puede cruzar el granjero el río con los tres
-- artículos, sin que ninguno se coma al otro?
--
-- El objetivo de esta relación de ejercicios es resolver el problema
-- del granjero mediante búsqueda en espacio de estados, utilizando las
-- implementaciones estudiadas en el tema 23 que se pueden descargar desde
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/codigos.zip
--
-- Las transparencias del tema 23 se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-23.pdf

-- ---------------------------------------------------------------------
-- Importaciones                                                       --
```

```haskell
-- ---------------------------------------------------------------------

import BusquedaEnEspaciosDeEstados

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir el tipo Orilla con dos constructores I y D que
-- representan las orillas izquierda y derecha, respectivamente.
-- ---------------------------------------------------------------------

data Orilla = I | D
              deriving (Eq, Show)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir el tipo Estado como abreviatura de una tupla que
-- representan en qué orilla se encuentra cada uno de los elementos
-- (granjero, lobo, cabra, repollo). Por ejemplo, (I,D,D,I) representa
-- que el granjero está en la izquierda, que el lobo está en la derecha,
-- que la cabra está en la derecha y el repollo está en la izquierda.
-- ---------------------------------------------------------------------

type Estado = (Orilla,Orilla,Orilla,Orilla)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir
--    inicial:: Estado
-- tal que inicial representa el estado en el que todos están en la
-- orilla izquierda.
-- ---------------------------------------------------------------------

inicial:: Estado
inicial = (I,I,I,I)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir
--    final:: Estado
-- tal que final representa el estado en el que todos están en la
-- orilla derecha.
-- ---------------------------------------------------------------------

final:: Estado
```

```haskell
final = (D,D,D,D)

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    seguro :: Estado -> Bool
-- tal que (seguro e) se verifica si el estado e es seguro; es decir,
-- que no puede estar en una orilla el lobo con la cabra sin el granjero
-- ni la cabra con el repollo sin el granjero. Por ejemplo,
--    seguro (I,D,D,I)  ==  False
--    seguro (D,D,D,I)  ==  True
--    seguro (D,D,I,I)  ==  False
--    seguro (I,D,I,I)  ==  True
-- ---------------------------------------------------------------------

seguro :: Estado -> Bool
seguro (g,l,c,r)
       | l == c    = g == l
       | c == r    = g == c
       | otherwise = True

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    opuesta :: Orilla -> Orilla
-- tal que (opuesta x) es la opuesta de la orilla x. Por ejemplo
--    opuesta I = D
-- ---------------------------------------------------------------------

opuesta :: Orilla -> Orilla
opuesta I = D
opuesta D = I

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    sucesoresE :: Estado -> [Estado]
-- tal que (sucesoresE e) es la lista de los sucesores seguros del
-- estado e. Por ejemplo,
--    sucesoresE (D,I,D,I)  ==  [(I,I,D,I),(I,I,I,I)]
-- ---------------------------------------------------------------------

sucesoresE :: Estado -> [Estado]
```

```
sucesoresE e = [mov e | mov <- [m1,m2,m3,m4], seguro (mov e)]
    where m1 (g,l,c,r) = (opuesta g, l, c, r)
          m2 (g,l,c,r) = (opuesta g, opuesta l, c, r)
          m3 (g,l,c,r) = (opuesta g, l, opuesta c, r)
          m4 (g,l,c,r) = (opuesta g, l, c, opuesta r)

-- ---------------------------------------------------------------------
-- Ejercicio 8. Los nodos del espacio de búsqueda son lista de estados
--    [e_n, ..., e_2, e_1]
-- donde e_1 es el estado inicial y para cada i (2 <= i <= n), e_i es un
-- sucesor de e_(i-1). Definir el tipo de datos NodoRio para representar
-- los nodos del espacio de búsqueda. Por ejemplo,
--    ghci> :type (Nodo [(I,I,D,I),(I,I,I,I)])
--    (Nodo [(I,I,D,I),(I,I,I,I)]) :: NodoRio
-- ---------------------------------------------------------------------

data NodoRio = Nodo [Estado]
               deriving (Eq, Show)

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    sucesoresN :: NodoRio -> [NodoRio]
-- tal que (sucesoresN n) es la lista de los sucesores del nodo n. Por
-- ejemplo,
--    ghci> sucesoresN (Nodo [(I,I,D,I),(D,I,D,I),(I,I,I,I)])
--    [Nodo [(D,D,D,I),(I,I,D,I),(D,I,D,I),(I,I,I,I)],
--     Nodo [(D,I,D,D),(I,I,D,I),(D,I,D,I),(I,I,I,I)]]
-- ---------------------------------------------------------------------

sucesoresN :: NodoRio -> [NodoRio]
sucesoresN (Nodo (n@(e:es))) =
    [Nodo (e':n) | e' <- sucesoresE e, notElem e' es]

-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    esFinal:: NodoRio -> Bool
-- tal que (esFinal n) se verifica si n es un nodo final; es decir, su
-- primer elemento es el estado final. Por ejemplo,
--    esFinal (Nodo [(D,D,D,D),(I,I,I,I)])  ==  True
--    esFinal (Nodo [(I,I,D,I),(I,I,I,I)])  ==  False
```

```haskell
-- ---------------------------------------------------------------------

esFinal:: NodoRio -> Bool
esFinal (Nodo (n:_)) = n == final

-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--    granjeroEE :: [NodoRio]
-- tal que granjeroEE son las soluciones del problema del granjero
-- mediante el patrón de búsqueda en espacio de estados. Por ejemplo,
--    ghci> head granjeroEE
--    Nodo [(D,D,D,D),(I,D,I,D),(D,D,I,D),(I,D,I,I),
--          (D,D,D,I),(I,I,D,I),(D,I,D,I),(I,I,I,I)]
-- ---------------------------------------------------------------------

granjeroEE :: [NodoRio]
granjeroEE = buscaEE sucesoresN
                     esFinal
                     (Nodo [inicial])
```

Relación 33

División y factorización de polinomios mediante la regla de Ruffini

```
-- ---------------------------------------------------------------------
-- Introducción                                                        --
-- ---------------------------------------------------------------------

-- El objetivo de esta relación de ejercicios es implementar la regla de
-- Ruffini y sus aplicaciones utilizando las implementaciones del TAD de
-- polinomio estudiadas en el tema 21 que se pueden descargar desde
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/codigos.zip
--
-- Las transparencias del tema 21 se encuentran en
--    http://www.cs.us.es/~jalonso/cursos/i1m-10/temas/tema-21.pdf

-- ---------------------------------------------------------------------
-- Importación de librerías                                            --
-- ---------------------------------------------------------------------

import PolOperaciones
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejemplos                                                            --
-- ---------------------------------------------------------------------
```

```haskell
-- Además de los ejemplos de polinomios (ejPol1, ejPol2 y ejPol3) que se
-- encuentran en PolOperaciones, usaremos el siguiente ejemplo.
ejPol4 :: Polinomio Int
ejPol4 = consPol 3 1
                 (consPol 2 2
                          (consPol 1 (-1)
                                   (consPol 0 (-2) polCero)))

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    divisores :: Int -> [Int]
-- tal que (divisores n) es la lista de todos los divisores enteros de
-- n. Por ejemplo,
--    divisores 4 == [1,2,4,-1,-2,-4]
-- ---------------------------------------------------------------------

divisores :: Int -> [Int]
divisores n = l1 ++ l2
    where l1 = [x | x <- [1..n], rem n x == 0]
          l2 = [-x | x <- l1]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    coeficiente :: Num a => Int -> Polinomio a -> a
-- tal que (coeficiente k p) es el coeficiente del término de grado k en
-- p. Por ejemplo:
--     coeficiente 4 ejPol1 == 3
--     coeficiente 3 ejPol1 == 0
--     coeficiente 2 ejPol1 == -5
--     coeficiente 5 ejPol1 == 0
-- ---------------------------------------------------------------------

coeficiente :: Num a => Int -> Polinomio a -> a
coeficiente k p | k == n                 = coefLider p
                | k > grado (restoPol p) = 0
                | otherwise              = coeficiente k (restoPol p)
                 where n = grado p

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
```

```haskell
--    terminoIndep :: Num a => Polinomio  a -> a
-- tal que (terminoIndep p) es el término independiente del polinomio
-- p. Por ejemplo,
--    terminoIndep ejPol1 == 3
--    terminoIndep ejPol2 == 0
--    terminoIndep ejPol4 == -2
-- ---------------------------------------------------------------------

terminoIndep :: Num a => Polinomio  a -> a
terminoIndep p = coeficiente 0 p

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    coeficientes :: Num a => Polinomio a -> [a]
-- tal que (coeficientes p) es la lista de coeficientes de p, ordenada
-- según el grado. Por ejemplo,
--     coeficientes ejPol1 == [3,0,-5,0,3]
--     coeficientes ejPol4 == [1,2,-1,-2]
--     coeficientes ejPol2 == [1,0,0,5,4,0]
-- ---------------------------------------------------------------------

coeficientes :: Num a => Polinomio a -> [a]
coeficientes p = [coeficiente k p | k <-[n,n-1..0]]
    where n = grado p

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    creaPol :: Num a => [a] -> Polinomio a
-- tal que (creaPol cs) es el polinomio cuya lista de coeficientes es
-- cs. Por ejemplo,
--     creaPol [1,0,0,5,4,0] == x^5 + 5*x^2 + 4*x
--     creaPol [1,2,0,3,0]   == x^4 + 2*x^3 + 3*x
-- ---------------------------------------------------------------------

creaPol :: Num a => [a] -> Polinomio a
creaPol []     = polCero
creaPol (a:as) = consPol n a (creaPol as)
    where n = length as

-- ---------------------------------------------------------------------
```

```haskell
-- Ejercicio 6. Comprobar con QuickCheck que, dado un polinomio p, el
-- polinomio obtenido mediante creaPol a partir de la lista de
-- coeficientes de p coincide con p.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_coef:: Polinomio Int -> Bool
prop_coef p =
    creaPol (coeficientes p) == p

-- La comprobación es
--    ghci> quickCheck prop_coef
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir una función
--    pRuffini:: Int -> [Int] -> [Int]
-- tal que (pRuffini r cs) es la lista que resulta de aplicar un paso
-- del regla de Ruffini al número entero r y a la lista de coeficientes
-- cs. Por ejemplo,
--    pRuffini 2 [1,2,-1,-2] == [1,4,7,12]
--    pRuffini 1 [1,2,-1,-2] == [1,3,2,0]
-- ya que
--      | 1  2  -1  -2          | 1  2  -1  -2
--    2 |    2   8  14        1 |    1   3   2
--    --+--------------       --+-------------
--      | 1  4   7  12          | 1  3   2   0
-- ---------------------------------------------------------------------

pRuffini :: Int -> [Int] -> [Int]
pRuffini r p@(c:cs) =
    c : [x+r*y | (x,y) <- zip cs (pRuffini r p)]

-- Otra forma:
pRuffini' :: Int -> [Int] -> [Int]
pRuffini' r = scanl1 (\s x -> s * r + x)

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    cocienteRuffini:: Int -> Polinomio Int -> Polinomio Int
```

```haskell
-- tal que (cocienteRuffini r p) es el cociente de dividir el polinomio
-- p por el polinomio x-r. Por ejemplo:
--     cocienteRuffini 2 ejPol4    == x^2 + 4*x + 7
--     cocienteRuffini (-2) ejPol4 == x^2 + -1
--     cocienteRuffini 3 ejPol4    == x^2 + 5*x + 14
-- ---------------------------------------------------------------------

cocienteRuffini :: Int -> Polinomio Int -> Polinomio Int
cocienteRuffini r p = creaPol (init (pRuffini r (coeficientes p)))

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    restoRuffini:: Int -> Polinomio Int -> Int
-- tal que (restoRuffini r p) es el resto de dividir el polinomio p por
-- el polinomio x-r. Por ejemplo,
--     restoRuffini 2 ejPol4    == 12
--     restoRuffini (-2) ejPol4 == 0
--     restoRuffini 3 ejPol4    == 40
-- ---------------------------------------------------------------------

restoRuffini :: Int -> Polinomio Int -> Int
restoRuffini r p = last (pRuffini r (coeficientes p))

-- ---------------------------------------------------------------------
-- Ejercicio 10. Comprobar con QuickCheck que, dado un polinomio p y un
-- número entero r, las funciones anteriores verifican la propiedad de
-- la división euclídea.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_diviEuclidea:: Int -> Polinomio Int -> Bool
prop_diviEuclidea r p =
    p == sumaPol (multPol coc div) res
        where coc = cocienteRuffini r p
              div = creaPol [1,-r]
              res = creaTermino 0 (restoRuffini r p)

-- La comprobación es
--    ghci> quickCheck prop_diviEuclidea
--    +++ OK, passed 100 tests.
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 11. Definir la función
--     esRaizRuffini:: Int -> Polinomio Int -> Bool
-- tal que (esRaizRuffini r p) se verifica si r es una raiz de p, usando
-- para ello el regla de Ruffini. Por ejemplo,
--     esRaizRuffini 0 ejPol3 == True
--     esRaizRuffini 1 ejPol3 == False
-- ---------------------------------------------------------------------

esRaizRuffini:: Int -> Polinomio Int -> Bool
esRaizRuffini r p = restoRuffini r p == 0

-- ---------------------------------------------------------------------
-- Ejercicio 12. Definir la función
--    raicesRuffini :: Polinomio Int -> [Int]
-- tal que (raicesRuffini p) es la lista de las raices enteras de p,
-- calculadas usando el regla de Ruffini. Por ejemplo,
--     raicesRuffini ejPol1 == []
--     raicesRuffini ejPol2 == [0]
--     raicesRuffini ejPol3 == [0]
--     raicesRuffini ejPol4 == [-2,-1,1]
-- ---------------------------------------------------------------------

raicesRuffini :: Polinomio Int -> [Int]
raicesRuffini p =
    aux (coeficientes p) (0:divisores (abs (terminoIndep p))) []
    where aux _ [] l3  = l3
          aux l1 l2@(r:rs) l3 | b == 0    = aux l1' l2 (r:l3)
                              | otherwise = aux l1 rs l3
              where ls = pRuffini r l1
                    l1' = init ls
                    b = last ls

-- ---------------------------------------------------------------------
-- Ejercicio 13. Definir la función
--    factorizacion :: Polinomio Int -> [Polinomio Int]
-- tal que (factorizacion p) es la lista de la descomposición del
-- polinomio p en factores obtenida mediante el regla de Ruffini. Por
-- ejemplo,
```

```haskell
--    ghci> factorizacion (creaPol [1,0,0,0,-1])
--    [x^2 + 1,1*x + 1,1*x + -1]
-- ----------------------------------------------------------------------

factorizacion :: Polinomio Int -> [Polinomio Int]
factorizacion p | esPolCero p = [p]
factorizacion p =
    aux (coeficientes p) (0:divisores (abs (terminoIndep p))) []
    where aux l1 [] l3  = (creaPol l1) : l3
          aux l1 l2@(r:rs) l3
              | b == 0    = aux l1' l2 ((creaPol [1,-r]):l3)
              | otherwise = aux l1 rs l3
              where ls = pRuffini r l1
                    l1' = init ls
                    b = last ls
```